Chapter 8
File I/O

This chapter describes the JForth words that provide a simple, complete interface to the AmigaDOS file routines. A file is simply a collection of bytes that can be stored on a disk or in memory. Any kind of data can be stored in a file. The bytes may be ASCII characters as in a document or a program's source code. They could also be data from an experiment or binary code for execution by the computer.

As you might expect, support is provided to perform the standard file operations. These are:

1. Opening a file for use and getting a "file pointer" that is used when referring to a file.
2. Using the file-pointer to read the data, write new data and/or reposition yourself in the file, *seek*.
3. Closing the file, again using the identifying file-pointer, when finished.

Some additional tools have been provided to round out the stock supply of file utilities. These may not be needed in a typical application. These involve the generation of NUL terminated file names which Amiga DOS requires internally, and fast buffered I/O.

Note that the Amiga OS already provides interfaces to some types of Amiga system files, such as icons (those that end in *.info*), libraries and others. Such files are not normally accessed with these functions, but with the Amiga-supplied ones.

Before we get into too much detail, let's explore some of these tools in a tutorial.

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. Rather than create a file on floppy disk, let's make one on the RAM: disk. Enter:

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

We just created a NEW file called "FILE1" on the volume RAM:. 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 RAM: disk was full which can happen if you are low on memory. 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 formatted disk in DF1:, with a filename of "DF1: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 entering in the CLI window:

```
TYPE  RAM:FILE1
```

or in JForth:

```
TYPEFILE RAM:FILE1
```

Reading a Text File

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

```
FOPEN RAM: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
```

JForth provides several tools that simplify reading text files. These include READLINE and DOLINES which will be discussed later.

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  RAM: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. See the JIFF:IFF_SUPPORT or JU:SHOWHUNKS for more examples.

File I/O Reference

Opening Files

Prior to reading from or writing to a file, it must be 'opened'. JForth provides four words concerned with opening files.

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> -- file-pointer | false , opens file)

This reads the filename from the input stream using FILEWORD, opens the file and returns a pointer to a file control structure.

```
FOPEN  DF1:DATAFILE
```

If you do not specify a pathname, Amiga DOS will default to the current directory as set by the CD command.

0FOPEN (0name -- file-pointer | false , opens file)

In this case the filename is a NUL terminated string passed on the stack.

```
0" DF0:THISFILE"  0FOPEN
```

$FOPEN ($name -- file-pointer | false , opens file)

This accepts a standard Forth string, with a count byte. You should use FILEWORD instead of word if you want to get a filename from input.

```
" DF0:THISFILE"  $FOPEN
```

If the file could not be opened, these words return false.

Files are normally opened as existing, read/write. This means that the file specified must exist, its contents will be preserved across the open, and both read and write operations are allowed.

Another word, NEW (--), may precede the 'FOPEN' word to create a new file, or clear the contents of an existing file.

The 'FOPEN' words access a variable called FILEMODE to determine the desired mode for opening. NEW places the value MODE_NEWFILE there, OLD replaces it with MODE_OLDFILE. Note that, at the end of every 'FOPEN' operation, the mode will be reset to OLD .

NOTE: Do NOT execute NEW unless it is immediately followed by the open' operation. It's UNNERVING to clear a wanted file on open, just because you executed NEW and forgot about it!

It is important to check the results of a file being opened because errors can easily occur. Open errors are typically due to the file not being found because the name is incorrect or you are in the wrong directory.

```
: OPENFILE ( <name> -- , open a file )
     FOPEN  ( gets name from input stream )
     DUP \ save the file-pointer in a variable
     IF   MYFILE !
     ELSE CR ." File could not be opened!"  QUIT
     THEN
;
```

Reading and Writing to files.

The words supplied in JForth are a high-level interface to the AmigaDOS calls READ, WRITE, and SEEK. One minor difference in their use is that all addresses passed as parameters are relative addresses. They are converted to absolute (required by Amiga calls) within the function.

Each function requires a 'file-pointer', which will have been acquired via FOPEN , 0FOPEN or $FOPEN . File pointers are not considered addresses and are used just as AmigaDOS returns them; they are NEVER converted to relative.

