Chapter 5
File Input and Output

Introduction

HForth supports UNIX-like *File I/O*. Filenames can include *logical volume names* which will be expanded to *full pathname*s. See the introductory chapter of this manual for more information on **ASSIGN**. The file I/O words are divided into Host Independant and Mac Specific words.

File I/O Tutorial

When entering this tutorial be sure to enter it exactly as written, especially when reading files.. Otherwise you could overwrite memory causing a harmless but annoying crash.

Creating a Text File

Let's create a new file. We can store some text in the file then read it back out. If we just use a filename without a volume name, the file will be created in the same folder as HMSL. Enter:

```
VARIABLE MYFILE
NEW FOPEN FILE1 .S
MYFILE !
```

We just created a NEW file called "FILE1". The number that was returned by FOPEN is a pointer to a special structure that we can use to access that file. We don't have to worry about what is in that structure. Just consider it as a unique identifier for that file. We can have multiple files open and refer to each of them by their file pointer. We saved the file pointer in a variable called MYFILE because we will need it later.

It is possible that your disk was full. If so, FOPEN would have returned a ZERO for a file pointer. It is important to check to make sure that the file pointer is not ZERO before proceeding. That is why we used .S to show the pointer. If you got a zero from FOPEN, try again using a blank formatted disk named TEST and use a filename of TEST:FILE1.

Now let's write to the file. Enter:

```
MYFILE @  ( get the file pointer )
 " Important Information" COUNT .S
FWRITE .
```

FWRITE expects a file pointer followed by an address and count. It writes the string to the file and then returns the number of characters written. If there is an error, it will return a -1.

If we are going to write more lines to the file, we should separate them with and "End Of Line" character. EOL is a constant equal to the character used to separate lines in a file. On the Amiga, this is an ASCII Linefeed. If we write again to the file, the new data will go right after the previous data. This is because file I/O uses an imaginary cursor that points into the file. This type of I/O is called "sequential I/O" because bytes are read or written one after the other in sequence. Enter:

```
MYFILE @  EOL  FEMIT
```

FEMIT is a handy word that uses FWRITE to write a single character to a file. We can now write another line to the file.

```
MYFILE @  " COST = 23"  COUNT  FWRITE .
MYFILE @  EOL  FEMIT
```

When we are finished with a file we should close it. Enter:

```
MYFILE @  FCLOSE
```

We have now opened a file, written data to it, and closed it. We can see the result of our work by opening the file in the text editor, or by entering:

```
TYPEFILE FILE1
```

Reading a Text File

Now let's open that file, and read what we wrote. Enter:

```
FOPEN FILE1 .S
MYFILE !
```

We don't need to say NEW because we are opening an existing file. Now let's read the first 8 characters from the file. Enter:

```
PAD 200 ERASE  ( clear PAD )
MYFILE @  PAD 8  FREAD .
PAD 8 TYPE
```

We should see the number 8 printed after the FREAD which is the number of characters read. The data was stored at PAD which we saw using TYPE . Let's now read the rest of the file. Reading a file uses a cursor just like when writing. We are now positioned after the 8th character and can read from that point. Enter:

```
MYFILE @  PAD 100 FREAD .
```

Notice that the number printed was less than 100. The number reflects the actual number of bytes read. Since we reached the end of the file, we got fewer bytes than we asked for. This is one way to tell when you reach the end of a file. We can look at our data by entering:

```
PAD 30 DUMP
```

Now let's close our file. Enter:

```
MYFILE @  FCLOSE
```

Using Binary Data Files

We can also use files to store numbers in the form of binary data. In fact, anything in memory, arrays, structures, parts of the dictionary, whatever, can be written to a file using FWRITE. Let's create an array of numbers then store them in a file. We should put a count of how many numbers there are at the beginning of the file so we know how to read it later. First let's make an array of data to use. Enter:

```
CREATE MYDATA 123 ,  2931 ,  7 , 99712 ,  49 ,
VARIABLE NUM-ITEMS
5 NUM-ITEMS !
```

Now let's make a file to store this data in. Enter:

```
NEW  FOPEN  BDATA .S
MYFILE !
```

At the beginning of the file we should store the number of 4 byte data cells that will follow. This will help us later when we want to read the file. Enter:

```
MYFILE @  NUM-ITEMS  4  FWRITE .
```

This wrote the 4 bytes at the address NUM-ITEMS to the beginning of the file. In other words, we just wrote the contents of the variable NUM-ITEMS to the file. Now let's write the data.

```
MYFILE @  MYDATA  NUM-ITEMS @ CELLS  .S
FWRITE .
```

Each number in MYDATA occupies 4 bytes or 1 cell. By calling CELLS we calculate how many bytes the table of numbers occupies.

