Unlocking Apple /// - Part 3

Alan Anderson

Hacker's Haven

Welcome back! This issue's discussion of the ever-less mysterious Apple /// will deal with Everything You Ever Wanted to Know about Writing Assembly Language (But Couldn't Find Anybody to Ask). This time, I'll discuss the two different ways of creating Assembly language programs for the ///, and in addition, we'll get into head-spinning detail on how to use a few important SOS calls, integral knowledge for most any Assembly language application. Also, we'll create a real, working, good-for-fun interpreter!

To Boot

As you know if you've read your Apple /// manuals (and who hasn't?), in order for a SOS diskette to boot, it has to have three files: **SOS.KERNEL, SOS.INTERP**, and **SOS.DRIVER**. **SOS.DRIVER** files are created by the System Configuration Program, which is documented in the *Standard Device Drivers Manual*. **SOS.KERNEL** is the operating system itself, supplied by Apple Computer, and not modified unless you want to disassemble it yourself (have fun and send me the source code, please). The **SOS.INTERP** file contains a machine language "control" program; that is, when the diskette is booted, the program in **SOS.INTERP** is executed after all the operating system stuff is installed. Some examples of **SOS.INTERP** files are Business BASIC, Pascal, Apple Writer ///, and VisiCalc ///. Note, however, that no matter what the true nature of this file (*i.e.* BASIC, VisiCalc, etc.), it must be called **SOS.INTERP**. Let's summarize the SOS booting process:

1. Powering on the Apple /// or pressing **Ctrl-{RESET}** causes a jump to the computer's only ROM: a small self-diagnostic program and diskette boot routine.
2. The diskette boot routine from the ROM then reads in a small chunk of code from the diskette in the built-in disk drive. Along the way, the ROM manages to be mapped out of memory, giving the Apple nothing but wide open RAM.
3. The code read from the diskette then reads in the directory of that diskette. It looks for our friend **SOS.KERNEL** and reads it in if found or issues an error message if not found.
4. **SOS.KERNEL** then proceeds to read in and relocate **SOS.INTERP** and **SOS.DRIVER**. After doing so, it finishes up the boot process by executing **SOS.INTERP**.

So, from this little scenario, it is obvious that one way to implement Assembly language programs on the Apple /// is by making them **SOS.INTERP** files. How practical is this? Well, in the case of large, independent applications like languages, VisiCalc, and Apple Writer, this is the ideal method. You share control of the machine only with the operating system and you don't have to worry about any non-essential code hanging around. On the other hand, if you want your Assembly language programs to coexist with BASIC or Pascal, remember that there can only be one **SOS.INTERP** per diskette. That means that making your program an INTERP file is not the way to go if you want a high-level language around.

The Module Squad

Is all hope lost? Of course not! As regular readers of this series know, there is a second method for implementing Assembly language software on the Apple ///. This method, in which the programs are called modules, is used to link the Assembly language code with Pascal or BASIC programs. The Assembly language thingie called Restart which we've been playing around with in the last two installments is an example of a module. A module is simply an Assembly language program which is loaded, relocated, and generally baby-sat by Pascal or BASIC.

There are a few rules to writing modules, and just about all of them are covered in the *Apple /// Pascal Program Preparation Tools Manual* (I keep telling you to read those!). The rules are pretty much the same as those which govern the use of Assembly language routines in Apple II Pascal. In fact, many parts of SOS appear to be descended from the Apple II Pascal Operating System, so a knowledge of that system doesn't hurt when you're working with the ///.

On the other hand, making a **SOS.INTERP** has thus far been documented only in the information received in Apple's OEM class for the ///. Basically, the syntactical rules for writing interpreters are quite simple, and I'll give them to you right here.

An interpreter (which becomes a **SOS.INTERP** file) is an Assembly language codefile with a few identifying items attached to the front. Specifically, these items are:

1. The eight ASCII characters "**SOS NTRP**", which is SOS INTERP with the

vowels removed. (Note the blank between the second 'S' and the 'N'.)

2. Two bytes giving the length of an optional header information block. This block can be used for a copyright notice. The optional header block (if used) follows these two bytes.

3. Two bytes giving the loading address of the interpreter. An interpreter is not relocatable. SOS will automatically load the interpreter at the address given here. Since the interpreter is not relocatable, the source text must contain the **.ABSOLUTE** command.

