PDP-11 FORTH USER'S GUIDE

John S. James
September 1979

PDP-11 FORTH USER'S GUIDE

This system implements the FORTH language model of the Forth Interest Group, PO Box 1105, San Carlos, Ca. 94070. Nearly identical systems also exist for 8080, 6502, 6800, 9900, and PACE; listings are available from the Forth Interest Group. All of these F.I.G. systems include full-length names to 31 characters, vocabularies, and extensive compile-time checks. They are aligned with the FORTH-78 International Standard.

This PDP-11 version of the F.I.G. common model was implemented and extended by John S. James. It currently runs under RT-11 or RSX-11M, and could be adapted to other operating systems or to run stand-alone.

Questions, requests for diskettes, etc. should be directed to John S. James, P.O. Box 348, Berkeley, Ca. 94701.

PDP-11 Forth User's Guide

CONTENTS

I. Introduction

This User's Guide, together with a language manual, has most of the information you will need for using the Forth Interest Group (F.I.G.) language model implemented on the PDP-11/LSI-11.

For a language manual, we especially recommend either A Forth Primer, by W. Richard Stevens of Kitt Peak National Observatory, and/or Using Forth, by Forth, Inc. Both are available through the Forth Interest Group. Any reasonably standard FORTH language manual will do, however. The "Forth Handy Reference" card from F.I.G. is also recommended as a convenience.

Advanced users and anyone modifying the system should have a copy of the Installation Manual, also from F.I.G. This manual includes a listing of the FORTH system written in FORTH; it corresponds to this system's FORTH.MAC file, which is written in Macro-11. For convenience, the glossary in the Installation Manual is reprinted here.

The programs associated with this user's guide (files FORTH.MAC and FORTH.DAT) are in the public domain and the author encourages modification and distribution, commercially or otherwise. Credit should be given to the Forth Interest Group, P.O. Box 1105, San Carlos, Ca. 94070.

II. Getting Started

This system is distributed on a standard diskette which boots and runs stand-alone, or can be assembled and run under RT-11 or RSX-11M. (To modify for other environments, only read and write a character, detect a character (optional), and read and write a disk block, would have to be changed.)

Either boot the diskette, or assemble, link and run the file FORTH.MAC (for details, see Chapter IX, "Bring-Up Options", below). The system should type

```
FIG-FORTH  V 1.3
```

when it comes up. Now FORTH is ready to run. The file FORTH.MAC contains the complete FORTH language.

If the FORTH screens file is available (on the 'DK:' volume in RT-11, on your default device and account number in RSX-11M, or on the system disk if stand-alone), a recommended way to begin is to type

```
1 LOAD
```

(followed by a carriage return). This system uses disk "screen" #1 as a "load screen", which is like a log-on file in other systems. Here, screen 1 loads an editor, an assembler, and a package of string routines; these are written in FORTH source code in FORTH.DAT in the RT-11 directory on the diskette. Screen 1 also changes an error warning mode so that error message texts (instead of error numbers) are

reported. (The error messages are stored on disk to save memory, and the system is designed to be able to run without a disk if necessary.)

Compiling the editor, assembler, and string package takes about one minute on an LSI-11, less on larger machines. The warning messages that certain words are not unique can be ignored.

III. Sample Session

This is an actual terminal session, with commentary interspersed. Some of the examples were chosen to illustrate special features of this system - the assembler, string package and editor have not been standardized in the FORTH community, though we are guided by common usage even when no formal standard exists.

The assembler, etc. are documented more fully later in this User's Guide; this section is only for illustration. Beginners in FORTH may want to start studying simpler examples, e.g. from a language manual.

In this session, output from the computer is underlined, or (if long sections) marked by a vertical line on the left.

```
 RUN FORTH
FIG-FORTH  V 1.3
88 88 * .
7744  OK
```

'RUN FORTH' is a command to the operating system; FORTH comes up with the message 'FIG-FORTH V 1.3'. At this point, the complete FORTH language is ready, though the editor, assembler, and string package have not been loaded. (Many other FORTH systems compile most of the language on start-up; our approach permits operation without disk if necessary, saves time when bringing up the system, and allows the entire language to be published in Macro-11, which is more

accessible to programmers not familiar with FORTH.)

Now let's test the disk:

```
1 LIST
SCR # 1
  Ø ( LOAD SCREEN)
  1 DECIMAL
  2 1 WARNING ! ( GET ERR MSGS, NOT #S)
  3
  4 CR ." LOADING EDITOR... "  6 LOAD 7 LOAD 8 LOAD 9 LOAD
  5 CR ." LOADING ASSEMBLER... "  1Ø LOAD 11 LOAD 12 LOAD 13 LOAD
  6           14 LOAD 15 LOAD
  7 CR ." LOADING STRING PACKAGE... "  19 LOAD 2Ø LOAD 21 LOAD
  8           22 LOAD
  9 CR
 1Ø : BYE FLUSH CR ." LEAVING FORTH.  HAVE A GOOD DAY." CR BYE ;
 11 CR
 12
 13
 14
 15
 OK
```

We have listed one FORTH screen; it happens to be a "load screen" used to load other screens. Later, when screen 1 is loaded, it will compile the editor, assembler, and string package, which are written in FORTH source code on other screens on the disk file FORTH.DAT. The load screen will also set the error-message warning mode as explained above, and redefine 'BYE' (which exits FORTH and returns to the operating system) to make sure that all disk buffers are flushed before exiting. (Incidently most FORTH systems run stand-alone, in complete control of the computer except for a ROM monitor, so they don't have a 'BYE' instruction.

This system is distributed on a disk which boots and runs stand-alone, but which also has an RT-11 directory, allowing RT-11 or RSX-11M operation if desired.

Let's try a useful definition:

```
: DUMP OVER + SWAP DO I @ U. 2 +LOOP ;
 OK
1000 20 DUMP
10374 18758 17486 41001 932 1012 5440 5441 4454 4390  OK
' DUMP 20 DUMP
1694 1608 1730 904 924 1834 6442 2052 852 65526  OK
```

'DUMP' takes a memory address and a number of bytes, and dumps those bytes (as words) in whatever number base in currently in use (here decimal). (Incidently, the 'U.' operation in 'DUMP' is an unsigned print.) The first example dumps 20 bytes (as 10 words) starting at memory address 1000; the second dumps 10 words starting at the object-code address of the definition of 'DUMP' itself. Here we are in decimal arithmetic, as the system comes up that way, but 'DUMP' would also work in octal, binary, or other number bases.

Now let's use the load screen to compile the assembler, etc:

```
1 LOAD

LOADING EDITOR... R ISN'T UNIQUE I ISN'T UNIQUE
LOADING ASSEMBLER... R0 ISN'T UNIQUE # ISN'T UNIQUE
LOADING STRING PACKAGE...
BYE ISN'T UNIQUE
 OK
```

The message that various operation names are not unique (i.e.

are being redefined) is a warning which we can ignore.

Now let's use the assembler to benchmark speed of execution of an empty loop in higher-level FORTH vs. machine code:

```
: TIME1 30000 0 DO LOOP ;
 OK
CODE TIME2 72460 # R0 MOV, BEGIN, R0 DEC, EQ UNTIL. NEXT, C;
 OK
: T1 30 0 DO TIME1 LOOP ;
 OK
: T2 30 0 DO TIME2 LOOP ;
 OK
T1      19 sec.
 OK
T2       3 sec.
 OK
```

'TIME1' does 30,000 empty loops in FORTH; 'TIME2' does the same number (72460 octal) of machine-code loops using two instructions (DEC and BNE; this benchmark is not intended to be definitive). 'T1' and 'T2' embed these tests in a 30-times loop, in order to allow the code test to be timed with a watch. 'T1', 900,000 empty loops in FORTH, takes about 19 seconds on a PDP-11/45; the code loops take 3 seconds, about six times faster. Incidently a comparable BASIC program

```
IAS/RSX BASIC V02-01

READY
10 FOR I%=1 TO 30
20 FOR J%=1 TO 30000
30 NEXT J%
40 NEXT I%
50 END
```

takes 695 seconds, 35 times slower than the higher-level FORTH. (FORTH is slower on number-crunching, however; one test ran only nine times faster than BASIC.)

