NEWSLETTER FOR AM-100 USER'S GROUP   Produced by   Lefford F. Lowden
616 Long Pond Road
Rochester, New York 14612

1 OCTOBER 1979   Number 2.4

DISCUSSION OF AMOS (TM) MONITOR, PART 1

by Peter A. Jacobson

The Alpha Micro system monitor (SYSTEM.MON[1,4]) is logically divided into seven modules. These modules (in the order in which they fall in memory) are:

SYSMON - base monitor

TRMSER - terminal service routines

FILSER - file service routines

FILERR - file error message routines

EXEC - executive program module

DSKSER - disk driver routines for device DSKn:

INITIA - the system initialization routine

SYSMON and EXEC are my own names for those modules. The following discussion of SYSMON reflects the monitor as of the 4.2 release.

The thirty-one 16 bit words addressed from 0 to 76 octal are reserved locations as defined in the WD-16 Programmer's Reference Manual that comes with every AM-100 (TM) board. A brief summary of these word locations and their functions follow (the reader is advised to refer to the manual for a full description of the definitions):

HEX	OCTAL	FUNCTION
00-10	00-20	R0-R5, SP, PC, and PS are saved or fetched for halt or power up (not implemented on the AM-100)

The following locations contain the value that is placed in PC when the described event occurs:

12	22	buss error
14	24	nonvectored interrupt powerfail
16	26	power up/halt option power restore
18	30	parity error
1A	32	reserved op code
1C	34	illegal op code format
1E	36	XCT (execute single instruction) error
20	40	XCT trap
22	42	SVCA table (see note 1 below)
24	44	SVCB
26	46	SVCC
28	50	vectored interrupt table (see note 2 below)
2A	52	nonvectored interrupt
2C	54	BPT (breakpoint) trap
2E	56	I/O priority mask
30-38	60-70	floating point operation storage
3A-3C	72-74	not used
3E	76	floating point error PC

Note 1: content of octal 42 plus twice the value of the argument is placed in PC. The content of the word thus addressed by PC is added to PC to get the final destination address.

Note 2: content of octal 50 plus the device code is placed in PC. The content of the word addressed by PC is added to PC to get the final destination address.

Once the monitor is loaded, either by the disk controller board's PROM routine, MONTST.PRG, or a bootstrap loader (eg HWKLOD, WNGLOD, etc.), a "CLR PC" is executed which sets the program counter to memory location 0. Initially that location contains the instruction "JMP @#31572", which in the 4.2 release of the monitor, is the address of the initialization routine, INITIA, which will be discussed later.

In that there are no SVCC's currently implemented on the system, location 46 octal (the SVCC PC) contains zero. Therefore, if an SVCC ever gets called, the program counter will be set to location 0. After the first clock interrupt, location 0 no longer contains the JMP instruction. Part of the job scheduler's task is to update the arrow display for the DYSTAT program. This is done by moving octal 15 to the memory location contained in the JOBDYS word of each job's JCB. If DYSTAT is not running on the system, the JOBDYS address is 0 and the original JMP instruction gets modified into an LEA R4,@R5. When an SVCC is executed the AM-100 starts executing garbage and the system crashes.

The next section of the monitor, starting at octal 100 is the system communication area as described in appendix B of the "AMOS MONITOR CALLS MANUAL". In addition to that description, the following information is known about selected words in that area:

SYSTEM - if bit 8 is set, the system is running from a cartridge disk.

```
DEVTBL - points to a contiguous area of memory.  The table is built by JOBS
            during system initialization.  Each entry has four words:
         Word 0 - device status (odd byte) and drive number (even byte)
                 status byte:
                    bit 0 is always set so that word 0 is never zero
                    bit 1  set ==> sharable
                    bit 2  set ==> non-sharable device is assigned (set by ASSIGN)
                    bit 3  set ==> same as bit 2, but it is not known how it is set
                    bit 4  set ==> mounted
         Word 1 - device name (packed RAD50)
         Word 2 - JCB address if device is assigned.  If address is odd, device
                     is assigned to "COMPUTER B"
         Word 3 - device bitmap address (if 0, device is not file structured)
                     address inserted here by BITMAP during system initialization
         Last word of the table is zero
```

CLKQUE, SCNQUE, and RUNQUE are discussed later with the job scheduler.

DRVTRK - the table initially contains -1's.

Following the system communication area is the vectored interrupt (I/O) table and an illegal interrupt trap routine. There are eight entries in the table, which initially are offsets to the trap routine.

The internal stack is 100 octal words in size and is used as a work stack by the job scheduler.

The SVCA and SVCB offset tables are the next area in the monitor. The execution of an SVCA is described in the WD16 manual. There are no undocumented SVCA's, however, there appears to be an error in those copies of SYS.MAC that I have seen. The BNKSWP call is defined as an SVCA 46, but its table entry in the monitor makes it an SVCA 45.

SVCB's are processed with a routine which follows their offset table. This routine decodes the PSI following the SVCB opcode and places the address of the first argument in R4 and the address of the second argument in R3. The function code is placed in R0 and the PC address is adjusted up to three words past the SVCB. The offset table is entered with a TCALL R5, which contains twice the value of the SVCB argument. If the SVCB uses an extra argument, as SVCB 0, function code octal 14 (DSKALC through DSKCTG) does, it is the responsibility of the final destination routine to adjust PC past this word. There are no undocumented SVCB's.

The monitor error trap routines follow the SVCB processing module. The first of these is the ubiquitous "BUSS ERROR - PC nnnn". The error control intercept words of the JCB are first tested for user error recovery. If these words are null (they are always reset to null by EXIT - SVCA 11), a system error message is generated. A "BUSS ERROR" is reported for buss, breakpoint, and floating point errors. There are no SVCA 0 or SVCA 1 calls defined, and they are trapped out, however, they will be erroneously reported as "??SVCA 1 CALLED" or "??SVCA 2 CALLED".

The queue system routines and the initial twenty queue blocks form the next section of the monitor. The queue calls are fully described in chapter five of the "AMOS MONITOR CALLS MANUAL". No test is made by any queue call to determine if there are any queue blocks available. If a particular system installation makes heavy use of the queue system, it is recommended that QFREE be tested before a queue call is executed.

The job scheduler follows the queue system. It is entered whenever a nonvectored interrupt (I1) is generated. This interrupt is the line clock. The job scheduler first executes a SAVE (PS and PC were pushed on the stack when the interrupt occured), switches over to the internal stack, and then increments

TIME. CLKQUE entries are processed every clock tick by first picking up the address of CLKQUE which links to the first queue entry to be processed and the first word of the queue entry links to the next, etc. The second word of the queue entry contains the address of the routine to be executed and the remaining six words can be used for the routine's data (after the routine is called, R3 will contain the address of the third word of the queue entry). The routine must exit with an RTN. If the "V" bit (overflow bit in the status information) is set (via an LCC 2 or other operation which sets this bit) on return from the routine, the entry is deleted from the CLKQUE by returning the queue block to the QFREE list and relinking the rest of the queue. The next entry is processed by picking up its link from the previous entry. Examples of CLKQUE entries are the SLEEP call (see below), HLDTIM, and DYSTAT (which is never descheduled). CLKQUE entries are added with a QINS call and placing the address of the routine in the second word of the queue block.

When the end of the CLKQUE is reached, the initial entry will be processed. The last entry is two words long; the first contains the link to SCNQUE and the second contains the address of an RTN instruction (allowing simulation of an actual routine).

SCNQUE entries are processed by the same routine that handles CLKQUE entries and the format is identical. The job scheduler does not make a distinction between them at this point. When the end of the SCNQUE is reached, its link points to the RUNQUE.