Their names and stack diagrams are as follows:

FEMIT (file-pointer char -- , emits character to file)

This will abort if an error occurs.

FKEY (file-pointer -- char , gets character from file)

This will abort if an error occurs. It is not recommended that this be used in commercial aplications because it does not handles gracefully. But it is handy.

FREAD (file-pointer addr cnt -- #read | -1)

The FREAD and FWRITE functions are straightforward in their operation, operating on the specified memory and file (at its current address). Each return the number of bytes processed, or -1 if an error occurred.

FWRITE (file-pointer addr cnt -- #written | -1)

FSEEK (file-pointer filepos mode -- prevpos | -1)

The FSEEK 'mode' parameter equates directly to the AmigaDOS declared parameters OFFSET_BEGINNING, OFFSET_CURRENT, and OFFSET_END. For example, to seek to the end-of-file minus 5 bytes:

```
MYFILE @  -5  OFFSET_END  FSEEK .
```

These 5 calls will also set a user variable, FERROR, if appropriate. (If you want to check FERROR, do so immediately after the function returns. It will be reset by the next file operation.)

Following are various examples of reading from and writing to a file after it has been opened and the file-pointer stored in a VARIABLE called MYFILE.

Change the current location to the beginning of the file:

```
MYFILE @  0  OFFSET_BEGINNING FSEEK  ( -- ret-code )
```

Read 100 bytes from current position, place them at PAD:

```
MYFILE @  PAD  100 FREAD   ( -- ret-code )
```

Write 100 bytes from PAD to the file at its current location:

```
MYFILE @  PAD  100 FWRITE  ( -- ret-code )
```

Closing Files.

The normal method of closing a file opened under JForth is:

FCLOSE (file-pointer -- , return file resources to AmigaDOS)

A normal program then, will use FOPEN when it starts, and FCLOSE at its completion. In development environments, however, applications often will not finish as an error condition may cause it to QUIT. In JForth, you may optionally mark your file to be automatically closed in this event by executing MARKFCLOSE on a duplicate of the just-opened file-pointer. If your application successfully completes, you should UNMARKFCLOSE your file(s), so that they are not closed multiple times.

MARKFCLOSE (file-pointer -- , mark file to auto-close at quit)

UNMARKFCLOSE (file-pointer -- , remove from 'auto-close' stack)

Example: illustrates FOPEN, MARK and UNMARKFCLOSE and FCLOSE

```
: EXAMPLE   ( -- , parses a filename from the input stream )
  FOPEN  ( -- file OR false )  ?dup
  IF  dup MARKFCLOSE   ( file -- )
              \ auto-close it at QUIT.
      MYFILE !         ( -- )
              \ save it in my variable
      Do-My-Thing      ( -- )
             \ do file processing, whatever it is
      MYFILE @         ( file -- )
             \ fetch the file pointer
      dup UNMARKFCLOSE ( file -- )
             \ remove from the auto-close stack
      FCLOSE           ( -- )
             \ and close it!
  ELSE  cr ." File could not be opened!"  QUIT
  THEN
;
```

Building AmigaDOS Filenames.

Three words help you build null-terminated strings for AmigaDOS, but are not normally needed; FOPEN can usually be used to specify a file. These are handy to modify filenames algorithmically and resubmit them to AmigaDOS (via the 0FOPEN word). They are:

DOS0 (-- addr , returns the address of the NUL-string buffer)

>DOS (addr cnt -- , place string in DOS0, NUL terminated)

+DOS (addr cnt -- , append this string to one already at DOS0)

Note that >DOS and +DOS maintain a count byte usable by the JForth string words. For example, the contents of DOS0 can be typed by:

```
DOS0 1- count type
```

Here is an example of using these words to build a file pathname.

```
: FOPEN.DATA  ( <name> -- , append suffix and open )
```

```
    FILEWORD COUNT >DOS
    " .DATA" COUNT +DOS
    DOS0  0FOPEN
;
FOPEN.DATA  EXPT1  ( open "EXPT1.DATA" )
```

Sequential Virtual File Utilities

Several words provided allow easy use of a 1024 byte virtual buffer area for file words designed to sequential single-character or cell-based transfers. Using these words, a program may realize a significant speed improvement for certain types of file I/O.

