# Strings and I/O in FORTH

## Contents



This chapter describes some input and output (**I/O**) functions and string handling facilities needed for scientific computing. We will illustrate with words that capture screen output to a disk file, and with a word that translates a FORTRAN expression –entered from keyboard or read in from a file– to FORTH code.

## §1 Standard I/O in FORTH

One area where more conventional languages excel FORTH in convenience for the casual technical user/programmer is standardized I/O. FORTRAN offers all the needed functionality –number formatting, strings, *etc.*– albeit in a somewhat rigid form.

BASIC, a more modern language, offers even more convenient built-in I/O facilities than FORTRAN.

I confess I personally do not find the practices of C or Pascal congenial, but most programmers seem able to live with them.

FORTH possesses no standardized I/O words except for writing to the screen and inputting from the keyboard. Thus there are no

required words for redirecting output to printer, disk file or plotter.

FORTH lacks standard I/O functions for several reasons:

- Historically, FORTH incorporated its own (rather primitive) operating system (OS) on many machines. The aim was compactness rather than sophistication, so it eschewed built-in OS or BIOS ("Basic Input/Output System) functions and/or device drivers.
- FORTH has been implemented on so many machines that hardware independence in I/O has seemed an unattainable goal.

FORTH developers have therefore deemed it better to let each programmer develop his own favorite techniques and specialized device drivers. The result is Bedlam.

## §2 Standard I/O in Scientific FORTH

While the freedom to custom tailor I/O is very nice when unusual problems crop up, no one likes having to roll his own subroutines to do routine tasks (95% of all I/O is routine). A scientific FORTH dialect needs to standardize this routine 95%. This book is intended for dialects like HS/FORTH that run under an operating system[1] and are file oriented. Such dialects necessarily contain a reasonable spectrum of words for disk, screen and printer I/O, equivalent to those in HS/FORTH that I will employ in this chapter.

Some FORTHS are screen oriented. That is, they are designed to read and write blocks of 1024 bytes, displayed as 16 lines of 64 characers per line. This was a great idea for primitive machines like the Timex/Sinclair ZX81 or Jupiter Ace, but in my opinion is thoroughly inappropriate for modern work stations with sophisticated displays.

---

1. Such as MS-DOS, OS/2, Pick, Unix, *etc.*

I strongly recommend users of screen oriented FORTHs to update to file oriented systems. Failing that, they should consult any of the numerous introductory books, or articles in journals such as *Dr. Dobb's Journal*, for suggestions on using 1K blocks to store and input numbers and text.

## §§1 Disk file input

We now tackle the problem of inputting data from disk files. HS/FORTH provides the words **OPEN-INPUT**, **CLOSE-INPUT**, **< FILE**, **$"**, **G#**, **G$**, **NR**, and **$- > F** that will suffice for our needs[2]. Their effects are:

```
OPEN-INPUT    \ open a file for input, whose name is a
              \ string whose address is on the stack.
CLOSE-INPUT   \ close an open input file
< FILE        \ get input from the open file
$"            \ put a string at PAD (terminate with ")
G#            \ get a 16-bit number from open device
G$            \ get a counted string from open  device
              \ (will not read past a CR-LF)
NR            \ skip CR-LF to next record
$- > F        \ convert the string at PAD to a floating
              \ point number on the 87stack.
```

Note: Most operating systems handle sequential files by maintaining a pointer into the file. After a datum is read, the pointer is advanced. Suppose the file *dat.fil* has 3 numbers in it: 17, 32 and -8. Then the phrase

```
$" dat.fil" < FILE  G#
```

will place 17 on the stack. A repetition will place the number 32 on the stack, and a third repetition will place -8 on the stack.

---

2. Voltaire once remarked that if Satan did not exist it would be necessary for God to create him. If your FORTH system lacks words with these functions, by all means, invent them. A MS-DOS programmer's manual (see below) is vital for this task.

We shall use the following data file convention for arrays:

- The first number is the type of the array.
- The second is the order N.
- The third is the number of entries– $N^2$ for a mat (2ARRAY), N for a vector (1ARRAY).
- One or more ASCII blanks (20h, 32d) separate the numbers the file.
- The entries in a 2ARRAY are stored row-wise.

These conventions let one word serve for **1ARRAY**s and **2ARRAY**

Coding from the top down, we plan to enter the following phras to load the file:

```
IN SUPERSEG  100 LONG REAL 1ARRAY  A{
A{  $" data.fil"  FILL<FILE
```

To minimize our labor let us plan to have the word **FILL<FILE** close *data.fil* after reading it in. For safety we also should chec whether the type and order of the array we are filling, and that the array stored in the file are commensurate.