The RUNQUE is five words in length; its initial entries are as follows:

```
RUNQUE: WORD    RUNQUE+6                ; link to current job's run address
        WORD    1004                    ; address of a RTN instruction
        WORD    RUNQUE                  ; link to next job's run address
        WORD    0                       ; last link
        WORD    2476                    ; end of the job scheduler
```

The RTN instruction (RUNQUE+2) will always be executed when the RUNQUE is entered (this provides for an initial simulation of a CLKQUE or SCNQUE entry). If any jobs are scheduled (see JRUN for scheduling information) the first RUNQUE word will contain the address of the fourth word of the currently scheduled job's JOBRNQ. This address is the run address for the job when it gets scheduled and will be either the entry point of the job context switching routine or the job priority timer routine. If the address is the first one, the job's SP is restored from JOBRNQ+14, the job's priority is copied into the timer word and incremented (to insure the job doesn't get 65535 ticks of CPU time), JOBCUR is updated, the address of the priority timer routine is placed in the fourth word of JOBRNQ, the DYSTAT arrow is sent, and memory bank switching (if active) is carried out. An RRTT then sends the job off to continue whatever it was doing before being interrupted.

When the priority timer routine is entered (either via an interrupt or JWAIT - see below), the time counter word(JOBRNQ+10) is decremented and, if non-zero, the job is allowed to continue. If the job has used all its allotted time and there are other jobs in the RUNQUE, a new job is scheduled (if no other jobs are scheduled, the job is allowed to continue). Before scheduling the next job, the current job's SP is saved, the context switching routine address is placed in the job's run address word, and the DYSTAT arrow is cleared. The link to the next job is loaded from RUNQUE+4. This is somewhat oversimplified as the RUNQUE linkage appears to be circular, but that is the general idea.

The initial link defines the end of the RUNQUE. If all scheduled jobs are comleted (and descheduled) within the clock tick (or no jobs were scheduled) the last link is reached. This link is to a section of the job shceduler that insures interrupts are enabled (so another clock interrupt will occur) and executes an SOB 400 times (presumably waiting for an interrupt). If a clock interrupt does not occur after the SOB argument reaches 0, the processor is locked and the SCNQUE entries are processed again.

After the job scheduler is the BNKSWP routine which will swap memory banks for the job in control of the CPU. The job scheduler does not use this call, but rather, does it's own swapping. BNKSWP will not update the JOBBNK word in the JCB and if the bank argument passed in R1 does not exist, the job will either be stuck here forever or return the wrong or non-existant memory bank.

Although JWAIT occupies the next block of memory, discussion of this call is postponed until after JRUN so that the scheduling of a job can be done first.

The JRUN call first locks the processor and then picks up the flag argument following the call (when the routine returns, the PC is adjusted past this argument). If none of the flag argument bits are set in the current JOBSTS word, the routine returns. If any bits are set, they are cleared from the JOBSTS word (if the flag argument is zero, it is a special case, and the bit test is bypassed). The JOBRNQ words are defined below to aid in the following discussion on job scheduling.

JOBRNQ (seven word block):

WORD 0 - unknown / always 0
+2 WORD 1 - link to last scheduled job
+4 WORD 2 - link to next scheduled job
+6 WORD 3 - job's run address
+10 WORD 4 - priority counter
+12 WORD 5 - job priority
+14 WORD 6 - current SP

The link to the next scheduled job (JOBRNQ+4) is tested and, if it is non-zero, the routine returns as the job is already scheduled. If the link is zero, it is loaded with the link from the job currently in control of the CPU to the next scheduled job. The link to the last scheduled job (JOBRNQ+2) is loaded with the link to the job currently in control. Thus, the JOB referenced by R0 when a JRUN is executed is inserted between the job which executed the JRUN and the next scheduled job. Because the interrupts were disabled by the call to JRUN, the job will be the next one scheduled. Obviously, a JRUN with R0 referencing its own job will accomplish nothing as the job must have been scheduled to execute the JRUN.

In the JWAIT call the flag argument is picked up, its bits are set in the JOBSTS word, and PC is adjusted past the argument. The JWAIT functions in an opposite manner to JRUN. JWAIT links the job scheduled before the one referenced by R0 to the job scheduled after it and clears the forward pointing link in JOBRNQ+4(R0). If the link was already cleared, JWAIT is exited as the job is already in a wait state. The job's SP is saved in JOBRNQ+14, the internal stack address is loaded, the RUNQUE is indexed, the DYSTAT arrow is cleared, and the next job is scheduled. JWAIT may be called with R0 referencing its own job; the caller should obviously provide a means for the job to be rescheduled.

The SCAN call follows JRUN and executes entries in the SCNQUE until the link to the RUNQUE is reached.

The SLEEP call inserts a queue block into CLKQUE by loading R3 with the address of the CLKQUE and calling QINS. Part of the SLEEP code is a routine which decrements the tick argument of the SLEEP call until it is 0; one decrement per clock tick. The address of this code is placed in the second word of the queue block (the first word in the block is the link to the next CLKQUE entry). The tick argument is placed in the third word of the queue block (referenced by R3 after the call to the routine) and the job's address is placed in the fourth word. The job itself is placed in a wait state with "JWAIT J.SLP". When the tick argument reaches zero, the job's address is picked up from the queue block and a "JRUN J.SLP" is executed, clearing the J.SLP flag from the job's JOBSTS word and rescheduling the job.

For those programmers interested in making use of the CLKQUE or the SCNQUE the following remarks and example should prove helpful:

As described above, CLKQUE and SCNQUE entries are identical in format and in the method by which they are processed. The major difference being that SCNQUE entries are processed not only at line clock interrupts, but also when the processor is idle and when the SCAN call is executed. The distinction can be important if the routine is to be used for controlling or monitoring a real time or external event. For the purposes of the following example, an entry is inserted into the CLKQUE, but the method is the same for the SCNQUE.

```
EXMPLE: LOCK                        ; no interrupts
        MOV     @#CLKQUE,R3         ; load address of chain
        QINS                        ; insert queue clock at R3
        LEA     R1,CODE             ; address of routine to be run
        MOV     R1,(R3)+            ; set address of routine in queue block
        MOV     DATA,(R3)+          ; remaining six words can be used for data
        UNLOCK                      ; routine is now queued
        ...
        ...                         ; balance of user program
        ...
        EXIT

CODE:   ...                         ; code which is executed when CLKQUE entry is
        ...                         ; called.  R3 will index the second word of the
        ...                       ; queue block for use as data area.  R4 must
        ...                       ; not be disturbed!  Calls which index a job
        ...                       ; may not necessarily index the caller's.
        RTN                       ; the routine must return
```

Note: The block of code that is inserted into either SCNQUE or CLKQUE may not reside in bank switched memory, it must be in memory common to all users as a job does not run this code, the monitor runs it. In addition, the queue entry must be deleted from SCNQUE or CLKQUE before the program terminates which inserted it if the code is a part of that program. If the code is another memory module which won't move after the program terminates, then the queue entry may remain. Otherwise, garbage will probably be executed the next time the entry is processed resulting in a system crash.

To allow the inserted code to remove itself from CLKQUE (or SCNQUE) when some condition is met, the following instructions (or appropriate alternatives) are added to the routine:

```
CODE:   ...                       ; user routine
        ...
        TST     VALUE             ; condition met yet?
        BNE     RETURN            ;   no
        LCC     2                 ; yes, set "V" bit to delete entry
RETURN: RTN
```

If it is desired that the routine only be executed at fixed intervals a SLEEP call cannot be used as the SLEEP call only deschedules jobs, not CLKQUE (or SCNQUE) entries. A routine such as the one described by Lefford Lowden (Method One) in issue Number 2.3 of this Newsletter should be used. The CPU loading is not severe as the overhead of job context switching is not incurred.

If the job which inserted the queue entry executed a "JWAIT FLAGS", suspending itself until some event transpired, the queue entry can revive it by executing a "JRUN FLAGS" with R0 indexing the job's JCB before the final return.