4. Two bytes giving the length of the code part (everything but this header stuff). The interpreter should be constructed so that it does not use any memory beyond $B7FF.

Could You Interpret That For Me?

If you've read the previous installments of this column, you've already experienced the wondrous thrill of creating and using a module in Pascal and BASIC. Well, in this very magazine, we're going to make an interpreter. But not yet! (Awwwww.) First, we're going to delve into a few essential housekeeping calls to the operating system: SOS calls. For all my noise about SOS calls in this series, I've only documented two, and boy, have I heard it from you folks! So, let's move on into some real SOSsy stuff.

Omniam SOSam in quartes partes divisus est

There are four distinct groups of SOS calls. They are the File System calls, the Device System calls, the Memory System calls, and the Utility System calls. The file calls are probably the most commonly used. They're the ones that let you create, open, close, read from, write to, delete, rename, and otherwise manage files on devices in the system.

The device calls are related to the file calls since files are physically implemented on devices. Device calls let you modify the way the device does something, inquire about the status of devices, and do some other things.

The memory system calls allow SOS to reserve sections of memory for a program's use, and they also allow the programmer to get information about the current use of memory in the Apple.

The utility calls manage some miscellaneous resources in the Apple ///, such as the joysticks and the system date and time.

I'd like to introduce a standard format for SOS call information. To recap briefly, a SOS call is performed with an Assembly language BRK, followed by a byte indicating the call number, followed by a self-relative pointer to a parameter list. It looks like Listing 1.

This chunk of code, called the *Call Block*, is placed in your program just like any other instructions. When SOS sees the BRK it finds the parameter list and attempts to execute the call. An error code is returned in the accumulator. If no error has occurred, the accumulator contains a zero. For a list of possible errors which the calls in this article can produce, see Table 1.

The information which is essential to making SOS calls is the call number and a description of its *parameters*. Parameters come in four flavors: value, result, value/result, and pointer:

Value: Data passed to SOS from the calling program's parameter list. This data is not modified by SOS. Values are 1, 2, or 4 bytes, as specified.

Result: Data passed to the calling program's parameter list from SOS. SOS puts this data in a specified location in the parameter list. Results are 1, 2, or 4 bytes, as specified.

Value/result: Data passed to SOS from the calling program's parameter list. SOS receives this data and passes back a modified value in the same location. This is basically a value parameter and a result parameter which share the same location in the parameter list.

Pointer: a 2-byte address pointing to an area into which SOS places data (for example, in a read from a file), or from which SOS takes data (for example, when writing to a file).

The first parameter in a SOS call's parameter list is always (always, always) a value which gives the number of parameters in the list. For example, if a SOS call has 3 parameters (as does our first example below), the parameter list will begin with a byte containing a 3. In practice, it looks like Listing 2.

Table 1

Possible errors
(returned in the accumulator):

01	Bad system call number
02	Bad caller zero page
03	Bad pointer extend byte
04	Bad system call parameter count
05	System call pointer out of bounds
27	I/O error
2A	Checksum error
2B	Volume is write protected
40	Invalid pathname syntax
41	Too many open character files
42	Too many open block files, or too many block devices
43	Invalid reference number
44	Path not found
45	Volume not found
46	File not found
47	Duplicate file name
48	Not enough room on volume (disk full)
49	Directory full
4A	Incompatible file format
4B	File storage type is neither 1 nor D
4C	End of file has occurred
4D	Position out of range
4E	Access not allowed
4F	Buffer too small
50	File already open, access denied
51	Directory structure has been damaged
52	Not a SOS volume
53	Invalid value in list parameter
54	Out of memory
55	Buffer table full
56	Invalid system buffer parameter
57	Duplicate volume error
58	Not a block device
59	Bad file level
5A	Invalid bit map address

```
BRK             ;Software interrupt triggers SOS call
.BYTE Callnum       ;Each call has an i.d. number
.WORD Params        ;Each call has a parameter list
```

Listing 1

```
PARAMS      .BYTE 03        ;Three parameters to come
            (first param)
            (second param)
            (third param)
```

Listing 2

When describing a SOS call, I will give the call's number (always one hexadecimal byte) and a description of its parameters. This description will give the order of the parameters, the name and type of each one, description of its use, and any other relevant information. I will also give an example of each SOS call. This reserves my place in documentors' heaven. However, please note that my examples will not contain any error checking, so beware.