The 'DUMP' operation defined above lets us look at the object code of the higher-level and assembly routines, respectively:

```
OCTAL
 OK
' TIME1 20 DUMP
1334 72460 3764 1610 1456 177776 2722 52205  OK
' TIME2 20 DUMP
12705 72460 5305 1376 12402 132 52202 120061  OK
DECIMAL
 OK
```

(The actual numbers in the dump of 'TIME1' will vary depending on your operating system and linkage editor if used, and upon bring-up options.) In this example, the machine code happens to take fewer words of memory than the FORTH. But usually FORTH object code is considerably more compact than machine code, especially for large programs.

When speed is important, the assembler allows us to rewrite critical routines in code; then they have all the convenience of FORTH, and the fact that they are code is transparent, so other operations which use those routines do not need to be changed. The assembler also allows calls to operating system services, or to subroutine packages written in FORTRAN or other languages, making these features interactively accessible from the keyboard, just as if they were native FORTH. As we have seen, the FORTH assembler allows structured

loops; these and structured conditionals can be nested (although branches to labels or numerical addresses may be used if desired). User-defined macros are available. And the code is ready to execute immediately after being typed in or loaded from disk, with no wait for assembly or linkage editing.

Let's look at part of the string package:

```
" THIS IS A STRING"
 OK
$.
THIS IS A STRING OK
$.
$STACK EMPTY
: XX " TESTING STRINGS" ;
 OK
XX
 OK
$.
TESTING STRINGS OK
$.
$STACK EMPTY
```

The '"' operation (quotation mark) introduces a string; it must be followed by one space. The string goes until the next quote, which is the delimiter, or until the end of the line. If used outside a colon definition, '"' immediately puts the string onto the string stack, a special stack which holds the actual characters of the string (with length information). If used inside a colon definition, '"' stores the string within the definition in the dictionary; later, when that part of the definition is executed, the string

is pushed onto the string stack. '$.' prints the string on top of the string stack (most accessible string). String length can be up to 32,767 characters, depending on memory availability. The string stack is always checked for underflow ('$STACK EMPTY') or overflow whenever these are possible.

The following illustrates the '$+', or string concatenation, and also the '$LEN', which gets the length of the string on top of the string stack.

```
XX
 OK
$LEN .
15  OK
" ANOTHER TEST STRING" $+
 OK
$LEN .
34  OK
$.
TESTING STRINGSANOTHER TEST STRING OK
```

The following operation ('LONG', below) is defined to double a string (concatenate it with a copy of itself) 7 times, i.e. it makes the string 128 times as long as it was.

```
: LONG 7 Ø DO $DUP $+ LOOP ;
 OK
" ABCD"
 OK
$LEN .
 4  OK
LONG
 OK
$LEN .
 512  OK
$.
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCD OK
```

To see some application examples which are on the disk, let's first make an index of the disk. The 'INDEX' operation takes two arguments (beginning and ending screens), and prints the first line of each screen in that range. By convention, the first line of each screen is a comment telling what that screen has on it. Blank lines should indicate empty screens available for use. The FORTH.DAT file as distributed on the diskette has 70 screens; this system could handle at least 32,767 if disk space were available.

```
1 39 INDEX

 1 ( LOAD SCREEN)
 2
 3
 4 (  ERROR, WARNING, AND OTHER MESSAGES - SCREENS 4 AND 5 )
 5 (  ERROR MESSAGES, CONTINUED  )
 6 ( EDITOR - SET-UP)
 7 ( EDITOR - OPERATIONS)
 8 ( EDITOR, SCREEN 3)
 9 ( EDITOR, SCREEN 4)
10 ( ASSEMBLER)                                  OCTAL
11 ( ASSEMBLER, CONT.)                           OCTAL
12 ( ASSEMBLER - INSTRUCTION TABLE)              OCTAL
13 ( ASSEMBLER - CONT.)                          OCTAL
14 ( ASSEMBLER - REGISTERS, MODES, AND CONDITIONS)   OCTAL
15 ( ASSEMBLER - STRUCTURED CONDITIONALS)  OCTAL
16
17 ( ASSEMBLER - EXAMPLES)
18
19 ( STRING ROUTINES)          DECIMAL
20 ( STRINGS - CONTINUED)
21 ( STRINGS - CONTINUED)
22 ( STRINGS - CONTINUED )
23
24 ( TRIG LOOKUP ROUTINES - WITH SINE *10000 TABLE)
25
26 ( FORTRAN LINKAGE, RSX)
27
28 ( RT-11 SYSTEM-CALL EXAMPLE - DATE)
29 ( RSX-11M SYSTEM-CALL EXAMPLE - DATE)
30 ( RSX-11M SYSTEM-CALL EXAMPLE - TERMINAL I/O)
31
32 ( EXAMPLES - RANDOM #S, VIRTUAL ARRAY, RECURSIVE CALL)
33
34 ( CREATE BOOTABLE IMAGE ON SCREENS 40-47.  FOR STAND-ALONE.)
35 ( CREATE A BINARY IMAGE ON SCREENS 40 - 47  )
36 ( CREATE BOOT LOADER.  NOTE - DOES NOT WRITE BOOT BLOCK)
37 ( CREATE BOOT LOADER, CONT.)       OCTAL
38 ( DISK COPY FROM SYSTEM DISK TO DX1)
39 ( ** CAUTION **   BINARY IMAGE IN SCREENS 40-47) OK
```

Screen 29 uses the RSX-11M system call for the date and time. Here is is loaded, and also listed so we can see it. The 'TIME' operation sets up the re-entrant form of the system call ('GTIM$S'), although of course the RSX-11M macro is never used here. (This screen as a whole is not re-entrant since it includes the 8-word buffer 'TBUFF' to receive the information, but this buffer could be located anywhere else, if re-entrancy were needed.)

```
 29 LOAD
 OK
 29 LIST

SCR # 29
  0 ( RSX-11M SYSTEM-CALL EXAMPLE - DATE)
  1 DECIMAL
  2 0 VARIABLE TBUFF 14 ALLOT
  3 CODE TIME TBUFF # SP -) MOV,  2 400 * 75 +  # SP -) MOV,
  4    377 EMT, NEXT, C;
  5 : YEAR ( -> N )  TIME TBUFF @ ;
  6 : MONTH ( -> N )  TIME TBUFF 2+ @ ;
  7 : DAY ( -> N )  TIME TBUFF 4 + @  ;
  8 : HOUR ( -> N )  TIME TBUFF 6 + @  ;
  9 : MINUTE ( -> N )  TIME TBUFF 8 + @  ;
 10 : SECOND ( -> N )  TIME TBUFF 10 + @  ;
 11 : TICK ( -> N )  TIME TBUFF 12 + @  ;
 12 : TICKS/SECOND ( -> N )  TIME TBUFF 14 + @  ;
 13
 14
 15
 OK
```

Screen 28 defines comparable 'YEAR', 'MONTH', and 'DAY' operations for RT-11.

Here are some uses of the date and time operations. The loop in 'X' is indicating the time (in cumulative clock ticks) required to print the characters.

```
YEAR .
80  OK
MONTH .
1  OK
DAY .
21  OK
HOUR .
20  OK
MINUTE .
17  OK
SECOND .
44  OK
: X 20 0 DO TICK . LOOP ;
 OK
X
0 8 16 28 40 52 4 12 24 36 48 0 8 16 28 40 52 4 12 24  OK
TICKS/SECOND .
60  OK
```

(Incidently 'TICKS/SECOND' is one name; tokens are separated by blanks. Names in this system can be up to 31 characters long, and the full name is remembered, in contrast to some FORTH systems which only remember the length of the name and the first several characters.)