Good luck!

The three memory calls (GETMEM, DELMEM, and CHGMEM) are all part of the same module and include extensive error testing. The GETMEM call will return a module cleared to nulls.

The most involved module in this section of the monitor is the FETCH/SRCH module. It is composed of over 150 instructions (compared to the barely 70 of the job scheduler). It first determines what control flags have been set and then limits its search on that basis. Unless the F.ABS flag was set a search of either user memory (F.USR) or system memory and user memory is made on a module by module basis.

If a disk fetch was requested, FETCH clears all flags and error codes from the user DDB, except flag bits 4 (transfer initiated) and 5 (read or write). The DDB is then pre-INITed (BIS #40000,DDB) and an INIT DDB is called. This is done to get the address of the disk driver in DDB+12 without allocating a buffer and also, FILSER will supply the user's default device if it was not specified in the specification. The FETCH call uses this address to determine the record size of the device (always the first word of a disk driver). It sets this record size in DDB+4 and then checks to make sure the caller has enough memory to accomodate a disk record. If not, the call will be aborted. If no PPN was supplied in the DDB, the user's PPN is picked up before a search of the disk MFD is made to find the address of the UFD. If the UFD was found, its records are read until the specified file and its link are located. If the file is found, its size is calculated and if the user has enough free memory, a memory module is built and the file is read in. If the user included the F.FIL control flag, the file's name will be copied from the DDB to the housekeeping words of the memory module and the module "FIL" flag will be set.

The KBD call follows the FETCH/SRCH module. If the job has no terminal attached, a "JWAIT J.TIW" is executed, descheduling the job until a terminal is attached. If the terminal is in image mode, a TIN is called and the input character is placed directly in R1. If the terminal is in normal mode, the job's command file size word (JOBCMZ) is tested for command file processing. If the word is zero (not processing a command file) TIN is called, adding characters to the input buffer, until either a line-feed is reached or the buffer is full.

The KBD routine also handles command file processing. If the JOBCMZ word is non-zero, KBD will set its input from the command file buffer, echoing the data if the Trace flag is set and placing it into the input buffer, until a line-feed or a special command file symbol (delimited by ":") is reached. If the command is ":<" the data is only echoed until a ">" is reached.

The TTY call tests the JOBCMZ word to determine if a command file is being processed. If the JOBCMZ word is zero, no command file is loaded and TTY calls TOUT. If the JOBCMZ word is non-zero, bit 4 of JOBCMS is tested to determine if the Trace flag is set; if not, TTY returns. If Trace is set, bit 2 of JOBCMS (Revive) is tested and, if set, TOUT is called else TTY returns.

The TTYI call executes a TTYL until a null character is reached. TTYL executes a TTY. In the TTYL call, a carriage return (octal 15) gets an automatic line-feed (octal 12) appended.

The TAB call executes a TTYI with octal 11 and 0 as immediate data and a CRLF does the same, but with octal 15 and 0 as immediate data.

The terminal service routines follow the preceeding calls and they will be discussed in the next article.

WARNING ON USE OF SLEEP MACRO

Related to the discussion above, one should not specify in an assembler routine a SLEEP period greater than 32767 clock ticks (77777 octal or 7FFF hex). In a system with a 60 Hz power line, this is equivalent to just under 9.1 min or 546 seconds. In the processing of the SLEEP call (see above), the argument is placed in a queue entry and decremented each clock tick until it reaches zero. With arguments larger than 32767, the argument is technically negative as viewed by the WD16. Thus, decrementing it will generate an overflow which is reflected in the setting of the status bits. On exit from the decrement routine the "V" bit has not been cleared due to the overflow, thus causing the related CLKQUE entry to be cleared. Thus, it will not be run next time. However, that code tests the decremented value for zero which would ultimately issue a JRUN instruction to restart the job! Ergo, the job will hang in limbo forever. Presumably at some later release of the system this minor error will be fixed.

TYPO IN SLEEP AND DING IN JUNE NEWSLETTER, NUMBER 1.C

GTDEC in the listings was mistakenly spelled as GETDEC, sorry.

DISCUSSION OF AMOS (TM) MONITOR, PART 2

by Peter A. Jacobson

The terminal service routines, TRMSER, follow the first section of the monitor which was discussed in the last article. The source for these routines is available from Alpha Micro in the file, "TRMSER.MAC" on the Driver Source Diskette (SFD-00003-00). Documentation on the major operational aspects of TRMSER is discussed in the document, "Terminal Service System" (DWM-00100-33), which is included with every AM-100 (TM) board. The supervisor calls which access TRMSER are well documented in the AMOS MONITOR CALLS MANUAL and the reader is advised to refer to that manual for further information. The following discussion will cover only the major points of TRMSER, those items which are undocumented, and some programming techniques.

TRMSER provides input and output linkage between the jobs allocated to the system and their terminals. This linkage is maintained through the JOBTRM word in each job's JCB. The JOBTRM word points to the terminal line table associated with the actual terminal. The terminal line table is part of a linked list which makes up the terminal definition chain (TRMDFC). In addition, each terminal line table entry is linked back to the JCB of the job to which the terminal is attached. This forward and backward linkage is an important safeguard feature insuring that when a terminal is attached to a job (through ATTACH), any job previously attached to that terminal is detached from it before the attaching is completed.

The JOBTRM link in the JCB does not actually point to the base entry in the terminal line table, but rather to the terminal status word (word 3 starting from zero). The first three words consist of: 1) link to the next terminal line table entry (0 for last entry), 2) first word of the terminal name (packed RAD50) and 3) second word of terminal name (also RAD50).

Before getting into terminal I/O, some definitions and explanations should prove helpful. The following components are necessary for terminal I/O in an AM-100:

TRMSER - The section of the monitor which passes terminal input and output
 to and from the job attached to it.

JOBTRM - The line table of the terminal to which the job is attached. It
contains links to the terminal driver, interface driver, the job
attached to it, and the next terminal definition chain entry. In
addition, it contains the I/O buffer addresses and the various
parameters associated with the buffers [see #1.2, LFL].

DRIVER - Two types of drivers are necessary:
1) A hardware interface driver (IDV), responsible for actually
receiving and sending data to and from the terminal.
2) A software terminal driver (TDV), which performs custom handling
of data if necessary.

The TDV has the following as an example of its first 5 words:

```
WORD    DATA            ; terminal attributes:
                        ; bit 6 if set says terminal has null output
                        ; bit 7 if set says terminal has has local echo
BR      INPUT           ; input routine
BR      OUTPUT          ; output routine
BR      ECHO            ; echo routine
BR      SPECAL          ; special processing routine (cursor, etc)
```

If custom handling is not required for a routine, the branch instruction
is replaced with a return (RTN).

For most terminal drivers, the attribute word is zero. The PSEUDO
terminal driver has bit 6 set and the NULL terminal driver has bits 6 and 7 set.

The character input routine can be used to convert the input of
non-ASCII devices to ASCII or to throw away characters. The charcter is passed
in R1 while R5 indexes the terminal line table.

The character output routine can be used to convert from ASCII to
whatever code the terminal accepts, add nulls to ouput (see SIL700.MAC), or
perform a software form-feed on hard copy terminals that don't support form-feed
(see example below).

The echo routine is used for special processing of rubouts (on hard copy
terminals) and control-U.

The special processing routine is generally used on CRT's to provide conformity with Alpha Micro's defined standard functions for X-Y cursor movement, clear to the end of line, home, clear CRT, etc. However, it could be used with any terminal to provide special features.