An application, at its start, may open a file-virtual buffer and store the resultant address in a variable. The buffer may be used by passing the address of the variable (not the buffer) as an argument to certain virtual-calls.

Also, some JForth-provided functions (such as READLINE) are only accessible through this scheme.

Those words dealing with virtual buffer management include:

OPENFV (var-addr -- buffer-addr)

Allocate a 1K buffer, and set the variable to its address. Even though the buffer address is not normally needed by applications, it is returned. If a zero is returned, an error occured.

CLOSEFVREAD (var-addr --)

Deallocate the buffer being pointed to by VAR. This buffer has only been read from. Clears the variable.

CLOSEFVWRITE (file-pointer var-addr --)

Flush any leftover data to the file, deallocate the buffer and clear the variable. (This buffer may have only been used for writing).

F, (file var n1 --)

Send 'n1' (32 bits) to the next available cell in FILE via the virtual buffer stored in VAR.

READLINE (file var addr-addr maxcnt -- addr cnt | addr -1)

Read FILE via the buffer stored in VAR, place at ADDR, do not exceed MAXCNT characters. Returns 0 if an empty line, -1 if end-of-file.

TEMPF, (n1 --)

Same as 'F,' but uses TEMPFILE and TEMPBUFF.

As an example, observe this simple word which opens the file whose name follows in the input stream, then types each line to the screen until the end of file. (Note: TEMPFILE & TEMPBUFF are user variables that are pre-defined in JForth for such uses.)

```
: LISTFILE   ( -- , eats name )
\ types inputted FILENAME to console
   FOPEN -dup
   IF   TEMPFILE !    \ save file pointer
\ allocate virtual buffer
      TEMPBUFF OPENFV ( addr -- )
      drop
\ tempbuff inited by OpenFV, don't need addr
      BEGIN
```

```
            TEMPFILE @  TEMPBUFF
            HERE 1000   READLINE
            DUP 0 < 0=
            ( addr #read true-if-not-eof -- )
         WHILE  CR TYPE
         REPEAT 2DROP
         TEMPBUFF CLOSEFVREAD   \ deallocate the buffer
         TEMPFILE @ FCLOSE           \
      ELSE cr ." Can't open "  DOS0 1- count type  quit
      THEN cr
   ;
```

DOLINES - Easy Text File Processing

The DOLINES system provides a simple way to process text files on a line by line basis. You can set a deferred word that will get called for each line of the file. It will be passed the line as a string. You can then do whatever you want with that string. Here is an example of a program that types a file to the screen.

First define a word that will process each line as it is read.

```
INCLUDE? DOLINES JU:DOLINES

: SHOWLINE ( $line -- , type it with line number )
    CR DL-LINENUM @ 5 .R
    SPACE  $TYPE ?PAUSE
;
```

DL-LINENUM is the line number that is maintained by DOLINES. Now write a word that will set the vector and call DOLINES.

```
: SHOWFILE ( <filename> -- , print file to screen )
    ' SHOWLINE IS DOLINE ( set deferred word )
    DOLINES
;
SHOWFILE JU:BSORT
SHOWFILE JU:ANSI
```

Once the vector DOLINE is set, you can call DOLINES which will take a filename from the input stream, open the file, and pass each line to DOLINE.

$DOLINES ($filename --)

Same as DOLINES but takes name on stack as string.

DL-LINENUM (-- addr , variable containing current line number)

DL.CLOSE.FILE (-- , close the doline file)

You should call this from the word that you set DOLINE.ERROR to.

DOLINE ($LINE -- , do something!?!)

This is a deferred word that the user can set to anything they want as long as it has the same stack diagram.

DOLINES (<filename> -- , open and process file)

DOLINE.ERROR (--)

Deferred word that is called if an error is encountered while processing the file. See DL.CLOSE.FILE.