Other notes of interest: some SOS calls have parameters that are optional; that is, the call can be made with or without these parameters. In these cases the call will have two special required parameters: a pointer to the optional parameter list, and a value which tells the number of optional parameters used. You can tell SOS you have zero optional parameters, in which case the pointer to the list is ignored. If this sounds a bit confusing now, it will probably become clear when you see it used in a SOS call.

Often a SOS call parameter will be a pathname, device name, or volume name. Whenever this occurs, a standard mechanism for the name is used. The parameter list will have a pointer to the name, and the name itself will consist of a byte giving the length of the name, followed by an ASCII representation of the name itself. In source code, it looks like Listing 3.

Those are the fundamentals, so let's get right into it!

File These Away for Reference

The first group of calls I'll present come from the file system. Some file calls work on closed files and some work on open files; none work on both. The trick to making a closed file into an open file is the OPEN call; the way to make an open file closed is with the (I can hear your mind racing) CLOSE call. We'll deal with some calls for closed files first.

CREATE

This call creates a new file on a block device, *i.e.*, a disk drive. Actually, it doesn't actually work with a closed file — it makes a new one.

call number: $C0

parameters: 3

1. Pathname pointer (2 bytes). The pathname of the file to be created.
2. Optionlist pointer (2 bytes). Points to the optional parameter list, if the Length (see next parameter) is between 1 and 8; otherwise, ignored.
3. Length value (1 byte). Length of the optional parameter list. Range is 0 through 8. Meaning:
   0 No optional parameters used
   1 or 2 File type parameter used
   3 File type and Aux type parameters used
   4 . . 7 File type, Aux type, and Stor type parameters used
   8 File type, Aux type, Stor type, and Eof parameters used

Optional parameters:

File type value (1 byte)
This byte tells the file's type. Range is 0 through FF.
Meaning: (the last column shows how the file is reported by the System Utilities filer)

```
00  typeless or unknown file (Unknown)
01  file containing bad blocks (Badfile)
02  Pascal or Assembly code file (Codefile)
03  Pascal text file (Textfile)
04  BASIC text or Pascal ASCII file (ASCIIfile)
05  Pascal data file (Datafile)
06  General binary file (Datafile)
07  Font file (character set) (Fontfile)
08  Screen image file (Fotofile)
09  BASIC program file (Basicprog)
0A  BASIC data file (Basicdata)
0B  Reserved (WPfile) ???
0C  SOS system file (SOSfile)
0D  Reserved (Datafile)
0E  Reserved (Datafile)
0F  Directory file (Directory)
10 - FF: Reserved (Datafile)
```

The file type defaults to 00 (unknown) if this optional parameter is not used.

Aux type value (2 bytes)
An auxiliary type identifier for the file. Used to store further information about the file. For example, BASIC uses this byte to store the record size of data files. Range is 0 through FFFF. The default is 0.

Stor type value (1 byte)
Indicates whether the file is a subdirectory (Stor type = D) or not (Stor type = 1). These are the only legal values. The default is 1.

Eof value (4 bytes)
Gives an amount of space in blocks to preallocate for a file. Files can grow and shrink dynamically, but if a file is known to be very large at creation type, using this parameter can help make access to it faster since the file will be contiguous. The range is 0 through FFFFFF. The default is 0.

An Example: (Listing 4)

Create a file named ASPHALT on the volume called MORK. The file will be used to contain a font.

```
            .WORD PATHNAME      ;Pointer to the name

PATHNAME    .BYTE 09            ;Length of the name itself
            .s095ASCIIs100 "/JOE/FRED"   ;Pathname is /JOE/FRED
```

Listing 3

```
        BRK             ;SOS call
        .BYTE 0C0           ;CREATE
        .WORD CR_PARAMS     ;pointer to parameters

CR_PARAMS   .BYTE 03        ;3 parameters
        .WORD CR_PATH       ;pointer to file pathname
        .WORD CR_OPTNS      ;pointer to optional params
        .BYTE 01          ;use File_type optional param

CR_PATH     .BYTE 0D        ;length of name
        .s095ASCIIs100 "/MORK/ASPHALT" ;pathname

CR_OPTNS    .BYTE 07        ;font file
```

Listing 4

DESTROY

As long as we're creating 'em, we might as well destroy some, too. This call deletes a file from a block device.

call number: $C1

parameters: 1

1. Pathname pointer (2 bytes)
   The pathname of the file to be destroyed.

An Example (Listing 5).