For hard copy terminals which may have optional form-feed capabilities, or which are missing them entirely, or which have the form-feed response fixed at one page size only, the following output routine can be inserted into the appropriate driver routine to generate software form-feeds:

```
PSZ     =       46              ; location of pagesize info
LNC     =       50              ; line counter -- both addresses must be changed
                                ;   if Alpha Micro starts to use them

OUTPUT: TST     PSZ(R5)         ; see if a page size defined
        BEQ     DONE            ;   no page size, ergo return
        AND     #177,R1         ; strip character to ASCII
        CMP     R1,#14          ; form-feed character?
        BNE     LF              ;   no
        LEA     R3,22(R5)       ; index terminal output queue           (1
        QINS                    ; get & insert block into output queue  (2
        MOV     LNC(R5),2(R3)   ; set number of lines left on page      (3
        MOV     #12,4(R3)       ; set literal character to a line-feed  (4
        CLR     R1              ; throw away form-feed character
        BR      DONE            ; return

LF:     CMP     R1,#12          ; line-feed character?
        BNE     DONE            ;   no
        DEC     LNC(R5)         ; decrement line counter - at zero yet?
        BNE     DONE            ;   no
        MOV     PSZ(R5),LNC(R5) ; reset the line counter to top of form
DONE:   LCC     10              ; set N bit for position processing     (5
        RTN
```

There have been two assumptions made in the above code: 1) word at location 46 in the terminal line table contains the number of lines per page for software control of form feeds. If this word is zero, it implies that the hardware on the terminal will interpret form-feeds. 2) The word at location 50 counts the remaining lines on the page. Currently these two words are unused in the terminal line table.

Terminal output is processed both by an actual output buffer and by queue block chains. Lines 1 thru 4 in the above code insert a queue block to handle the software form-feed. Line 1 loads the address of the output queue into R3. QINS obtains a queue block from the monitor, links that queue block at the front of the queue addressed by R3, and returns with R3 addressing the first data word in the block obtained. With the 4.1 release of AMOS (TM) each queue block consists of 8 words -- one link word and 7 data words (numbered in this context 1 thru 7). The first data word contains a command code which determines how the queue entry is to be handled. A summary of these commands follows:

DATA	COMMAND
0	data block (buffer or literal)
2	subroutine call
4	output image data
6	cancel output wait state
10	output suspended

The queue block entries are processed before output buffers. Since the QINS call returns the block with the data words zeroed, the command in the example is 0 - data block. The processing of this command takes two forms: 1) buffered data, and 2) literal characters. The first form allows the user to queue up a buffer of data for immediate output and the second form allows the user to specify a particular character to be output. In both cases the second data word in the queue block contains the character count. The third data word is either the literal character or the address of the buffer. If the third word is greater than octal 377, it is assumed to be an address.

Note that when the form-feed is replaced by the queue block entry for the appropriate number of line-feeds, the line count is not restored. One is tempted to think that this is an error. It isn't, as the line-feeds generated by the queue block entry also go through this routine and when all of them have gone through, this routine resets the line count by copying the page size into the line counter and a new page begins.

Line 5 sets the return code from the terminal output routine. The possible codes are:

```
N bit set (LCC 10) - char is output and positioning is adjusted
Z bit set (LCC 4)  - char is bypassed (assumed the routine performed output)
N and Z bits off   - char is output without positioning
```

Output positioning is maintained to handle tabs, rubouts, and control-U's.

In addition to now being able to send form-feeds to the terminal, if the hard copy device has a key board, typing a control-L will echo a form-feed.

Briefly (and somewhat over-simplified) terminal input is handled as follows:

1) For interrupt driven interface drivers (AM-300 (TM) and AM-310 (TM)):
   A key is depressed on the terminal's keyboard, causing the board to generate a vectored interrupt. The location of the interrupt handling routine is determined by fetching PC from absolute location 28 (hex) and adding the device code to it. The contents of this intermediate location is added to PC to form the final address of the input routine.

   For non-interrupt driven interface drivers (IMSAI SIO (TM)):
   When the driver is initialized (at its first apearance in a TRMDEF command in the SYSTEM.INI file) an entry is placed in the SCNQUE. Every time the queue is scanned the device's status will be checked for characters awaiting input. When a character is ready, the input begins. (For these drivers only: the scanning routine will also check for possible output to process).

2) The address of the terminal line table is placed in R5 and TRMICP is called. TRMICP calls the input routine of the terminal driver which will supply any special processing the character needs.

3) If the character makes it past the tests for image mode input; control-S, -Q, or -C, double escape, rubout or control-U, printable, etc.; it will be echoed via a call to TINIT.

4) If the character is a carriage return, a free line-feed will be appended. When a line-feed is reached, if the terminal has a job attached which is in

a terminal input wait state, the job will be rescheduled to process the input line.

Again briefly terminal output is handled as follows:

1) TOUT adds characters to the terminal output buffer, initializing terminal output via the TINIT call, and putting the job (if any is attached) into a terminal output wait state if the buffer is full. TBUF works the same way, but uses the data from a user designated buffer rather than taking single characters from R1 as TOUT does. TCRT calls the terminal's special output handling routine, which in turn initiates output by calls to TTY, TTYI, and TTYL where a call to TOUT is made.

2) The interface driver for the terminal uses a call to TRMOCP to get the next character from the output buffer.

In order for keyboard data to be processed by an assembly language program, the program should execute a KBD call which will deliver one line of data, indexed by R2 and terminated by a line-feed. If the terminal is in image mode, the character is delivered directly in R1. KBD makes repeated calls to TIN, filling the input line buffer until a line-feed is reached. If the job has no terminal attached, execution of the program will stall in KBD by descheduling the job (JWAIT J.TIW) until a terminal is attached and keyboard data is available or is delivered by a FORCE command. TRMICP reschedules the job (JRUN J.TIW) when a line-feed has been entered.

The TIN call stalls via a "JWAIT J.TIW" until a line-feed is entered and all data has been echoed. The data is then transferred, character by character, from the input buffer to R1 where if TIN was called by KBD, the character is placed in the input line buffer. The TIN call is the only routine in the monitor which will perform lower to upper case ASCII conversions. The conversion is made if bit 4 of the status word is not set (the EXIT call always clears this bit to insure the entry of system commands in uper case). The conversion does not affect how the character is echoed, it only places the upper case character in R1.

One disadvantage to using either a KBD or TIN call to get data from a terminal is that the job is descheduled until a carriage return or line-feed is received. If the terminal is in image mode, the job is still stalled until at least one character is received. If the programmer is writing real time programs, this technique will not work to get input data. In the August issue (Number 2.2), Lefford Lowden provided a method of retrieving a single character from a terminal only if one had been entered. The subroutine bypassed the KBD call until data had actually been placed in the input buffer. Below is a routine which will accomplish the same thing bypassing the KBD call altogether:

```
;       terminal line table equates (from TRMSER)

ICC = 10                        ; input character count
LCH = 37                        ; last character input

START:  CALL    CHAR            ; test for user input
        BEQ     REAL            ; branch if no input
        ...
        ...                     ; process input data
        ...
REAL:   ...
        ...                     ; continue real time processing
        ...
        BR      START

CHAR:   MOV     @#JOBCUR,R0     ; load this job's JCB
        MOV     JOBTRM(R0),R0   ; load its terminal line table
        TST     ICC(R0)         ; any input?
        BEQ     DONE            ;   no
        CLR     ICC(R0)         ; reset input character count
        MOVB    LCC(R0),R1      ; load last character input
DONE:   RTN
```

This method does not require setting the terminal into image mode nor does it require a call to KBD. [It does, however, require that the loop in the real time programming section never stall longer than about 1/20th second (the expected typing speed of a good typist). A heavy computing load due to several other jobs in the CPU might make this assumption invalid. For stand alone programs it is probably OK. LFL]

The routine can be modified to process buffered data by using a TTYIN call as follows:

```
CHAR:   MOV     @#JOBCUR,R0         ; load this job's JCB
        TTYIN   R1,@R0              ; get character from input line buffer
        TST     R1                  ; set Z bit reflecting presence of character
        RTN
```

The TTYIN call adjusts the input character count, returns the first character in the input buffer, and adjusts the buffer. If no characters are available, R1 will be cleared.