To check whether the file is too large or of the wrong type, we have

```
0 VAR  FILE.TYPE
0 VAR  FILE.LEN
: READ.FILE.STATS  <FILE
    G# IS FILE.TYPE
    G# IS FILE.LEN  ;

: MAT>=FILE?                          ( a{ – a{ )
    DUP D.TYPE    OVER  D.LEN         ( -- a{ T L )
    FILE.LEN  >                       ( -- a{ T f1 )
    SWAP  FILE.TYPE  =                ( -- a{ f1 f2 )
    AND NOT
    ABORT" File bigger than array"  ;
```

The rest of the definition will look something like this:

```
: COMPLEX?  FILE.TYPE  10 > ;      \ is it complex?
: EOR?   < FILE G$  COUNT 0 = ;
: F#.NR  EOR?
    IF   < FILE NR G$ DROP THEN    \ start new record
    PAD $- > F  DROP  ;            \ convert to floating

: FILL < FILE                      ( A{ $filespec - - )
    OPEN-INPUT                     \ open file for input
    MAT > = FILE?                  \ check type & length
    FILE.LEN  0
          DO  DUP  I 0}            ( - - A{ seg off type)
                COMPLEX?
                IF F#.NR  THEN     \ enter 2 #'s
                      F#.NR  GI    \ put away
          LOOP   DROP
    CLOSE-INPUT  ;                 \ clean up
```

## §§2 Disk file output

Suppose one wants to write an array to a file. We can use

```
A{{  $" data.fil" MAT > FILE
```

or

```
B{   $" data.fil" VEC > FILE
```

where we define[4]

```
: MAT > FILE                   ( a{{ $adr - - )
    MAKE-OUTPUT                \ $adr = file.spec
    > FILE  DUP D.TYPE  .
    DUP  D.LEN  DUP  .  **2  .
    .M  CLOSE-OUTPUT ;
```

---

4. We use .M and .V from Chapter 9. The words > FILE, MAKE-OUTPUT and CLOSE-OUTPUT are the output analogues of the input words < FILE, *etc*.

```
: VEC>FILE                    ( V{ $filespec - - )
   MAKE-OUTPUT
   >FILE D.TYPE .
   D.LEN DUP . .
   .V CLOSE-OUTPUT ;
```

## Logging screen activity

Most FORTHs handle output to a device by vectoring through a user variable called **EMIT**. This allows relatively easy output redirection. Hence, for example, everything sent to the screen can be echoed to the printer (in an MS-DOS system operating on a PC-clone this is trivial: just press ctrl-PrtSc to toggle printer logging on or off).

The terminal sessions reproduced in preceding chapters were captured by logging development sessions to disk files. This is far safer than logging to a memory segment or ram-disk because it produces a record that remains even if one's experiments crash the system. Logging to the printer is also permanent, but the results are not easily inserted into a book!

HS/FORTH permits (at least!) two simple methods for redirecting output, which we now describe.

### §§1 Redirection by re-vectoring

The first word in this set of definitions creates the logging file.

```
: MAKE.LOG.FILE
   CR    ." Insert a formatted disk in drive A:"
   CR    ." Press any key to continue, Esc to abort ..."
   KEY 27 =   ABORT" ABORTED!"
   $" A:LOG.TXT"       MAKE-OUTPUT ;

: REOPEN.LOG.FILE
   $" A:LOG.TXT" OPEN-OUTPUT ;
```

```
: FWEMIT        \ a word to replace standard EMIT
   DUP          \ duplicate character
   FEMIT        \ emit to open file
   OUT  1-!     \ decrement the output count
   WEMIT ;      \ emit to the CRT (this increments OUT)

: FCRT  CFA'  FWEMIT  IS EMIT ;  \ vector new routine

: LOG-ON    FCRT      \ all I/O to file & CRT
   IO-STAY  ;         \ normally FCRT would stop
                      \ after word executes.
                      \ IO-STAY leaves FCRT on.

: LOG-OFF    CLOSE-OUTPUT     IO-RESET  ;
                      \ IO-RESET restores I/O
                      \ to standard device
```

### §§2 Redirection using MS-DOS resources

Assembly language programmers familiar with MS-DOS[5] versions 2.x and higher will be aware that the operating system includes many useful functions. Those employed here are invoked by means of an "interrupt" (meaning the CPU halts what it was doing and performs the system function).

The interrupt code to invoke functions is 21h (hex). The system functions are usually specified by placing the function number in register AH, and other needed information is placed in other registers. Then an INT 21 instruction is issued.

For example, to create a new file (or to overwrite an old one of the same name), a program operating under MS-DOS would include the following code:

```
MOV  DX, FILESPEC   ; FILESPEC is address of
                    ; a string ending with ASCII 0
                    ; containing [d:][\path\]file.ext
MOV  AH, 3CH        ; the function number of "create"
MOV  CX, 0H         ; the file attribute
INT  21H            ; now perform the interrupt
```

5. © Microsoft Corporation, 1983-.