Screen 24 contains a table-lookup 4-digit sine and cosine routine; it is fast, accurate enough for most graphics applications, and it doesn't require floating point. 'SIN' and 'COS' take an integer number of degrees (-32K to 32K), and return the sine or cosine value multiplied by 10,000; linear interpolation would allow fractional degrees with little loss of accuracy.

```
24 LOAD
 OK
Ø SIN .
Ø  OK
90 SIN .
10000  OK
10000 SIN .
-9848  OK
7 SIN .
1219  OK
345 SIN .
-2588  OK
15000 SIN .
-8660  OK
15000 COS .
-5000  OK
-29000 COS .
-9397  OK
```

The '*/' operation (multiply, then divide, keeping a double-precision, 32-bit intermediate product) allows scaling by sine or cosine, e.g.

```
30000 45 SIN 10000 */ .
21213  OK
```

This scaling would be useful in figure rotation.

The whole sine/cosine lookup routine and table is on one screen. Its detailed operation will be described in an advanced section, but let's take a look at it now:

```
24 LIST

SCR # 24
  0 ( TRIG LOOKUP ROUTINES - WITH SINE *10000 TABLE)
  1 : TABLE <BUILDS 0 DO , LOOP DOES> SWAP 2 * + @ ;
  2 10000 9998 9994 9986 9976 9962 9945 9925 9903 9877
  3  9848 9816 9781 9744 9703 9659 9613 9563 9511 9455
  4  9397 9336 9272 9205 9135 9063 8988 8910 8829 8746
  5  8660 8572 8480 8387 8290 8192 8090 7986 7880 7771
  6  7660 7547 7431 7314 7193 7071 6947 6820 6691 6561
  7  6428 6293 6157 6018 5878 5736 5592 5446 5299 5150
  8  5000 4848 4695 4540 4384 4226 4067 3907 3746 3584
  9  3420 3256 3090 2924 2756 2588 2419 2250 2079 1908
 10  1736 1564 1392 1219 1045 0872 0698 0523 0349 0175
 11  0000         91 TABLE SINTABLE
 12 : S180 DUP 90 > IF 180 SWAP - ENDIF SINTABLE ;
 13 : SIN ( N -> SIN)  360 MOD DUP 0< IF 360 + ENDIF DUP 180 >
 14    IF 180 - S180 MINUS ELSE S180 ENDIF ;
 15 : COS ( N -> COS) 90 + SIN ;
 OK
```

Line 1 creates a new data type, 'TABLE', using the pair of operations '⟨BUILDS' and 'DOES⟩' which create new data types. Line 11 executes 'TABLE' to create a table called 'SINTABLE'; of course 'TABLE' can also be used elsewhere, creating other tables of this type (but of any length desired, not always 91). Similarly other data types could create arrays with various (or variable) numbers of dimensions and with run-time bounds checks if desired, various user-defined record structures, etc. Here, 'TABLE' is defined in higher-level FORTH, making it machine-independent; but if optimum execution speed were desired, it could be defined in machine code, using the operation ';CODE' (not to be confused with 'CODE' - see the Assembler section).

Incidently the operation 'SINTABLE' behaves exactly like 'SIN', except that its arguments must be 0-90 degrees. 'S180' reflects to extend this to 0-180, 'SIN' extends it to all values, and 'COS' is sine differing 90 degrees in phase.

```
BYE
LEAVING FORTH.  HAVE A GOOD DAY.
```

IV. Editor

Philosophy

There is little standardization among Forth editors. This system is a unified "line" and "string" editor, avoiding redundant commands. It de-emphasizes the 64-character line, because some machines (e.g. Apple) can't use it, and because Forth may abandon the screen concept at some future time. Most of this editor's commands allow default arguments. They work well together in practice, giving an exceptionally convenient teletype-style editor (which runs on almost all terminals, without customization). New users can learn it comfortably within an hour. This editor requires a Forth system which keeps screens in a contiguous 1K buffer (or the editor would have to be modified to move them to such a buffer).

Entering and Exiting the Editor

The editor must be loaded (e.g. by typing '1 LOAD'). Normally the editor stays in memory for the rest of the session.

To edit screen 'n', type

```
n EDIT
```

and after the editing is complete, exit with

```
EX
```

which updates the disk. Or exit with

```
SCRATCH
```

to throw away the result of the session, leaving the screen

on disk unchanged.

To get back later and edit the same screen, type

```
E
```

with no argument. This way you don't have to remember the screen number. The cursor position (see below) is also remembered.

Unlike many other Forth systems, the command

```
n LIST
```

does not affect the editor. While editing one screen, you can list others to look at them.

To list the screen currently being edited, type

```
L
```

with no argument. After listing the screen, it prints the line currently holding the cursor, with the cursor position shown as an underscore character. The cursor starts at the beginning of line zero, when the editing session begins (with 'n EDIT').

Moving the Cursor

To move the cursor to the beginning of any line (0-15), type

```
n T
```

which types that line after the cursor has been moved. To type the current line without moving the cursor, just type

```
T
```

The editor keeps track of the stack depth to know whether or not an argument has been given.

To move the cursor 'n' character positions, type

n M

where 'n' may be positive or negative. Without the argument, 'M' defaults to '1 M'. The editor prevents the cursor from being moved off the screen.

Insert, Delete, and Replace

To insert at the current cursor position, type

I

followed by a space, followed by the string to insert, followed by a carriage return. The first space after the 'I' is a delimiter and is not part of the string. When the string is inserted, the rest of the line will be pushed over, with characters lost on the right. If the insert string goes beyond the end of the line, it will replace characters on the next line. 'n I' (with an argument) moves the cursor to the beginning of line 'n', then inserts there.

To delete 'n' characters, type

n D

where 'n' may be positive or negative. The rest of the line is moved over with blank fill characters added on the right. 'D' defaults to '1 D'. To prevent serious errors, the command will only delete characters within one line.

To replace, starting with the current cursor position, type

R

followed by a blank, and then by the string to replace. 'n R' moves the cursor to the beginning of line 'n' first. A maximum

of 64 characters can be replaced in one command; these 64 may overflow from one line to the next. Any characters not replaced on the line are unchanged.

String Search and Replace

To search for a given string, type

```
S
```

followed by a space, the string, and carriage return. The search goes from the current cursor position to the end of the screen. If the string is found, the cursor is moved to just before that string. If the string is not found, the cursor is not moved. In either case, the line now containing the cursor is typed.

The string search argument is saved. To search for the same string again, type 'S' followed by a carriage return.

'n S' moves the cursor to line 'n' first. E.g. '0 S' (zero argument) searches the entire screen.

After a search string has been found, a convenient way to replace it is to type

```
-R
```

followed by a space, the replacement string, and a carriage return. '-R' is particularly useful when the search and replacement strings have different lengths. The current editor version does not save the replacement string.

Typing Multiple Lines

To enter more than one line without having to give a command for each line, type

```
n NEW
```

followed immediately by a carriage return. Then type the lines. To terminate the 'NEW' entry mode, type two carriage returns in a row. Without the argument, 'NEW' will start at the current cursor line.

'NEW' differs from the single-line replace, 'R', in several ways. 'R' requires the replacement to be on the same line as the 'R'; 'NEW' requires a carriage return, so the first replacement line must be on a line by itself (anything after the 'NEW' on the same line will be lost). Also, 'NEW' will blank out the remainder of any line it affects, if less than 64 characters are entered; 'R' preserves any part of the line which is not replaced.

'NEW' is the normal way to enter new programs. Incidently if a blank line is desired within the range of the 'NEW', type <u>two</u> or more blanks on that line (as the first blank is the delimiter of the 'NEW').

Moving Lines Around

```
n m TRADE
```

will swap lines 'n' and 'm'. Since no characters are lost in the trade, it is possible to recover from errors.

```
n SPREAD
```

will push down all lines from 'n' through 14, and clear line n; the last line (15) is lost. The argument 'n' is optional; if omitted, the line currently holding the cursor will be assumed. 'SPREAD' is used to insert lines into the middle of a screen.

Moving Screens

To copy one Forth screen into another (destroying any information which was in the destination screen) type

```
n m SCREENMOVE
```

which copies from screen 'n' to 'm'. 'SCREENMOVE' is in the FORTH vocabulary, not the EDITOR vocabulary, so it can be used any time, not only while editing. It is loaded into memory with the editor, however, so it cannot be used until the editor has been loaded.