Delete a file called LENDER in a subdirectory called HAPPY.TIMES on a volume named THURSDAY.

OPEN

Before we can read from or write to a file, we have to open it. This is call that performs that function.

call number: $C8

parameters: 4

1. Pathname pointer (2 bytes)
   The pathname of the file to be opened.
2. Refnum result (1 byte)
   When a file is opened, SOS assigns it a reference number (refnum). This number is then used in subsequent reads and writes with that file.
3. Optionlist pointer (2 bytes)
   Points to the optional parameters list, if the Length (see next parameter) is between 1 and 3; otherwise, ignored.
4. Length Value (1 byte)
   Length of optional parameter list. Meaning:

   0 No optional parameters used
   1 . . 3 Req access parameter used

Optional Parameters:

Req access value (1 byte)
Allows the file to be opened only for reading or only for writing. Range is 0 through 3.

Meaning:

00 open for as much access as permitted
01 open for reading only
02 open for writing only
03 open for reading and writing

The access defaults to 0 (open for as much access as permitted) if this optional parameter is not used.

An Example (Listing 6).

Open the file we created earlier (/MORK/ASPHALT).

After this file is opened, we would use the result returned at location OPEN-REF to refer to this file in read and write calls (read on!).

WRITE

This is the call you use to transfer information from a buffer to a file.

call number: $CB

parameters: 3

1. Refnum value (1 byte)
   The Refnum assigned to the file when it was opened.
2. Buf pointer (2 bytes)
   Points to a buffer area where the information to be sent comes from.
3. Bytes value (2 bytes)
   The number of bytes to be written.

An Example (Listing 7).

Write 10 bytes to the file we opened earlier (/MORK/ASPHALT).

Executing this call after using the preceding OPEN to open the file and get the Refnum would cause the 10 bytes listed above to be written to the file. Remember that when we created this file, we gave it a 'Type' parameter indicating that it was to contain a font. However, when dealing with files at the SOS call level, SOS doesn't really care what a file contains or is supposed to contain — it simply reads and writes data.

```
        BRK             ;SOS call (but you knew that already)
        .BYTE 0C1         ;DESTROY's i.d. number
        .WORD DES_PARAMS     ;pointer to parameters

DES_PARAMS    .BYTE 01          ;1 parameter
        .WORD DES_PATH      ;pointer to file pathname

DES_PATH     .BYTE 1C          ;length of name
        .s095ASCIIs100 "/THURSDAY/HAPPY.TIMES/LENDER" ;pathname
```

Listing 5

```
        BRK          ;Guess what (have you been reading along?)
        .BYTE 0C8         ;I.D. number for OPEN
        .WORD OPEN_PARAMS     ;pointer to parameteters

OPEN_PARAMS    .BYTE 04          ;4 parameters
        .WORD OPEN_PATH      ;pointer to file's name
OPEN_REF     .BLOCK 1          ;reserve 1 block for Refnum result
        .WORD 0000         ;we're not using any optional params...
        .BYTE 00          ;...so we make these all zeroes

OPEN_PATH     .BYTE           ;length of name
        .s095ASCIIs100 "/MORK/ASPHALT" ;the pathname itself
```

Listing 6

```
        LDA OPEN_REF       ;move the Refnum we obtained earlier
        STA WRITE_REF       ;into WRITE's parameter list
        BRK             ;call up SOS (hello, SOS?)
        .BYTE 0CB         ;call i.d. number
        .WORD WRIT_PARAMS     ;pointer to parameters

WRIT_PARAMS    .BYTE 03          ;3 parameters
WRIT_REF     .BLOCK 1          ;the above STA puts the proper Refnum
                    ;value in this byte
        .WORD DATA_BUF      ;pointer to our data buffer
        .WORD 000A         ;write 10 decimal (0A hex) bytes

DATA_BUF     .BYTE 01,23,45,67,89,AB,CD,EF,FF,FF ;10 randomly
chosen data
```