Consulting the MS-DOS Programmer's Reference[6] reveals th Function 3Ch either leaves an error code in AX (if there was a error – depicted by having the carry flag bit CF = 1); or leave the file handle number (if all went well – depicted by CF = 0) in A

The HS/FORTH word **MKFILE** expects the address of a counte string on the stack. This string contains the drive, path an filename of the new file. Then **MKFILE** performs the interrup (using code akin to that above) to invoke the function, issues a error message if something went wrong, and leaves the **file handl number** on the stack if all went well.

This **file handle number** is MS-DOS's way to refer to the new fil internally. All subsequent references – say to read or write – are made to this number rather than to the FILESPEC (which would be much more tedious). The word **MAKE-OUTPUT** used above contains **MKFILE** as a component.

Now what does all this have to do with the price of bananas in the Maldive Islands? One of the features of MS-DOS is to treat *everything* like a file, including the printer, CRT, serial ports, *etc.* Thus output intended for the printer can be sent to a file by fooling DOS into thinking the file was the printer. This can be done by placing the file's file handle number where the printer's should be.

But MS-DOS actually has functions we can use to **splice** two file handle numbers so that they are subsequently treated as a single file. These are Function 45h (duplicate a file handle specified in BX: take an already open file and return a new handle that refers to the same file at the same position) and Function 46h (force a duplicate of a file handle specified in BX: take an already open file and force another handle in CX to refer to the same file at the same position; if the file handle specified in CX was open, close it).

Function 45h is used in the HS/FORTH word **DUPH**, and 46h is used in **SPLICEH**. We can combine them with the (already-

6. Radio Shack Cat. No. 26-5403.

defined) printer-logging word **PCRT**, as well as with **IO-STAY** and **CRT**, to let DOS perform the redirection of output:

```
0 VAR LOG-HANDLE
0 VAR P-H#              \ VARs for storing handles
: LOG-ON  $" A:LOG.TXT" MKFILE
                        \ make file, leave handle#
    IS LOG-HANDLE       \ store file handle#
    4 DUPH  IS P-H#     \ make new printer handle#
                        \ and store it
    LOG-HANDLE
    4  SPLICEH         \ splice the handle#'s
PCRT IO-STAY ;          \ redirect output to
                        \ printer & CRT & leave on

: LOG-OFF  P-H#  DUP
    4 SPLICEH           \ splice printer back to printer
    LOG-HANDLE  CLOSEH
                        \ flush to log-file, close
    CLOSEH              \ close new printer handle
    CRT ;               \ go back to CRT only
BEHEAD" LOG-HANDLE  P-H#        \ make VARs local
```

## §§3 Redirection while DEBUGging

The only problem with the logging functions defined above is that they depend on an idiosyncracy of MS-DOS called the *environment*. That is, when we execute a program from within another using the HS/FORTH word **DOS"** we automatically create a new **shell** and run all DOS commands under a second copy of the command processor, COMMAND.COM[7]. This means the environment variables —specifically the handles of open files— will not be valid. Hence the output sent to the screen by a program being run *via* **DOS"** will not be echoed to the log file.

---

7. **DOS"** must be defined in machine language to invoke the appropriate DOS function through another INT 21H call

To capture screen output while DEBUGging, I resort to a utility published in PC Magazine called PRN2FILE[8]. This utility replaces the DOS printer-interrupt routine (INT 17h) by a modification that sends the output to a file instead of the printer. The assembly language listing can be downloaded from PCMag-Net, and is included on the program disk as a convenience to the reader.

To use this utility from within HS/FORTH, first say **DOS"** <cr> to get into the DOS shell. A DOS prompt such as C> should appear. Now give the DOS commands

```
C>prn2file log.fil [/b8]
C>ctrl-PrtSc          ( toggle echoing to printer)
C>debug               ( get into DEBUG)
..... debugging session .....
-q                    ( quit DEBUG)
C>ctrl-PrtSc          ( stop echoing to printer)
C>prn2file            ( with no argument, closes log.fil)
                      ( and redirects back to printer)
C>type log.fil        ( check that it has worked)
C>exit                ( get back to HS/FORTH)
```

Although this is somewhat cumbersome, **it does what I need**, and I did not have to write my own.

With the illustration of how to use HS/FORTH's MS-DOS extensions to fill an array from a file, to send the contents of an array to a file of standard structure, and to capture programming sessions to a file, we have reached the end of the I/O trail.

---

8. PRN2FILE 1.0 © 1987 Ziff Communications Co. Written by Tom Kihlken.