NOTE TO SERIOUS GAME PLAYERS: You should now have enough information to construct some really good games in a multi-tasking environment (Klingons that shoot first, real time games between two terminals and/or jobs, etc.). The game can proceed whether or not the player is ready and the CLKQUE (discussed in the previous article) can be utilized to allot a specific amount of time to each player. Don't keep these programs to yourself.

Processing of control-C's does not require the use of a KBD call. When TRMICP detects a control-C, it clears the buffer indices, echos a bell code, sets the J.CCC bit in the JOBSTS word and reschedules the job if it was waiting for terminal input (an earlier JWAIT J.TIW). CTRLC calls placed appropriately in a program will test the J.CCC bit of the JOBSTS word, and, if it is set, will adjust PC to the trap routine specified as the argument of the CTRLC call.

Directly following the TRMSER routines are the initial entries in TRMIDC (terminal interface driver chain) and TRMTDC (terminal driver chain). The TRMIDC has one entry in it; the PSEUDO interface driver. Its input and output routines are obviously both returns. The TRMTDC chain contains two initial entries; PSEUDO and NULL. The PSEUDO terminal driver contains three RTN's as neither input, output, nor echo require special handling with a PSEUDO terminal. The NULL terminal driver contains RTN's for both input and echo, but the output routine sets the Z bit, inticating that the terminal driver handled output, with the result that a NULL terminal discards all output.

NEWSLETTER FOR AM-100 USER'S GROUP   Produced by Lefford F. Lowden
                                                  616 Long Pond Road
1 DECEMBER 1979   Number 2.6   Rochester, New York 14612

DISCUSSION OF AMOS (TM) MONITOR, PART 3

by Peter A. Jacobson

The next section of the monitor includes the file service routines, FILSER, and the file error message routines, FILERR. As with TRMSER, source for these routines is available from Alpha Micro on the Driver Source Diskette (part number SFD-00003-00). The macro calls to these routines are well documented in the AMOS MONITOR CALLS MANUAL and, again, the reader is advised to refer to that manual for information on the individual calls. The following discussion will focus on some of the operational aspects of calls to FILSER. It is assumed that the reader has knowledge of the structure of a DDB.

Entry into FILSER is made only through an SVCB 0. As discussed in the first article (Number 2.4), the SVCB decoding module will place the function code in R0, the address of the first argument in R4 (for SVCB 0 this will always be the address of a DDB), and the address of the second argument (if any) in R3. Upon entry into FILSER, the function code and the address of the error recovery routine are pushed onto the stack as a return address for possible error trapping. If a device was not explicitly entered in the DDB, the user's default device code is picked up from the JCB. The address of the general device driver routine, DSKSER (discussed in the next article), is placed in R2 and the address of the system disk driver (first word of DSKSER) is placed in R1 as a default, in that most file calls will be to DSK.

The device driver will be fetched either from DSK0:[1,6] or memory if the device was not DSK. An error is reported if the driver can not be located. The driver attribute word (described below) is tested to determine if the device is file structured; if not, the address of the DSKSER routines is replaced by the driver's address. If the flag bits in the DDB indicate that INIT was called, execution procedes to the indicated subroutine. All calls except INIT require that a buffer be allocated first; thus, if the DDB has not been INITed, an error occurs and processing depends on the settings of the flag bits in the DDB (default is that an error message is produced and the routine EXITs). If the DDB is INITed, a TJMP to the appropriate routine within FILSER is executed where a call may be made to either DSKSER (for file structured devices) or to the actual device driver. NOTE: If DSKSER is being used, it in turn will call the actual disk driver routines.

Both DSKSER and unique device drivers have a communication area at their beginning, where words three through eleven are the addresses of routines. The basic layout of this area is:

WORD	CONTENT
1	physical record size
2	driver attributes (function supported if bit set)
	bit 0: read
	bit 1: write
	bit 2: assign
	bit 3: open
	bit 4: close
	bit 5: input/output
	bit 6: rename/delete
	bit 7: not used
	bit 8: special (bitmap maintained)
	bit 9-14: not used
	bit 15: file structured
3	read/write
4	open
5	close
6	assign
7	input
8	output
9	delete
10	rename
11	special (bitmap routines)

Word 1 of DSKSER is the address of the system disk (DSK) driver routine. To clarify: if the device is file structured, the FILSER routines will call DSKSER which, in turn, will make calls to the specific disk driver; if the device is not file structured, the FILSER routines will call the specific device driver. R1 will always contain the address of the actual driver, and R2 will contain either the address of DSKSER, or, if not file structured, the address of the driver.

The INIT call performs two functions, 1) places the address of the device driver routine in DDB+12 and 2) allocates a DDB buffer based on the size of the physical record for the device. The size of the physical record is put in DDB+4. The GETMEM call, with the MCB pointing at DDB+2, puts the address of the buffer in DDB+2. The driver address will always be placed in the DDB. However, if a buffer has already been allocated, a new buffer will not be allocated.

The OPEN call ASSIGNs the device (if possible) and sets the open code in DDB+34. The open code is automatically supplied as the second argument of the BVCB 0 by the macro coded in SYS.MAC[7,7]. There are four types of open calls available, and they will set the following data in DDB+34:

Macro Name	Data
LOOKUP	0
OPENI	1
OPENO	2
OPENR	4

Note that the fourth code differs from that defined in the AMOS MONITOR CALLS MANUAL which appears to be in error.

The OPEN call then resets the logical record size in DDB+4, clears the buffer index (DDB+6), and reclears the DDB buffer (addressed by DDB+2). The device driver attribute word is tested to determine if an OPEN routine exists, and if it does, the routine is executed. Note that although the OPEN routine in FILSER clears the DDB buffer to nulls, the device driver's open routine may use the buffer and it may not remain null.

The CLOSE routine will return the error message "FILE NOT OPEN" if the file is not open. If the open code indicates the file is open for sequential input or random processing, the device's CLOSE routine is bypassed. If the DDB was open for sequential output, the final record is written and the device's CLOSE routine is executed. When control is returned to the FILSER CLOSE routine, the open code is cleared and the device is deassigned (if necessary).

The DELETE, RENAME, and SPECL calls are contained in the same routine of FILSER and call the driver's routine (or the DSKSER routine) if it exists. If the device is file structured, DSKSER is called and the SPECL calls perform modifications to the device's bitmap.

The ASSIGN and DEASGN calls will lock non-sharable devices to the calling job. If the device is sharable, the calls are ignored. If the device is already ASSIGNed to another job, the "DEVICE IN USE" error message is returned.

The READ and WRITE calls clear or set bit 5 of the DDB flag byte, test the DDB record size word (to determine if a read or write is possible) and call the driver's routine or the DSKSER routine. Through this call, physical transfers to and from the device are initiated. The source code in FILSER for the READ and WRITE calls indicates interrupt driven devices will have READ/WRITE requests queued, but in the 4.2 release of the monitor, this is not yet implemented. The two lines of code which test for interupt driven service and branch if true should be commented out, as they do not exist in the actual monitor.

Alpha Micro has not yet documented the IODQ call in that queued I/O is not yet supported. Its function will be to dequeue a transfer request from the DDB chain after the transfer is completed.

The INPUT and OUTPUT calls both insure that the file is open for random or sequential input or output. If the device driver does not have input and output routines, the read and write routines of the driver are called.