Listing 7

READ

This call attempts to transfer a given number of bytes from a file to a specified buffer. The other half of the world-famous read/write team!

call number: $CA

parameters: 4

1. Refnum value (1 byte)
   The Refnum assigned to the file when it was opened (as in the WRITE call).
2. Buf pointer (2 bytes)
   Points to a buffer area where the information will be placed after it is read (again, note the symmetry with the WRITE call).
3. Bytes value (2 bytes)
   The number of bytes to be read.
4. Bytes-read result (2 bytes)
   SOS returns the number of bytes actually read in these locations.

An Example (Listing 8):

Attempt to read 10 bytes from the file we opened earlier, named (/MORK/ASPHALT).

Where's **DATA-BUF**? Remember, we defined it in the WRITE call. Can this buffer area be reused? Sure! In fact, that's one of the benefits of SOS's system of parameter lists and pointers. You can use the same area in memory as a read and write buffer.

If you executed this call after just having written to the file earlier (as we have done in this article), you would get an error #4C, End of file. Wait a minute, you may say — we just wrote 10 bytes of data, so why won't they be read? The answer lies in the fact that whenever SOS reads from or writes to a file, it maintains a pointer, or mark, into that file, kind of like a book marker, so that it knows where to read from or write to next. After we wrote the 10 bytes out (with our example WRITE call), that marker was pointing to the end of file. When the subsequent READ came up, there was nothing left to read.

What do you do if you want to move the mark without reading or writing anything? Why, there just happen to be a couple of SOS calls (GET-MARK and SET-MARK) that let you look at and modify the mark. I won't go into them in depth here, but be advised of their existence.

CLOSE

This is the call to use to finish up the use of an open file.

call number: $CC

parameters: 1 (this is a simple one)

1. Refnum value (1 byte)
   The Refnum assigned to the file when it was opened.

An Example (Listing 9).

Close the file we've been working with.

You now have the basic tools necessary to work with SOS's file system. **CREATE** makes the files, **OPEN** gets them ready for reading and writing, **READ** and **WRITE** perform the actual transfer of data to and from the files, **CLOSE** finishes the reading and writing process, and **DESTROY** gets rid of the files.

I Promised You an Interpreter

Yes, I did, way back at the beginning of this article, say that we'd create a real, working interpreter before we were done, and we're about to do just that. Just as our first SOS call and module examples were simple, we'll begin with a fairly mindless interpreter. This one will simply print a welcoming message on the screen and then sit there. Not terribly exciting, I admit, but we need a place to start! (We'll get fancy later).

As noted earlier, **SOS.INTERP** files start with a special header block, then get right into the code. Well, our code will consist of three things:

1. OPENing the .CONSOLE device (so that we can print on the screen).
2. WRITing the message to the .CONSOLE.
3. Looping infinitely.

Since the how-to of all this stuff has been explained, let's proceed with the source text listing, Listing 10.

```
        LDA OPEN_REF        ;move the Refnum we obtained earlier
        STA READ_REF        ;into READ's parameter list
        BRK             ;now call SOS
        .BYTE 0CA           ;call i.d. number
        .WORD READ_PARAMS     ;pointer to parameters

READ_PARAMS   .BYTE 04          ;4 parameters
READ_REF     .BLOCK 1         ;our STA instruction above loads this
;byte with the proper Refnum value
        .WORD DATA_BUF      ;pointer to the buffer where data read
                        ;will go
        .WORD 000A          ;0A hex is 10 decimal; read 10 bytes
BYTES_READ    .BLOCK 2          ;reserve two bytes for SOS to put the
;number of bytes actually read
```

Listing 8

```
        LDA OPEN_REF        ;as we did with READ and WRITE,
        STA CLOSE_REF       ;bring in the desired Refnum
        BRK             ;then do the SOS call itself
        .BYTE 0CC           ;CLOSE call i.d. number
        .WORD CLOSE_PARAMS    ;parameter list

CLOSE       .BYTE 01          ;one parameter only
CLOSE_REF     .BLOCK 1          ;reserve a space for the Refnum
```