Multiple Commands on One Line

Several commands can be on one line, except that any command which takes the rest of the line as a string argument (eg 'I', 'R') must of course occur last on the line. Sometimes an entire editing session can be on one line, including entry and exit from the editor, e.g.

```
31 EDIT 8 T 40 D L EX
```

This line edits screen 31, deletes the first 40 characters of line 8, lists the screen, and exits. With proper care concerning vocabularies (see below), such a line could be made into a command, used in loops, etc.

Extending the Editor

Occasionally you may want to add your own operations to the editor, e.g. to define editor macros. If you change the editor by changing its source screens, make sure you are backed up first. In case of error, it may not be possible to load the editor again in order to correct the problem.

A safer way to extend the editor is to leave the original alone, but add your own operations (either typed in or loaded from disk screens). To do this the editor must first be loaded, but normally you would compile editor operations while not editing a particular screen. Begin with 'EDITOR DEFINITIONS', and end with 'FORTH DEFINITIONS'. This way the new definitions will be put into the EDITOR vocabulary.

Note that EDITOR redefines the word 'I' (for Insert), while the FORTH vocabulary uses 'I' for the index of the 'DO' loop. Within 'EDITOR DEFINITIONS', 'I' will compile as the EDITOR 'I' - resulting in bizarre errors if the 'DO' loop 'I' was intended. Within the editor you can reach the FORTH 'I' by

```
FORTH I EDITOR
```

For example, here is a definition of an operation 'FLIP' which exchanges the top and bottom halves of the screen being edited:

```
EDITOR DEFINITIONS
: FLIP 8 0 DO FORTH I EDITOR DUP 8 + TRADE LOOP ;
FORTH DEFINITIONS
```

******

A command summary appears on the following page. Copy it for a quick reference at the terminal.

This editor takes less than 1500 bytes of memory. Total time to design, code, and debug it was about 6 working days.

Editor Command Summary

```
Entry and Exit
     n EDIT          Begin editing.
     EX              Terminate editing session.
     SCRATCH         Terminate, throw away changes.
     E               Begin new session where last left off.

Enter Data (Multiple Lines)
     n NEW           Accept starting at line n; null line terminates.
                        Without argument - start at current line.

Move Cursor and List
     n T             Move cursor to line n, and type it.
                        Without argument - type current line.
     n M             Move cursor n positions.
                        Without argument - '1 M'.
     L               List the screen being edited.
     S string        Search from current cursor position.
                        Without the string - use same one as last 'S'.

Insert, Delete, Replace
     n I             Insert at beginning of line n, move rest over.
                        Without argument - insert at cursor.
     n D             Delete n characters, moving rest of line.
                        Without argument - '1 D'.
     n R             Replace at beginning of line n.
                        Without argument -  at cursor position.
     -R string       Use after 'S'; replace found string.
                        Without the string - delete found string.

Moving Lines
     n m TRADE       Swap lines n and m.
     n SPREAD        Move all nines n and below down one; blank n.
                        Without argument - assume cursor line.

Moving Screens
     n m SCREENMOVE  Move screen n to m, destroying m.  May be
                     used while not editing any screen.
```

V. Assembler

FORTH assemblers are usually used to code short, critical routines , sequences for device handlers, etc; generally most of the program stays in higher-level FORTH. The code routines have FORTH names and behave exactly like other FORTH operations (taking their arguments from the stack, etc.), except that they run at full machine speed. Often the code routines of a package are also written in higher-level FORTH, so that the package can be transported to different CPUs, and later optimized with native code routines as desired.

Code routines created by this assembler are ready to execute immediately when entered, with no wait for separate assembly and linkage steps. They execute at full machine speed of course. The conventional op codes are provided, though it is customary in FORTH assemblers to end op code names with commas; the commas are just part of the name, not punctuation. This particular assembler is optimized for user convenience; it takes 2.6K bytes of memory, more than most FORTH assemblers. Normally the assembler is loaded by the '1 LOAD' command, and unless memory is tight, it can remain in memory throughout the session.

This as other FORTH assemblers accepts source code in postfix; mode symbols follow operands, and op codes come last. In this assembler, arguments of two-address instructions are in the conventional order, however.

The examples reproduced below just begin to illustrate the capabilities of this assembler:

```
SCR # 17
  Ø ( ASSEMBLER - EXAMPLES)
  1 CODE TEST1 33ØØ6 # 33ØØØ MOV, NEXT, C;
  2 CODE TEST2 555 # 33ØØØ () MOV, NEXT, C;
  3 CODE TESTDUP S () S -) MOV, NEXT, C;
  4 CODE TESTØ RØ S -) MOV, NEXT, C;
  5 CODE TESTBYTE 33ØØ6 R1 MOVB, R1 S -) MOV, NEXT, C;
  6 CODE TEST3 33ØØØ # R1 MOV, 444 # 2Ø R1 I) MOV, NEXT, C;
  7 CODE TEST-DUP S () TST, NE IF, S () S -) MOV, ENDIF, NEXT, C;
  8 CODE TESTLP1 15 # R1 MOV, BEGIN, R1 DEC, GT WHILE, R1 S -) MOV,
  9    REPEAT, NEXT,  C;
 1Ø CODE TESTLP2 15 # R1 MOV, BEGIN, R1 S -) MOV, R1 DEC,
 11    EQ UNTIL, NEXT, C;
 12 : TESTVARIABLE CONSTANT ;CODE W S -) MOV, NEXT, C;
 13
 14
 15
 OK
```

Line 1 creates an operation 'TEST1'; when 'TEST1' is executed, it moves literal 33006 (octal) to octal address 33000. (The address could have been a label or constant, or been computed.) 'NEXT,' is a macro which assembles the two-instruction inner interpreter which resumes Forth execution. (A 'TRAP ERROR' message when a word defined by the assembler is first executed often means that 'NEXT,' was forgotten.)

'CODE' begins the definition; like ':' which begins higher-level definitions, it takes the name of the following word in the input stream, and enters it in the dictionary. 'C;' terminates the definition. Naturally 'CODE' and 'C;' should always be used as a pair. If 'C;' is forgotten, the definition will not be executable - it will be treated as

undefined, not found by a search of the dictionary - although it will still take space in the dictionary, and it will appear in a 'VLIST'.

'CODE' sets the 'CONTEXT' vocabulary to ASSEMBLER, so that the op codes, mode symbols, etc. will be recognized. (They are unknown outside of a code definition.) 'C;' sets the vocabulary back to FORTH. 'CODE' also sets the number base to octal, but it saves the base previously in effect; 'C;' restores that base. 'CODE' also notes the stack depth upon entry; 'C;' tests to see if the depth is the same upon leaving the definition. If not, a message 'ERROR, STACK DEPTH CHANGED' is given. This usually means that a mode symbol or operand was forgotten. (Of course when code routines execute, they are allowed to change the stack depth; the test is at assembly time.)

Unlike colon definitions, which set the system into a special compile state, 'CODE' definitions leave it in regular Forth execution. Therefore the whole FORTH language is available for address arithmetic, macros, labels, etc.

In line #2, 'TEST2' is defined to move literal '555' into indirect address 33000. The symbol '()' was chosen for indirection instead of '@', which is used for Fetch in the FORTH vocabulary.

'TESTDUP' (line 3) is the same as 'DUP' - it moves from the stack pointer indirect to the stack pointer autodecrement. Note the symbol 'S', which refers to the FORTH stack. This assembler also has a symbol 'SP' to refer to the machine

stack; in this implementation, they are different.

'TEST0' pushes R0 onto the stack, showing how FORTH can get access to register contents.

'TESTBYTE' (line 5) moves the byte at address 30006 to the stack. It goes by way of R1, a register available as a temporary. The byte cannot be moved directly into the FORTH stack, as the 'MOVB' instruction would autodecrement by 1 instead of 2, destroying the stack and crashing FORTH.

Registers R0, R1, and R2 are available for CODE routines to use without restoring. R4 is the FORTH instruction pointer and R5 is the FORTH stack; the machine stack is the FORTH return stack. R3 points to FORTH's "user" area.

'TEST3' illustrates an indexed address - literal 444 is moved into 20 indexed by R1 - or address 33020 octal in this case.