After a call to FILSER has been completed, the file I/O return (FIORT) processor in FILSER is executed. Error codes are returned in DDB+1, but if bit 7 is set, error processing is bypassed (bit 7 is set by LOOKUP calls which do not actually open files). If an error condition exists, a call is made to FILERR to print the error message and execution of the porgram is aborted with an EXIT call. Both the printing of error messages and the abort on error can be suppressed by setting the appropriate bits in the first word of the DDB as described in the AMOS MONITOR CALLS MANUAL (page 6-4).

The next section of the monitor contains the executive program and the remainder of the supervisor calls which will be discussed in the next article.

NEWSLETTER FOR AM-100 USER'S GROUP   Produced by   Lefford F. Lowden
616 Long Pond Road
1 JANUARY 1980   Number 2.7   Rochester, New York 14612

DISCUSSION OF AMOS (TM) MONITOR, PART 4

by Peter A. Jacobson

The EXEC module (my designation) follows the FILSER routines which were discussed in the last article. This module contains the executive program and the remainder of the supervisor calls.

The first routine in EXEC is EXIT. When a job is allocated to the system, the JOBS program sets EXIT as the first program the job will execute. EXIT first enables interrupts (in case the user program left the processor LOCKed) and then determines if the EXIT call was forced by a CTRLC call. If a Control C is waiting (J.CCC set in JOBSTS), "^C" is sent to the job's terminal and the JOBCMZ word is cleared, which aborts any remaining commands in a command file. All flags except J.ALC are cleared from the JOBSTS word and the J.MON flag is set.

The DDBCHN is than scanned to determine if the job has any I/O queued. If it does, the remainder of the job's CPU time is used to process it. (Queued interrupt driven I/O is not supported in the 4.2 release of the monitor).

The error control intercept (JOBERC) and the breakpoint vector address (JOBBPT) are both cleared and the job's stack pointer is restored to the top of the job's stack. If the job has memory assigned (the J.NUL bit is not set in the JOBTYP word) the DEVTBL is scanned and any devices assigned to the job are deassigned, and the following memory tests are performed.

If the job has a new memory allocation, the first word of the job's memory will be cleared, insuring the memory partition contains no garbage. The partition and any memory modules in it are tested for address errors. The system displays "[MEMORY MAP DESTROYED]" if the partition's base address or a module's address is odd, or the partition's base address or a module address is above MEMEND. If none of the module flag codes: FIL, FGD, or LOK are set, the module is removed from the partition. This operation gets rid of buffers and other temporary memory modules that were created with the INIT or GETMEM calls, the program file unless it was placed in memory earlier with a LOAD command (which sets the FIL flag automatically), and modules that were set for removal with the DEL command (ERASE on Lefford's system).

The memory calls, GETMEM and CHGMEM will always return an even module size and clear the word following the module, but memory module errors can be caused by a program which itself modifies the module size word, either by error or intentionally. The location of ascending modules is determined by adding the module size word (the first housekeeping word) to the address of the current module, generating the address of the next expected module. If the content of this location is zero, the end of memory modules is assumed to have been reached. If the address is odd, caused by an odd byte count, or the address is beyond the end of user memory (MEMEND), the error is reported and the content of the offending address is cleared.

If a program leaves some garbage modules in the partition (VUE used to) which cannot be deleted with DEL, a MEMORY 0 command followed by MEMORY xxxxx will restore the partition. xxxx takes on the value of the previous size of the partition. If you don't know this, type MEMORY with no argument, and the system will tell you what it is.

Normally, this type of memory error should not be fatal to the job or the system, but there are two bugs in EXIT which can cause the job or the system to crash. The first is that no test is made to determine if a module, either real or erroneous, extends beyond the end of the job's partition. The second is in the method by which EXIT clears the erroneous address. The current module's address is saved in R3 before the next module's address is calculated. If the module creates an error, the content of the word addressed by R3 is cleared, however, R3 is not set until after the first module is tested. Therefore, if the first module creates an error, the content of an undetermined word addressed by R3 is cleared.

If the job has terminal output in progress (the OIP flag is set in the terminal status word for the terminal ASSIGNed to the job), it will loop in EXIT until all terminal output has been processed after which the terminal status flags will be reset (restoring normal terminal I/O) and control will pass to the executive program.

The executive program (hereafter called EXEC) is at AMOS (TM) command level, which upon entry issues the monitor prompt, ".", and calls KBD where the job is descheduled until terminal input is available, or where, as described in the first of these articles, if a command file is being processed, the input line buffer is filled from the command file data area. After a command is entered, EXEC resets the job stack pointer to the top of the stack and determines if the job has any memory allocated. If the job does not have any memory, EXEC will allocate to the job all of the available memory in the job's bank. If at least 2000 decimal bytes are not available, the "[NO MEMORY AVAILABLE]" message is issued and EXIT is called.

An FSPEC call with a null default extension is performed on the command line with the job run block (JOBRBK - partial DDB in the JCB) indexed to receive the file specification. The file name is copied into the JOBNAM words, the J.MON bit cleared, the J.LOD bit set in the JOBSTS word, and a search for the file is made. The AMOS USER'S GUIDE is somewhat ambiguous on what format is used for system commands (page 7-3), but full file specifications are allowed. The file search order is documented, although not accurately, in Appendix B of that manual, and is briefly summarized below. If the Device, Drive, Extension, or PPN are supplied on the command line, the appropriate search is bypassed.

DEVICE	EXTENSION	PPN
system memory	PRG	N/A
user memory	PRG	N/A
DSK0:	PRG	1,4
DSK0:	CMD	2,2
user	PRG	user
user	CMD	user
user	PRG	user library

If the search failed up to this point, DSK0:MDO.PRG[1,4] is loaded which attempts to locate the file in the following order:

DEVICE	EXTENSION	PPN
user	DO	user
user	DO	user library
DSK0:	DO	2,2

If the file is found, it is loaded into the job's memory, providing enough memory is available. If the file is not found, the command is echoed to the job's terminal, bracketed with question marks. The message "?Insufficient memory for program load" is returned if the job did not have enough memory.

After the file is loaded, EXEC tests its first word to determine if the program can be run not logged in. If the first word of the file is non-zero, the JOBUSR word is tested, and if it is zero (the job is not logged in), the "[LOGIN PLEASE]" prompt is issued and the executive program EXITs.

The file's extension is tested, and if it is "PRG", program execution begins with MOV R3,PC. Upon entry into the program the registers contain:

R0 - base address of JCB
R1 - cleared
R2 - remainder of input line buffer
R3 - base address of program
R4 - cleared
R5 - cleared

A file with an extension other than "PRG" is assumed to be a command file. EXEC moves the file in reverse order to the top of the job's partition, sets the C.SIL bit in the JOBCMS word, and branches to the entry point of EXEC. If the file was loaded under control of MDO, the specified parameters are inserted into the command file and a block move places the command file at the top of the job's partition after which MDO EXITs.

The numeric conversion calls, DCVT and OCVT (or HCVT - hex convert), follow EXEC. They perform binary to ASCII conversions based on table entries, which limit the magnitude of the output.

Following the conversion routines is FSPEC which is well documented in the "AMOS MONITOR CALLS MANUAL" (pp 6-8 through 6-9). Lower to upper case ASCII conversions are not made in FSPEC, therefore whatever R2 points to (terminal input line buffer or program buffer) must be in upper case.

PFILE follows FSPEC and performs in essentially the opposite manner to FSPEC. Its output is directed only to the job's terminal (via a TTYL) and PPN output is always in octal regardless of the setting of the J.HEX flag in the JOBSTS word.