Listing 9

```
;first, some administrative stuff

        .ABSOLUTE       ;required for interpreters
        .PROC MYINTERP    ;this is the title (clever, huh?)

START     .EQU 0B000      ;code will load here
        .ORG START-0E     ;move back 14 bytes for header

;required header follows

HEADER    .ASCII "SOS NTRP"    ;required header information
        .WORD 0000        ;no optional header block (length 0)
        .WORD START       ;loading address
        .WORD CODELENG    ;length of code

;this is the working program

        BRK             ;the code itself: OPEN call
        .BYTE 0C8
        .WORD OP_LIST

        LDA OP_REF        ;put file's Refnum in WRITE's
        STA WR_REF        ;parameter list

        BRK             ;WRITE call
        .BYTE 0CB
        .WORD WR_LIST

LOOP      JMP LOOP        ;run around in circles forever

;parameter lists come next

OP_LIST     .BYTE 04        ;OPEN has four parameters
        .WORD CONS_PATH     ;pointer to the pathname to open
OP_REF      .BLOCK 1        ;reserve a place for SOS to put Refnum
        .WORD 0000        ;no optional parameters needed...
        .BYTE 00        ;...so these are zeroes

CONS_PATH   .BYTE 08        ;length of pathname
        .ASCII ".CONSOLE"   ;the file to open

WR_LIST     .BYTE 03        ;three parameters for WRITE
WR_REF      .BLOCK 1        ;save space for Refnum
        .WORD WR_BUF      ;pointer to our data
        .WORD 001F        ;length of our message

;the greeting message

WR_BUF      .ASCII "Hi, I'm Irving the interpreter!"

;close up shop

CODELENG    .EQU *-START      ;figures length of code for header

        .END            ;all done
```

Listing 10

```
program interp_maker;  { By Alan Anderson; from Apple Orchard }

  var
   infile, outfile : file;
   inname, outname : string;
   data : packed array [1..512] of 0..255;
   block_num, count : integer;


begin
  write ('Enter the pathname of the codefile to be converted
-->');
  readln (inname);
  write ('Enter the pathname for the output file -->');
  readln (outname);
  reset (infile, inname);
  rewrite (outfile, outname);
  count := blockread (infile, data, 1);
   while not eof (infile) do
     begin
      count := blockread (infile, data, 1);
      count := blockwrite (outfile, data, count);
     end;
  close (infile);
  close (outfile, lock);
end.
```

Listing 11

That's it! Type Listing 10 in the Pascal Editor and assemble it and . . .you're almost there. There's one more item to consider: When the Pascal assembler writes a codefile, it writes a single block of information which is placed at the front of the codefile. However, although the assembler always writes this block, the information therein is useful only for modules, and this block must be removed from the front of interpreter files. The ideal solution to this situation would be a pseudo-op (called, perhaps, .MAKEINTERP), which would generate files without the information block. The current solution, though, is to have a Pascal program to rewrite the file without the information block.

When the *Apple /// SOS Reference Manual* is distributed, Apple plans to include a program to perform this function. Until then, here is a program, Listing 11, (without any error checking) to accomplish the same purpose.

After you've assembled the interpreter listed above, enter and compile this program, then execute it and convert the codefile. The final product is now a real, live, almost useful interpreter. What do you have to do to use it? Just format a diskette and put **SOS.KERNEL** and **SOS.DRIVER** on it. Then copy the converted codefile from our interpreter maker to the new disk and call it (naturally enough) **SOS.INTERP**. If you've done everything right and the stars are smiling upon you, you should then be able to boot the diskette and have it say "Hi" to you! All right!

You have just created an Assembly language program which executes all by itself, without BASIC or Pascal or any other high level language hanging around. Although it performs no useful function other than as a demonstration, it allows you to view the basic structure needed to write your own interpreters.

I hope all these goodies about interpreters and SOS calls will be enough to keep you going until next time. If not, please write to me. I can be had at:

Alan Anderson
c/o **Apple Orchard**
910 A George St
Santa Clara, CA 95050

Next time, I'll probably present some more tools for programming the bejeebers out of the Apple ///, probably in the form of more SOS calls and ways to exploit all the power in the .CONSOLE driver. However, this is changable according to your whims, so let me know what you want.

Okay, everybody. . .

HIT THE SOS!