'TEST-DUP' introduces a code-level structured conditional. (This operation is the same as '-DUP', which duplicates the top of the stack if it is non-zero.) First, 'S () TST,' tests the top of the stack non-destructively. Then 'NE IF,' sets up a branch instruction which executes the code between 'IF,' and 'ENDIF,' only if the 'NE' condition code is true - i.e. it does branch around if 'NE' is false. 'S () S -) MOV,' works as in the 'TESTDUP' example, line 3 above. Note that 'IF,' is different from the higher-level analogous operation 'IF'. In case you forget the comma, an error message will be given, since 'IF' can only be used inside a colon definition.

The following condition tests may be used with 'IF,'. These same tests may also be used with the 'WHILE,' and 'UNTIL,' looping structures described below.

EQ NE MI PL VS VC CS CC

LT GE LE GT LOS HI LO HIS

Note that there are no commas in the names. The convention is to use commas to end the names of instructions which actually place code in the dictionary ('BEGIN,' - described below - is an exception).

Incidently, the complete set of mode symbols is:

)+	autoincrement
-)	autodecrement
I)	indexed
@)+	autoincrement deferred
@-)	autodecrement deferred
@I)	indexed deferred
#	immediate
@#	absolute
()	either register deferred or relative deferred

'()' tests whether its argument is 0-7, in which case it assumes register deferred; otherwise it is relative deferred.

No mode symbol at all defaults to register mode if the argument is 0-7, relative otherwise. To address memory locations 0-7, the instruction would have to be specially created in octal. (Also, a '-1' (177777) address may have to be specially created, as the assembler uses '-1' as a flag

during the handling of the mode default.)

Lines 8-9, 'TESTLP1', create a 'BEGIN,...WHILE,...REPEAT,' loop in code. First, octal 15 is moved to R1. Within the loop, R1 is decremented, and while the result is greater than zero, the loop continues. 'R1 S -) MOV,' pushes R1 onto the stack; this is done so that we can later check that the loop worked right.

Similarly, 'TESTLP2' creates a 'BEGIN,...UNTIL,' loop in code. These loop structures can be nested.

These structured conditionals use the 'BR' instructions, and give an error message in case of attempt to branch outside the allowable range of those instructions (this shouldn't happen often, because FORTH code definitions are customarily short). The assembler could be modified to substitute 'JMP,' in that case.

Line 12 creates a new data type 'TESTVARIABLE' (it's the same as 'VARIABLE'). The word ';CODE' is the machine-language equivalent of the pair '<BUILDS...DOES>'. 'W', which is another name for 'R1', is a register used as a temporary by the system. When 'TESTVARIABLE' is executed, it will grab the next word in the input stream, and make a dictionary entry for it (this is done by 'CONSTANT'). ';CODE' then changes that dictionary entry so that when the new word gets executed, 'W S -) MOV,' will execute (instead of the usual 'code routine' associated with the data type CONSTANT. Full discussion of ';CODE' is beyond the scope of this User's Guide, but for comparison, see

the definition of 'VARIABLE' in FORTH.MAC.

The previous examples have used numeric literals to represent addresses. Naturally, FORTH constants could be used just as well; the assembler doesn't care how the address gets on the stack. FORTH variables can be used as labels, as illustrated below. The label must be defined before it is used, so these labels can only refer backwards. Usually this isn't a problem because the structured conditionals take care of most of the branching, and FORTH constants, variables, or allocation algorithms provide the addresses of data areas. If a non-structured forward branch were required, it would have to be patched.

The following example uses variable 'XX' as a label. The variable is created outside of a code definition (else the dictionary entry, including the name, would be in the line of code). The 'HERE XX !' places the current dictionary address into 'XX'; it generates no code, but it saves the address where 'TESTSUB', a subroutine, begins. The subroutine moves '333' to the stack (so we can check that it worked), then it does 'PC RTS,' No 'NEXT,' is needed, because this subroutine returns through the RTS instruction instead of continuing with FORTH execution.

'TESTCALL' uses 'XX @' to get the address for the 'JSR,'. A number, FORTH constant, or computed address would have been equivalent.

```
Ø VARIABLE XX
 OK
CODE TESTSUB HERE XX !     333 # S -) MOV, PC RTS, C;
 OK
CODE TESTCALL PC XX @ JSR, NEXT, C;
 OK
TESTCALL .
219  OK
219 OCTAL .
333  OK
DECIMAL
 OK
```

Note - Many FORTH systems only remember the first three characters and the length of operation names; therefore 'TEST1' and 'TEST2' used in the above examples would not be distinguished, so this naming scheme could not be used. This and the other Forth Interest Group systems distinguish the full name, up to 31 characters. Long names take more memory than short ones, but the difference is slight because the name is only remembered once (when defined), regardless of how often it is used.

This system does permit the user to cut off names at three characters or some other maximum length if desired, by changing the value of the system-defined variable 'WIDTH', which is normally 31. This feature is useful if memory is very tight, or to check that programs will run on other FORTH systems which do not remember the full names.

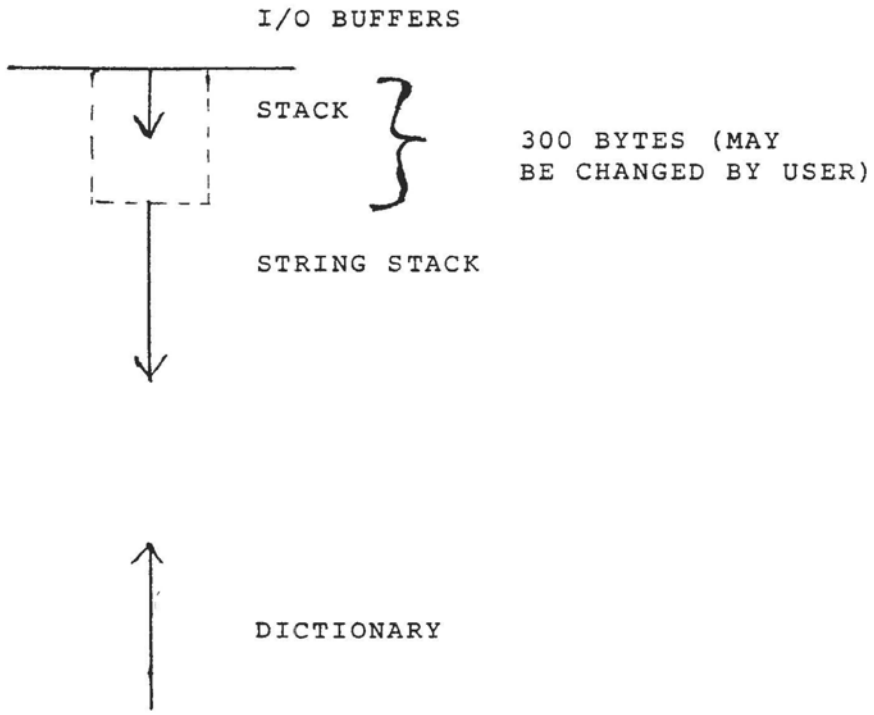
VI. Strings

The "Sample Session" chapter includes examples of use of the string package. This chapter gives an overview and a glossary of the string operations available.

Strings are kept on a separate string stack, or in string variables in memory. In either case, the string is represented as a length word, followed by the string of characters. The length word is on a word boundary, and if the number of characters is odd, an extra byte is left.

The usual Forth convention is that operations should destroy their arguments on the stack. Some of the string operations, however ('$LEN', '$SEG' for example) do not destroy their string-stack arguments; others (e.g. '$.') do.

In this as in most Forth systems, the dictionary grows upward in memory, while the stack (data stack) grows downward. In this system, the string stack is located on a "platform" over the data stack; it also grows down toward the dictionary. The following diagram shows a simplified memory map:



This layout was chosen because in practice very few Forth programs, even the largest, ever use a stack depth of over 50 words (recursive programs are an exception of course). As distributed, this system allows a depth of 150 words (300 bytes) in the data stack, before the string stack begins. The user can increase this size by changing the number '300' in the definition of 'STACKSIZE' (screen 19 line #2); no reassembly of FORTH.MAC is required. And a program could use over 150 words of stack without increasing 'STACKSIZE', as long as it did not use strings at the same time as the stack depth was over 150.