The JOBIDX, JOBGET, and JOBSET require more words to describe than the amount of code they occupy in the monitor. They are all SVCB calls and require a considerable amount of time to decode in the SVCB processing module. If speed of execution for a program is a consideration, the following macro definition will mimic these calls saving over sixty instructions to be executed:

```
J.IDX = -1                                        ; JOBIDX control flag
J.GET =  0                                        ; JOBGET control flag
J.SET =  1                                        ; JOBSET control flag

DEFINE  JCB     TAG,ITEM,CTRL
        PUSH    #ITEM                             ; load JCB index
        ADD     @#JOBCUR,@SP                      ; build job table entry
        IF      LT,CTRL,  POP TAG                 ; JOBIDX
        IF      EQ,CTRL,  MOV @(SP)+,TAG          ; JOBGET
        IF      GT,CTRL,  MOV TAG,@(SP)+          ; JOBSET
ENDM

Instead of      JOBIDX  R0,JOBTRM       use     JCB     R0,JOBTRM,J.IDX
and             JOBGET  R0,JOBTRM       use     JCB     R0,JOBTRM,J.GET
and             JOBSET  R0,JOBTRM       use     JCB     R0,JOBTRM,J.SET
```

The object code resulting from this macro will use between 62 and 77 machine cycles, while just the SVCB instruction and the ensuing RSVC instructions alone consume 135 cycles (which doesn't include any of the decoding timing).

The three calls USRBAS, USREND, and USRFRE follow next in the monitor. They function as described in the "AMOS MONITOR CALLS MANUAL".

The CTRLC call was briefly described earlier article on TRMSER. It tests the J.CCC bit in the JOBSTS word and, if set, it means that a Control C was entered at the job's terminal (or by the KILL program) and, consequently, the argument address of the CTRLC is added to the saved PC.

The PRNAM call sends its output to the job's terminal via a TTY call as does PRPPN. Like PFILE, PRPPN displays the PPN in octal regardless of the setting of J.HEX flag in the JOBSTS word.

The terminal input line processing calls BYP, ALF, NUM, TRM and LIN occupy the next section of the monitor and are unremarkable except to note that ALF tests for upper case alphabetic characters only.

The FILNAM call functions exactly as described in the manual.

The GTOCT call follows FILNAM and is really two calls; GTOCT and GTHEX. The GTHEX code is used if the J.HEX bit is set in the JOBSTS word. If the leading character is a digit, it does not ned to be preceded by zero. If the input is greater than 1777777 octal (FFFF hex), causing an eror to be reported (N flag set), the result contained in R1 will be meaningless.

GTDEC, unlike GTOCT will stop processing the input line if the next character will cause the result to be greater than 65535.

The GTPPN call will always process the input PPN on an octal basis regardless of the setting of the J.HEX bit.

The final two calls in this section of the monitor are the PACK and UNPACK calls for which no description will be attempted except to note that UNPACK uses a small (two word) table for unpacking.

The concluding article will discuss DSKSER and INITIA.

USER COMMENTS ON AMOS 4.2 (TM), SUGGESTIONS FOR IMPROVEMENT

by Logical Software, Inc.

MONITOR:

Absolute, immediate check for <^C> (control-C) character input from the terminal. Would always cause immediate exit from the program in control except where specifically defeated within the program. Many A-M programs can't be halted by <^C> entry or KILL, etc. EG, COPY TRM:slowterminal=bigfile.

Buss errors: should write-protect the disk, report all buss errors to the CONSOLE (terminal connected to JOB1) then to all other terminals, then halt or go to an error recovery procedure. Possibly should re-boot automatically. Most buss errors are serious enough to require the attention of the system programmer, the users should not be left to wonder where everything went. Usually, nothing will work right anyway, but a user who is unaware that the system is down can waste a lot of his time before he finds that his new file is now gone forever.

Should incorporate FLTCNV, SCNWLD, and TODCNV.

Ability to save away an existing running program, load and execute another, then continue the first program without error.

New special character, Control-T, would mean "Are you still alive, AMOS?" Would respond with a beep for yes, silence for reboot time. Second <^T> within about 1 second would cause systat to be displayed. Third <^T> would save away the existing program, make processor and memory available to user. User can continue first program via CONT.PRG.

Auto program loading: BASIC and others allowed to load subroutines (.SBR and .RUN) as required by the program in control (from library areas), keep in memory or delete by internal command. Would greatly simplify program writing, and avoid clogging up the BASIC program and memory with seldom-used subroutines. Would effectively give BASIC programs much more memory to work with, or allow running in smaller chunks of memory.

DESIRED FEATURES IN SYSTEM PROGRAMS

BASIC: A) Structured programming including WHILE, REPEAT UNTIL, EXIT ON, and DO-ENDO. B) PRINT USING 'LLL 'RRRR 'CCCC. C) Multi-line function definitions (with local variables). D) A full renumber. E) Provision for initializing arrays at compile time, within MAP statements. F) CASE statements. G) MAT statements. H) Edit capabilities - line and global. I) INCLUDE (would work the same as C-BASIC in CP/M -- reads in a named source file). J) Allow RUN to look in library or public file area for subroutines written in BASIC. K) Allow program in control to determine whether to delete .SBR files and subroutines written in BASIC (from memory). L) Allow system functions (date, time, IO, etc.) from the terminal attached to Job1 only. M) Include FLOCK as

NEWSLETTER FOR AM-100 USER'S GROUP Produced by Lefford F. Lowden
616 Long Pond Road
1 FEBRUARY 1980 Number 2.8 Rochester, New York 14612

DISCUSSION OF AMOS (TM) MONITOR, CONCLUSION

by Peter A. Jacobson

The next section of the monitor is DSKSER which is mentioned by Alpha Micro only briefly in FILSER. DSKSER is the generalized device driver for file structured devices such as floppy disks and hard disks. It simplifies the actual code needed for a disk driver and allows the system to access several devices without a great amount of code. Like other device drivers, it has a communication area at its beginning which was defined in the December issue. The attribute word of DSKSER is zero, but all functions are supported, except ASSIGN. File structured devices cannot be assigned to one job, but for obvious reasons, two jobs cannot be given access to the same file structured device simultaneously. The controlling job cannot lock other jobs out by disabling interrupts because other I/O devices might lose data. The solution is to increase the controlling job's priority considerably until it is done with the disk. In DSKSER, this is accomplished by setting the job's priority counter to 177777 octal (about 18 minutes).

Before discussing DSKSER's routines, the structure of a disk driver communication area will be described as well as the three support routines used by DSKSER.

Disk drivers, like non-file structured device drivers, have a communicaions area at their beginning, which contains some differences from other drivers. the 200DVR.DVR disk driver communication area is described below as an example:

Word 1	physical record size
Word 2	driver attributes
Word 3	driver entry address offset
Word 4	physical sector size
Word 5	physical sectors per logical record
Word 6	maximum record number

Words 7-16	not used
Word 17	maximum record number
Word 18	directory entries per record
Word 19	bitmap size

The remaining five words are specific to the physical drive type.

DSKSER includes three support reoutines for locating devices and files. The first of these is a routine (DEVTST) to determine of the device specified in the DDB exists and is mounted. If the device is not specified in the DDB, DEVTST will pick up the job's default device and drive. If the device is specified, but the drive is not (DDB+23 contains 377), device 0 is used. The appropriate error code is set in DDB+1 if the device is not found in the DEVTBL or the device is not mounted.

The second routine (FILTST) is used to locate the UFD entry of the file specified in the DDB. The calling sequence to FILTST is somewhat different than a normal subroutine call in that control flags are set to limit the actions FILTST can perform. The sequence is:

4.4A/21656

```
        CALL    R5,FILTST               ; R5 is used as the linkage register
        WORD    FLAGS                   ; control flags
```

Control flags (when on):

Bit 0	locate file
Bit 1	return error if file already exists
Bit 2	test for programmer number match
Bit 3	lock directory (DSKDRL)

Various combinations of flags can be used. When FILTST returns, R5 is incremented past the control flags.

The third routine (MFDTST) reads the MFD (record one) of the device to locate the specified UFD. If the PPN is not specified in the DDB, JOBUSR is used to locate the UFD link.