Rather than close the file and reopen it, let's just reposition ourselves to the beginning and start reading. The word FSEEK will move our cursor to anyplace in the file. We can move to a location relative to our current position, or relative to the beginning or end. Let's move to the beginning of the file.

```
MYFILE @  0  OFFSET_BEGINNING FSEEK .
```

The number printed was our old position in the file. (You can move zero bytes relative to your current position to find out where you are!) We can now read the number of data items in the file. Enter:

```
0 NUM-ITEMS !
NUM-ITEMS ?
MYFILE @  NUM-ITEMS 4 FREAD .
NUM-ITEMS ?
```

NUM-ITEMS now contains the number of data items in the file. Let's read the data. Enter:

```
MYFILE @  PAD  NUM-ITEMS @ CELLS FREAD .
PAD @ .  ( print 123 )
PAD 8 + @ .  ( print 7 )
```

The data is now stored on the PAD.

Sometimes, a data file can be so big that we don't want to load the whole thing into memory. You can write a word that will read randomly from a given position in a file. This word will check for errors when seeking. FSEEK will return a -1 if you have an error. A common error is tryin to go outside the bounds of the file. You may want to enter this example in a file for future use. Enter:

```
: GRABDATA  ( item# -- item , read an item )
\ Calculate offset, skipping count at beginning.
    CELLS CELL+
\ Position cursor in file.
    MYFILE @  SWAP OFFSET_BEGINNING  FSEEK
    0<  ABORT"  File Seek Failed!"
\ Read the number.
    MYFILE @  PAD 4 FREAD
    4 = NOT ABORT" File Read Failed!"
    PAD @
;
0 GRABDATA .  ( print 123 , the first item is # 0 )
3 GRABDATA .  ( print 99712 )
70 GRABDATA .  ( should report failure )
```

Now close the file. Enter:

```
MYFILE @  FCLOSE
```

This demonstrated the use of a simple binary data file. Very complex files, like the IFF files can also be accessed with these techniques.

Host Independant Words

These words will work on the both Amiga and Macintosh versions of HMSL. Code written using these words will work without modification on both machines. The only difference is that filenames on the Amiga, and most other machines, use '/' as a separator while the Macintosh uses ':'s. If you want filenames to be host independant, use logical volume names (see ASSIGN). Names like "HWORK:MY-DATA" will work the same on both machines.

$FOPEN ($filename -- refnum | 0 , open a file)

Attempt to open a file. The filename is passed on the stack as a string. Return a refnum if successful. Otherwise return a NULL or 0.

FILEWORD (<filename> -- $addr , parse file name from input)

If you have a file that has spaces in the name, then you cannot use WORD to get the filename because it will only get the first word up to the space. FILEWORD will check to see if the first letter of a filename is a ", if so it will parse up to the next " for the end of the name. This name can then be passed to words that use $FOPEN.

```
: TESTF  ( -- ) FILEWORD COUNT TYPE ;
TESTF  mydata
TESTF "name with spaces"  ( this will work!)
```

FOPEN (<filename> -- refnum | 0 , open a file)

Attempt to open a file. The filename follows FOPEN (this is the difference between this word and $FOPEN). Return a refnum if successful. Otherwise return a NULL or 0.

FEMIT (refnum char -- , emit character to file)

This will send a single character to the file. If it fails, it will abort.

FKEY (refnum -- char , read character to file)

This will read a single character from the file. If it fails, it will abort.

FREAD (refnum addr num_bytes -- bytes_read)

Read bytes from the file starting at the current position to the address given. Return the actual number of bytes read. If you hit the end of file, you will read fewer bytes then requested. The current file position will be left after the last byte read.

FWRITE (refnum addr num_bytes -- bytes_written)

Write bytes from the address given.to the file starting at the current position. Return the actual number of bytes written. If you run out of disk space, you will write fewer bytes then requested. The current file position will be left after the last byte written.

FSEEK (refnum offset mode -- old_position)

Change the current position of the file. Three modes are allowed. They are defined as constants for use with FSEEK:

```
OFFSET_BEGINNING  ( -- n , offset from beginning of file)
OFFSET_CURRENT  ( -- n , relative to current position )
OFFSET_END  ( -- n , negative offset from end of file)
```

A value of zero will either put you at the beginning of the file, have no effect, or move you to the end of a file for each of these modes.

FCLOSE (refnum -- , close file opened with FOPEN)

NEW (-- , set flag so that next FOPEN will create a file)

Macintosh Specific Words

The Macintosh has a file system that is unlike that on other computers. For example, many Macintosh file functions require a "Volume Reference Number". These words support features, and address problems, that only exist on the Macintosh. These words will not work with HMSL on other machines.

$ASSIGN ($logical-name $real-name -- , see ASSIGN)

This allows you to set a logical name when the real name has a blank character in it. The names must be in a colon definition. For example:

```
: DO.ASSIGN
   " music:" " Hard Disk:Music Tools" $assign ;
```

ASSIGN on the Amiga is part of the operating system.

$FCREATE.VR ($filename vRefNum -- error | 0)

$FOPEN.VR ($filename vrefnum -- fileid | 0)

Similar to $FOPEN except this allows the specification of a volume reference number.