This arrangement allows the string stack to use almost all of available memory if required; and yet when it isn't used, it doesn't take any memory.

These are the string operations of most interest to the user:

"	Introduce a string literal. The quote must be followed by a blank, then by the string, followed by a terminating quote. The string literal must all be on the same line. This operation cannot be used to enter a null string. If executing, the string literal is immediately placed on the string stack. If compiling, it is included in the dictionary definition, and placed on the string stack when that part of the definition is executed. This is analogous to the handling of numeric literals (ordinary numbers) in Forth.
$.	Print the top string on the string stack, and destroy it.
$DUP	Duplicate the top string.
$DROP	Drop string.
$SWAP	Swap strings.
$OVER	Over - copy second string into string stack.
$@	Fetch the string from memory (address on data stack) and put it on the string stack. This string in memory must start with a length word.
$!	Store from the string stack into memory.
$@TEXT	Like '$@', but takes only the string of characters from memory, not the length word. This operation takes two arguments on the data stack, the count (number of characters) on top, and the memory address where the text string begins. Note that these are the same arguments expected by the Forth word 'TYPE'.

$DIM Creates a string variable of the given length; e.g. '80 $DIM X' creates the Forth word 'X' which, when executed, returns the address of an area for an 80-character string. The string variable is not initialized. This implementation does not store the length in the area, so operations which move strings there cannot automatically guard against too long a string being moved in (which would usually crash Forth).

$VARIABLE Does '$DIM' for the string on top of the string stack, and initializes. It drops the string from the stack.

$LEN Returns length of top string - doesn't drop it.

$SECOND Returns address of length word of second string.

$SEG Segment. Takes beginning and ending index of the segment (1-origin) as arguments. Does not destroy its string argument. Returns the segment string to the string stack.

$STR Creates string from the numerical value on the data stack.

$VAL Converts top string to its numerical value. (Note - this implementation accepts positive numbers only, and no leading blanks.) Stops conversion at first non-digit. Drops string.

$< Compares top two strings (and drops them). Returns boolean True if second string is less, False otherwise.

$= Same, for equal.

$> Same, for greater.

$+ Concatenate top two strings.

$NULL Enter a null string on the string stack.

$CLEAR Clear the string stack.

Two enhancements which didn't make this release in time are a generalized '$VAL' (allowing minus signs and/or leading

blanks) and a string search operation. (The latter should probably be written to allow a code option - e.g. written to use an operation which might be called 'CCOMP', like 'CMOVE', only compare instead of move. Then 'CCOMP' could be available in higher-level Forth for transportability, and easily written in code for various computers, for speed.) These enhancements when made will not affect any programs which are already running on this system .

The string package currently takes less than 1200 bytes of object code.

VII. Operating System Calls

A. RT-11

Screen 28 shows an RT-11 system call to get the date. The Forth assembler sets up the same EMT (with its argument in R0) as the '.DATE' macro would in Macro-11. (Naturally, '.DATE' is not used here.)

```
SCR # 28
  Ø ( RT-11 SYSTEM-CALL EXAMPLE - DATE)
  1 CODE DATE 12 400 * # RØ MOV, 374 EMT, RØ S -) MOV, NEXT, C;
  2 : YEAR ( -> N )  DATE 31 AND 72 + ;
  3 : DAY ( -> N )  DATE 32 / 31 AND ;
  4 : MONTH ( -> N)  DATE 1024 / 15 AND ;
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

As described in the '.DATE' documentation in RT-11 Advanced Programmer's Guide, R0 must have octal 12 in the high-order byte, zero in the low-order, when EMT 374 is executed. The Forth assembler code '12 400 * # R0 MOV,' could have been written '6000 # R0 MOV,', but the former is more explanatory, and the extra multiplication is only at assembly time. The call returns the date in R0; the right-most 5 bits are for year (the year minus 72), next 5 bits for day, and next 4 for month. The 'YEAR', 'DAY', and 'MONTH' operations break out these bits, and 'YEAR'

adds 72. If 'YEAR' returns 72, it probably means that the operator has not entered a date.

B. RSX-11M

The RSX call for the date (and time) is described in the Sample Session chapter. Here the address of an 8-word buffer to receive the information is pushed onto the PDP-11 stack ('SP' in Forth assembler is different from the Forth stack 'S'). Then a word containing the code for the call and the size of the block on the stack (2 words) is pushed. Then an EMT 377 is executed. See documentation of '$GTIM$' in RSX-11M Executive Reference Manual.

```
SCR # 29
  0 ( RSX-11M SYSTEM-CALL EXAMPLE - DATE)
  1 DECIMAL
  2 0 VARIABLE TBUFF 14 ALLOT
  3 CODE TIME TBUFF # SP -) MOV,  2 400 * 75 +  # SP -) MOV,
  4    377 EMT, NEXT, C;
  5 : YEAR ( -> N )  TIME TBUFF @ ;
  6 : MONTH ( -> N )  TIME TBUFF 2+ @ ;
  7 : DAY ( -> N )  TIME TBUFF 4 + @  ;
  8 : HOUR ( -> N )  TIME TBUFF 6 + @  ;
  9 : MINUTE ( -> N )  TIME TBUFF 8 + @  ;
 10 : SECOND ( -> N )  TIME TBUFF 10 + @  ;
 11 : TICK ( -> N )  TIME TBUFF 12 + @  ;
 12 : TICKS/SECOND ( -> N )  TIME TBUFF 14 + @  ;
 13
 14
 15
```

Even though the information is returned very differently than in RT-11, the end user, who sees only 'YEAR', 'MONTH', 'DAY', etc., uses them identically.

Screen 30 (below) defines a system call to read a line from the terminal (using LUN 4, which has already been

assigned in FORTH.MAC). Here 'PUSH' is defined to avoid repetitive writing while pushing the required 12 words onto the stack; note the excursion into the ASSEMBLER vocabulary, necessary since 'SP', etc. are here being used outside of a 'CODE' definition ('CODE' automatically sets the ASSEMBLER vocabulary). An 80-byte buffer and an I/O status block are created; there are more graceful ways to create such buffers, but using 'ALLOT' to extend the two bytes already available in the variable will do. If re-entrancy were desired, these buffers could be assigned elsewhere in memory.

```
SCR # 30
  0 ( RSX-11M SYSTEM-CALL EXAMPLE - TERMINAL I/O)
  1
  2 : PUSH ASSEMBLER SP -) MOV, FORTH ;
  3 0 VARIABLE INBUF 78 ALLOT
  4 0 VARIABLE IOSTAT 2 ALLOT
  5 CODE INPUT 0 # PUSH 0 # PUSH 0 # PUSH 0 # PUSH
  6    120 # PUSH INBUF # PUSH 0 # PUSH IOSTAT # PUSH
  7    4 # PUSH 4 # PUSH 10400 # PUSH 6003 # PUSH
  8    377 EMT, NEXT, C;
  9
 10
 11
 12
 13
 14
 15
```

Forth can create FDB's, etc. for the I/O calls. And Forth's interactive access makes learning and using the system calls faster and more pleasant, as new tests can be run immediately, and Forth eliminates the need for using a separate debugger to see error returns, etc.

VIII. Linkage to Other Languages (RSX)

Forth operations can call subroutines written in other languages, giving access to features not yet implemented in this system (e.g. floating point), and more importantly allowing use of previously written packages, e.g. for statistical analysis or database management. After the calling sequences are defined, the resulting operations behave like any other Forth operations, allowing interactive access to packages not normally used interactively.

This chapter is for systems programmers who are setting up the linkage operations. Application programmers who use the operations don't need to know these details.

At the time of this release, use of this linkage has only begun, and it has only been tested under RSX-11M. (RT-11 should be similar, as the subroutine linkage conventions are the same.) The example presented uses a Fortran subroutine for writing RSX-compatible sequential files from Forth. Any language which can be called from assembly should be callable from Forth.

The RSX-11M (also RT-11) linkage conventions are explained in IAS/RSX Fortran IV User's Guide, Section 2.4; most other PDP-11 operating systems also use the same. Briefly, the calling program points R5 to a word in memory which contains the number of arguments being passed. That word is followed by the addresses of the arguments. Then

the calling program jumps to the entry point of the subroutine, using a JSR.

Forth could set up this calling sequence in various ways; we chose to create the argument list dynamically on the Forth stack. This implementation uses R5 as the Forth stack pointer. PDP-11 stacks grow down; so the Forth calling primitive 'ACALL' pushes the arguments onto the Forth stack in reverse order, then pushes the number of arguments, copies the Forth stack pointer into R5 (unnecessary in this implementation) making R5 point to the argument list, which has been created on the stack, and then does the JSR. After the call, 'ACALL' cleans up the stack by dropping the argument list. Also, 'ACALL' saves and restores R3, R4, and R5, which are important to Forth and might not be restored by the subroutine. Naturally the end user doesn't have to worry about these details.

Screen 26 defines the 'ACALL' primitive. It accepts the argument addresses (in reverse order), the number of arguments, and the entry point address:

```
SCR # 26
  Ø ( FORTRAN LINKAGE, RSX)
  1 CODE ACALL ( ARGS... N ADDR -> .  CALL FORTRAN, ETC.)
  2    S )+ R2 MOV,              ( SAVE ENTRY ADDRESS IN REGISTER)
  3    R3 RP -) MOV, R4 RP -) MOV, R5 RP -) MOV,    ( SAVE R3,R4,R5)
  4    S R5 MOV,                 ( THE STACK WILL BE THE ARG. LIST)
  5    PC R2 () JSR,                        ( LINK THROUGH R2)
  6    RP )+ R5 MOV, RP )+ R4 MOV, RP )+ R3 MOV, ( RESTR R3,R4,R5)
  7    S )+ R2 MOV, R2 R2 ADD, R2 S ADD, ( DROP THE ARGS)
  8    NEXT, C;
  9
 1Ø ( THIS IS AN EXAMPLE - WRITE LINES ON AN RSX FILE)
 11 Ø VARIABLE NFORT
 12 : FILECALL 2 VLINK @ ACALL ;
 13 : OPEN 1 NFORT !  Ø NFORT FILECALL ;
 14 : CLOSE 3 NFORT !  Ø NFORT FILECALL ;
 15 : WRITE ( ADDR ->.  WRITE A LINE) 2 NFORT !  NFORT FILECALL ;
 OK
```

Screen 26 also contains a simple example of a call to a Fortran subroutine which can open, close, or write a file. The subroutine, reproduced below, takes two arguments; a parameter ('1', '2', or '3') telling whether to open, write, or close, and the address where writing begins. Naturally the end user of the file routines won't need to be bothered with the arbitrary parameter values. In this example, the number of characters per line (80) is fixed in the Fortran subroutine; Forth must set up a line of this length.

The end user need only see the operations 'OPEN', 'CLOSE', and 'WRITE'. 'WRITE' takes one argument, the memory address to start writing from; 'OPEN' and 'CLOSE' take no arguments. (A 'READ' operation could have been added of course.)

'FILECALL' simply supplies the number of arguments which will be received by the Fortran (2 arguments), and the entry-point address ('LINKAGE @', explained below), and then performs the 'ACALL'. The variable 'NFORT' is used because the argument list sent to Fortran must contain addresses of the arguments, not the actual values; 'OPEN', 'CLOSE', and 'WRITE' stuff the proper value into 'NFORT', and then provide its address to 'FILECALL'. 'OPEN' and 'CLOSE' supply a dummy buffer address, '0'; 'WRITE' supplies the address which was given to it on the stack.

Linking the Routines

Before the calls can be executed, the Fortran (or other language) routine must be linked with Forth into a task image. This linking is needed only when the subroutines are changed or added to; program development within Forth is still immediate, without need to wait for a link step, and the use of Fortran, etc. is normally transparent to the applications Forth programmer, once the linkage operations have been written.

To avoid reassembly of FORTH.MAC every time a different set of subroutines is linked, FORTH.MAC links indirectly through a single global symbol, VLINK. VLINK is the address of a vector of entry addresses of the subroutines being used. The Forth operation 'VLINK' returns the address of this vector. Therefore 'VLINK @' gets the address of the first entry point, 'VLINK 2 + @' gets the second subroutine's entry address, etc. The Forth operation 'VLINK' does not exist when the 'LINKS' symbol in FORTH.MAC is commented out.

At least under RSX-11M, a special Forth object program must be assembled if any linkage is to be used. This is because Forth normally uses a global symbol to link to the subroutine(s), and if there weren't any routines, a confusing warning message would be produced by the linker. Also, Fortran I/O has difficulty when called from a Macro-11 main program. So when Fortran I/O is to be used, a dummy Fortran main program calls

Forth, which never returns to the dummy. (This dummy program calls 'GFORTH', a global symbol defined at the FORTH.MAC entry point.) Then Forth can call the Fortran I/O subroutines as needed. This procedure requires a change to FORTH.MAC so that an object program is assembled with no entry address.

To make these changes to FORTH.MAC, (1) remove the semicolon which comments out the definition of the 'LINKS' symbol near the beginning of FORTH.MAC, (2) change the last line of FORTH.MAC, from '.END ORIGIN' to '.END', and (3) assemble to get a FORTH.OBJ suitable for linking.

The dummy Fortran program is

```
      CALL    GFORTH
      END
```

and the link vector module is

```
VLINK:: .WORD   OUT
        .END
```

In this example, we are linking to only one Fortran subroutine, named 'OUT'. This subroutine is

```
            SUBROUTINE OUT(N,L)
      C     OUTPUT RSX-COMPATIBLE FILES FROM FORTH
            DIMENSION L(40)
            IF(N .NE. 1) GOTO 2
            CALL ASSIGN (1,'OUT.DAT')
            RETURN
      2     IF(N .NE. 2) GOTO 3
            WRITE(1,101)L
      101   FORMAT(' ',40A2)
            RETURN
      3     IF(N .NE. 3) GOTO 4
            CALL CLOSE(1)
            RETURN
      4     WRITE(4,102)N
      102   FORMAT(' ERROR, BAD ARG TO FORTRAN SUBROUTINE', I7)
            RETURN
            END
```

The following session uses TECO to edit FORTH.MAC for RSX assembly and to allow linkage, assembles the new FORTH.MAC and the link vector module VLINK.MAC, and compiles the Fortran dummy main program DUMMY.FTN and the Fortran OUT.FTN. Then it links these modules, runs Forth, loads the assembler, etc. and loads the linkage example in Screen 26:

```
>TEC FORTH.MAC
*^^^EV$$
^.TITLE F.I.G.
*S;R$$
;R^SX11=1                ; COMMENTED OUT UNLESS RSX11M
*-L$$
^RT11=1          ; COMMENTED OUT UNLESS RT-11
*I;$$
;^RT11=1                 ; COMMENTED OUT UNLESS RT-11
*L$$
^;RSX11=1                ; COMMENTED OUT UNLESS RSX11M
*D$$
^RSX11=1                 ; COMMENTED OUT UNLESS RSX11M
*2L$$
^;LINKS=1        ; COMMENTED OUT UNLESS SUBROUTINE LINKAGE FROM
*D$$
^LINKS=1         ; COMMENTED OUT UNLESS SUBROUTINE LINKAGE FROM
*NHIMEM:$$
HIMEM:^
*2L$$
^        .END    ORIGIN
*KI      .END
$$
^*EX$$

>MAC FORTH=FORTH
>MAC VLINK=VLINK
>FOR DUMMY=DUMMY
>FOR OUT=OUT
>FTB FORTH=DUMMY,FORTH,VLINK,OUT
>RUN FORTH
FIG-FORTH  V 2.Ø
1 LOAD
BYE ISN'T UNIQUE
LOADING EDITOR... R ISN'T UNIQUE I ISN'T UNIQUE
LOADING ASSEMBLER... RØ ISN'T UNIQUE # ISN'T UNIQUE
LOADING STRING PACKAGE...

 OK
26 LOAD
 OK
```

Now we can use 'OPEN', 'WRITE', and 'CLOSE' to define an operation 'FLIST', which lists a Forth screen to an RSX-compatible file (named OUT.DAT - see the Fortran listing above).