The following is a discussion of the DSKSER routines. Keep in mind that these routines are called by FILSER and are not directly entered with an SVCB 0.

FILSER can call any device driver on the system, of which DSKSER is only one.

The first routine in DSKSER is the physical READ/WRITE routine. This is the only section of DSKSER which actually makes calls to the device driver. A call to DEVTST insures that the device exists and is mounted. The specified record number (DDB+10) is tested to determine if it is within the range of the device. If the READ/WRITE call is valid, the device is locked to this job by resetting the job's priority counter as described above. The DDB record number is multiplied by the number of sectors per logical block for the device to get the physical sector number for the device and the transfer is initiated. Upon return from the device driver, the job's priority counter is reset to 1 and, if no other jobs are currently scheduled, the routine returns. If other jobs are scheduled, the job is put to sleep for one clock tick (presumably to give other jobs CPU time).

The OPEN routine handles all four OPEN calls: LOOKUP, OPENI, OPENO, and OPENR. The OPENO call insures that the user has access to the PPN and that the file does not already exist with a call to FILTST and then allocates a disk record for file data. The physical record number is returned in DDB+10 and DDB+42. The buffer index (DDB+6) is set to two. The OPENI and OPENR calls locate the file and if it exists, insure that it has the same type as specified in the open code of the DDB (DDB+35). If the file type matches, the number of records in the file is placed in DDB+36, the byte count of the last record is put in DDB+42 and DDB+10. The LOOKUP call does not distinguish between random and sequential files.

The CLOSE call locks the disk directory and then locates the UFD of the PPN specified in DDB+32 with an MFDTST call. If a directory for the PPN is not allocated, CLOSE will allocate a record. If a directory for the PPN already exists, CLOSE locates the first empty or deleted entry, ir if the record is full, allocates another directory record. Once an empty directory entry is found, the file name and the file directory parameters are inserted inte the directory, the bitmap is updated, and the directory is unlocked. If it was necessary to allocate a directory record, the unused words in the record are cleared to nulls to insure thate are no spurious entries.

The INPUT routine follows CLOSE. For files open for sequential input, INPUT executes a READ based on the record number in DDB+10. The link to the next record is updated in DDB+10, and the buffer index, DDB+6, is set to 2, bypassing the link word. For files open for random processing, INPUT uses the data in DDB+10 as an offset from the base of the file contained in DDB+42. In this case, DDB+10 is not updated.

The DSKSER OUTPUT routine for files open for sequential output can be called on two levels (DDB+22 contains the call level). A level one call is made with a normal user program OUTPUT call which first allocates another disk record for the next call, inserts that record number link into the first word of the DDB buffer, writes the current record, and sets the next record number in DDB+10. A leverl two call is made through a FILSER CLOSE call, which in turn calls the FILSER OUTPUT routine to write the last record. This call does not allocate another record, instead it determines the amount data in the DDB buffer and fills out to the record size with nulls before writing the record.

The OUTPUT routine for files open for random processing insures that the user has access to the PPN with a FILTST call and then writes to the record using the file base offset in DDB+10 added to the file record base in DDB+42 as the current record number.

The DELETE routine locates the directory, insuring the user's PPN grants access to the file, sets the first word of the file name in the directory entry to -1, and then deallocates each record the file had used from the bitmap.

The RENAME routine also locates the directory, insures the user's PPN allows access to the directory, determines that a file of the same name does not already exist and then enters the new name, contained in the three words following the DDB, into the directory in place of the previous name.

The three support routines (DEVTST, FILTST, and MFDTST) are physically located just after the above mentioned routines.

The FILSER SPECL routines are SVCB 0 calls with a function code of octal 14. They essentially are bitmap service calls named DSKDRL, DSKDRU, DSKALC, DSKDEA, DSKBMR, DSKBMW, and DSKCTG. They reside in the monitor directly after the above described DSKSER routines. These calls are fully described in the AMOS MONITOR CALLS MANUAL in the section entitled "DISK SERVICE MONITOR CALLS" (pp 6-17 through 6-21) and the reader is advised to refer to that document for detailed information.

Directly following these seven calls is a special bitmap service routine. The function of this routine is to locate the device's bitmap (if any), rewrite it if necessary, and recompute the hash total to insure the current bitmap is correct. If the bitmap is currently locked by some other job, this routine will stall until the bitmap is free. If the bitmap hash total is not correct, the drive is apparently disabled by setting the drive number contained in the bitmap's partial DDB to 177.

Following this routine is the supervisor call HTIM which is called by 200DVR.DVR if PERSCI drives are being used. HTIM copies the HLDTIM argument into HLDTIM+2 and inserts a clock tick counter routine into the CLKQUE which will decrement the HLDTIM+2 argument until it reaches zero. Upon reaching zero, the routine sends the command to unload the heads of the disk drive and then deschedules itself. The routine will not be scheduled again if it is already contained in CLKQUE.

[illegible]

The next area of the system monitor is a 2000 (decimal) byte area which is reserved for the system disk driver. MONGEN.PRG inserts the driver routine here and updates the MEMBAS work in the system communication area to reflect the end of the driver. Describing the functioning of a disk driver is beyond the scope of this article. However, in passing, it should be noted that disk drivers maintain data storage areas within their code.

The last section of the monitor is INITIA, the initialization program. INITIA is not actually part of the monitor, it merely defines the initial system parameters and starts the first job running the SYSTEM.INI command file; it is eventually overwritten as it resides above the base of system memory as contained in MEMBAS.

As described in the first article, after one of the various monitor loaders (the controller board's PROM routine, MONTST, WNGLOD, HWKLOD, etc.) has finished loading the monitor root, interrupts are disabled and a CLR PC is executed which effectively causes a JMP to absolute 0. This location initially contains a JMP instruction to INITIA. INITIA branches around a temporary stack, sets SP to the address of this temporary stack, and starts a memory test to find how much memory the system has in BANK 0, starting at the end of INITIA and testing in 1K increments. When the end of memory is found, it is set into MEMEND. JOBTBL and JOBCUR both get the address of MEMBAS, JOBESZ is defined (currently 292 decimal bytes), and the first JCB entry is cleared. MEMBAS is updated to the end of this first JOBTBL entry.

An 8K partition is then set up at the end of memory which contains a temporary terminal line table for a pseudo terminal with a pseudo interface driver. This terminal is attached to the first job and allows it to communicate until the first terminal is defined. A temporary device table (DEVTBL) entry is also allocated in the partition for DSK0: and has no bitmap. The various JCB entries are then inserted into the first JOBTBL entry including logging the job into DSK0:[1,4].

"SYSTEM.INI<CR><LF>" is entered in the command file buffer at the top of the partition and the address of EXIT is set on the job's stack as the saved PC of the first program the job will run. The job is then scheduled for CPU time with a JRUN, after which INITIA enables interrupts and loops, waiting for the first clock interrupt to start the job with system initialization.

ERROR WITH THE FUNCTION FIX IN ALPHABASIC (TM)

It has been reported to me that the function FIX in AlphaBASIC (TM) has an error in it. It seems that, for example, if one issues the command PRINT FIX(-.5), the job that one is running crashes. It is not a system crash in that other jobs in the system remain running. However, whatever FIX is doing, the algorithm will not terminate, that job goes into a compute bound state. I have checked release 4.1 and found that the same problem is there also so I think that the problem has been present for some time and merely has not been detected to date (I was unaware of the function in the first place). FIX operates correctly over the range of real numbers with the exception of those arguments between 0 and -1. A rather simple substitution of operators can be made so that the problem can be avoided until the function is corrected. The substitution is the following:

```
For  FIX(X)  substitute   SGN(X)*INT(ABS(X))
```

This substitutes three function calls for one and will run a little slower, but I suspect that FIX is not used all that often and will, thus, make little difference. Thanks to Jack Hobbs.