ASSIGN (<logical-name> <real-name> --)

The word allows you to assign "nick names" to folders on the Macintosh. This can save typing. It also allows you to move files or change the names of folders with having to change all the code that refers to that folder. Just change the assignment. As an example, suppose you kept all the files for a

piece you are working on in a folder called "77MHD:MUSIC:HMSL:PROJECT1". You could assign a short name to this folder by entering:

```
ASSIGN  P1: 77MHD:MUSIC:HMSL:PROJECT1
```

You could then refer to a file called TUNINGS in that folder as:

```
P1:TUNINGS.
```

This would be equivalent to typing:

```
77MHD:MUSIC:HMSL:PROJECT1:TUNINGS
```

See the introductory chapter and the Glossary for more information.

ASSIGNS? (-- , print current assignments)

EXPAND.FILENAME ($logical-name -- $full-name)

Convert a filename. For example:

" HP:XFORMS" EXPAND.FILENAME $TYPE

CD (<volume-name> -- , change directory)

Change the default directory(folder). Similar to the CD command in UNIX. Calls SetVol. For example:

```
INCLUDE HP:SPLORP
```

--or--

```
CD HP:
INCLUDE SPLORP
```

FERROR (-- var-addr , variable containing errors)

After performing a file operation, this variable will be set to the result code. Use REPORT.MAC.ERROR to interpret the number.

FEXPECT (addr max --)

Similar to EXPECT but it reads a line from a file whose FILEID is in a variable called FSOURCE.

FFLUSH (fileid --)

WARNING!!! The Macintosh does not really write everything to the disk when you use FWRITE or FCLOSE! It stores some of the information in a buffer then writes it all to disk when you finally eject it or when the buffer is full. The Mac does this in an attempt to speed up file I/O. Unfortunately, however, if you crash before ejecting the disk the information can be lost! Sometimes the entire disk can be lost. Fun huh? This word allows you to flush the file so that it is all on disk. After closing the file you will probably want to call this word, and FLUSHVOL.

FDELETE ($filename vRefNum -- result)

Delete a file from disk.

FILE-CREATOR (var-addr -- , holds current creator)

This is used by NEW FOPEN. It is reset to 'HMSL' after use.

FILE-TYPE (var-addr -- , holds current type)

This is used by SFGETFILE and FOPEN. It is reset to 'TEXT' after use. See SFGETFILE.

FLUSHVOL (vRefNum --)

See FFLUSH.

FOPEN.ASK.PUT ($origname x y $prompt -- $filename fileid true | 0)

Uses SFPUTFILE to open a file for writing.

FREWIND (fileid -- oldpos | -1)

FSOURCE (-- var-addr , see FEXPECT)

FTRUNCATE (fileid --)

This writes an EOF (end of file) at the current position. If you want to reduce the size of a file, you must call this to avoid having junk left at the end of the file.

REPORT.MAC.ERROR (error# -- , try -35)

Print a message corresponding to the given Macintosh error number.

SFGETFILE (-- $filename vrefnum true | false)

SFGETFILE, 5-Puts up a standard file open dialog to allow the user to select a file in a familiar manner. (Making this a standard system call is the smartest thing Apple ever did!) Uses FILE-TYPE and FILE-CREATOR to select the desired file type. For example:

" Midi" 1+ ODD@ FILE-TYPE ! SFGETFILE

Try this then look in the Pieces folder and the MIDIFiles folder.

SFPUTFILE ($origname x y $prompt -- $filename vRefNum true | 0)

Puts up a standard dialog for creating a new file. Uses FILE-TYPE and FILE-CREATOR. For example:

```
: ASK.FILE  ( $filename vRefNum true | 0 )
   " Untitled"  50 30
   " Name to save data as..."  SFPUTFILE
;
```

File Dialogs

To simplify the task of writing interactive application on both machines, we have provided host independant file open dialogs. The Macintosh version uses the SFGETFILE and SFPUTFILE dialogs from above. The Amiga version will use the ARP or REQ library.

DIALOG.GET.FILE ($prompt -- $filename file-refnum true | 0)

Ask the user to select a file by putting up a dialog box, then opens it. This is intended for Opening or Loading operations. If the user selects CANCEL, then it will return 0.

DIALOG.PUT.FILE ($original $prompt -- $fname refnum true | 0)

Puts up a dialog box that prompts the user and suggests a default filename. "Untitled" is a good default name. It then creates and opens a file for output or saving.

Logging to Files

The output of HForth can be made to echo to a file as well as the keyboard by changing the deferred word TYPE. This is useful for generating records of your work, dumping memory to a file for later analysis, etc. If you are trying to generate formatted data files, you can get the code to work properly on the screen first. Then just log your output to a file.

$LOGTO ($filename -- , Send copy of output to file.)

LOGTO (<filename> -- , Get filename and pass to $LOGTO)

LOGSTOP (-- , Temporarily stop echoing.)

LOGSTART (-- , Continue echoing, used after LOGSTOP)

LOGEND (-- , Stop echoing, close file opened by LOGTO)

As an example, if you want to dump some memory and have it saved, enter the following:

```
LOGTO SAVEDUMP
```

```
1000 200 DUMP
LOGEND
```

File Transfer Between Mac and Amiga

See the main manual for a description of the MIDI File Transfer Utility, a way to transfer files between the Macintosh and Amiga using a MIDI interface.