```
: IOBUF <BUILDS ALLOT DOES> ;
 OK
8Ø IOBUF OUTBUF
 OK
: FLIST ( 3CREEN -> )  OUTBUF 8Ø BLANKS  OUTBUF WRITE ( BLANK LINE)
   16 Ø DO DUP BLOCK I 64 * + OUTBUF 64 CMOVE OUTBUF WRITE LOOP DROP ;
 OK
: OUT3Ø OPEN 31 1 DO I FLIST LOOP CLOSE ;
 OK
OUT3Ø
 OK
```

Now the RSX file OUT.DAT contains Forth screens 1-30.

The operation 'OUT30' writes the screens. Incidently the data type defined by 'IOBUF' may also be useful elsewhere. 'IOBUF' creates a buffer in the dictionary, but it could have been defined to accept an address and allocate buffers wherever desired in memory.

In case of a trap error in Fortran, RSX cannot return control to Forth, so the job is aborted. In that case, the file FORTH.DAT will probably need to be unlocked.

Various shortcuts for linking subroutines are sometimes possible; FORTH.MAC may not need to be changed and reassembled for linking. The procedure described here is for the general case.

IX. Bring-Up Options

Four pages of the program listing (reproduced below) describe options available through conditional assembly of FORTH.MAC. These options control:

(a) Stand-alone, RT-11, or RSX-11M assembly.

(b) Whether to use the EIS instructions (hardware multiply/divide, etc.)

(c) Whether to produce an object module for linking to subroutines in other languages (see Chapter VIII).

FORTH.MAC as distributed is edited for RT-11 assembly without EIS and without linkage.

If you are using an old version of RT-11 (version 2), note special instructions.

```
                        ;*****************************************************************
                        ;
                        ;       BRINGING UP THE SYSTEM
                        ;
                        ;*****************************************************************
                        ;
                        ;
                        ;
                        ;  TO RUN STAND-ALONE:
                        ;       - BOOT THE DISKETTE LIKE ANY OTHER SYSTEM DISK, FROM DX0.
                        ;         FORTH SHOULD COME UP AND TYPE 'FIG FORTH' AND THE VERSION
                        ;         NUMBER.  TEST AS DESCRIBED FOR RT-11 BELOW.
                        ;       - MAKE A COPY OF THE DISK; THIS STAND-ALONE SYSTEM DOES NOT
                        ;         PROTECT AGAINST ACCIDENTALLY OVERWRITING THE SYSTEM OR THE
                        ;         SOURCE PROGRAMS.  TO MAKE AN EXACT COPY OF THE ENTIRE DISK,
                        ;           1. PUT A BLANK DISK INTO THE SECOND DRIVE (DX1).  FOR
                        ;              SAFETY, SET THE WRITE-PROTECT SWITCH ON THE DRIVE
                        ;              WHICH CONTAINS THE ORIGINAL SYSTEM DISK.
                        ;           2. TYPE '38 LOAD', AND CARRIAGE RETURN.  THE SYSTEM SHOULD
                        ;              RESPOND 'OK'.  THEN TYPE 'COPY' AND RETURN.  EACH OF
                        ;              THE 77 TRACKS WILL BE READ FROM DX0 AND WRITTEN ON DX1.
                        ;       - NOTE THE LAYOUT OF THE DISKETTE.  IT IS SET UP TO BOOT AND
                        ;         RUN STAND-ALONE, BUT IT ALSO CONTAINS AN RT-11 DIRECTORY,
                        ;         AND A MACRO-11 SOURCE PROGRAM 'FORTH.MAC' (WHICH PRODUCED
                        ;         THIS LISTING).  THIS ALLOWS THE SAME DISK TO BE BOOTED
                        ;         AND RUN, OR TO PROVIDE SOURCE FOR MODIFICATION AND RE-ASSEMBLY.
                        ;              AS PROVIDED, THE FILE 'FORTH.DAT' CONTAINS FORTH SCREENS
                        ;         1-70.  YOU CAN USE LOCATIONS BEYOND 70, BUT THESE WILL
                        ;         OVERWRITE THE 'FORTH.MAC' SOURCE PROGRAM.  STAND-ALONE USERS
                        ;         MAY NEVER NEED TO USE THIS SOURCE, AND MAY WANT TO REMOVE IT
                        ;         AND USE THE SPACE FOR SOMETHING ELSE.  MAKE A COPY FIRST.
                        ;       - STAND-ALONE USERS CAN ADD THEIR OWN OPERATIONS AND THEN
                        ;         SAVE A BOOTABLE IMAGE OF THE NEW SYSTEM.  THE NEW OPERATIONS
                        ;         WILL BE AVAILABLE WHEN THE DISK IS BOOTED IN THE FUTURE.
                        ;         THE LOADER WHICH IS USED WILL ONLY LOAD IMAGES UP TO 7.9K;
                        ;         THIS LEAVES SEVERAL HUNDRED BYTES FOR NEW OPERATIONS, WHICH
                        ;         CAN INCLUDE EXTENDING THE SYSTEM BY BRINGING IN SOURCE OR
                        ;         OBJECT CODE.  TO SAVE THE CURRENT SYSTEM, EXECUTE 'FORTH DEFINITIONS'
                        ;         IF NECESSARY TO GET INTO THE FORTH VOCABULARY, THEN 'DECIMAL 34 LOAD'.
                        ;         SOME WARNING MESSAGES WILL BE PRINTED (MSG #4); THEY CAN BE
                        ;         IGNORED.
                        ;       - IF YOU DO WANT TO RE-ASSEMBLE THE SYSTEM FOR STAND-ALONE
                        ;         USE (WHICH MOST USERS SHOULD NEVER FIND NECESSARY),
                        ;         YOU MUST USE RT-11 TO EDIT AND ASSEMBLE 'FORTH.MAC'.  NOTE
                        ;         THAT ALTHOUGH THIS LISTING IS ASSEMBLED FOR STAND-ALONE,
                        ;         THE SOURCE PROGRAM SUPPLIED IS SET FOR RT-11 ASSEMBLY;
                        ;         COMMENT OUT THE 'RT11' DEFINITION, AND REMOVE THE COMMENTING
                        ;         ON 'ALONE'.  ASSEMBLE, LINK, AND RUN, AND THE SYSTEM SHOULD
                        ;         COME UP STAND-ALONE.  IMMEDIATELY REMOVE THE RT-11 SYSTEM DISK
                        ;         AND PLACE THE FORTH DISK IN DRIVE ZERO.  TO REVISE
                        ;         THE BOOTABLE IMAGE ON THE FORTH DISK SO THAT YOUR NEW SYSTEM
                        ;         BOOTS STAND-ALONE, LIST SCREEN 34 (DECIMAL), AND FOLLOW THE
                        ;         INSTRUCTIONS THERE.  THE RUN TAKES ABOUT ONE MINUTE.
                        ;       - THE BOOTABLE SYSTEM DOES NOT USE HARDWARE MULTIPLY AND DIVIDE.
                        ;         IF YOU DON'T HAVE RT-11 TO EDIT AND RECOMPILE WITH 'EIS'
                        ;         CONDITIONAL ASSEMBLY, THE MULTIPLY/DIVIDE ROUTINES CAN BE
                        ;         PATCHED.  IF YOU PATCH FROM THE KEYBOARD MONITOR, THE
```

```
                        ;         RESTART ADDRESS IS 1000 OCTAL ( COLD START) OR 1004 (WARM
                        ;         START).  SAVE THE NEW VERSION AS A BOOTABLE SYSTEM, AS
                        ;         DESCRIBED ABOVE.
                        ;       - THE SKEWED DISK I/O OPERATIONS SKIP TRACK ZERO, FOR COMPATIBILITY
                        ;         WITH STANDARD PDP-11 SECTOR SKEWING.  THE PHYSICAL READ
                        ;         OPERATIONS ('RTS', 'WTS', 'NRTS', 'NWTS') CAN READ ANY SECTOR,
                        ;         HOWEVER.
                        ;       - ALSO THE SYSTEM AS DISTRIBUTED SKIPS THE FIRST 56 SECTORS
                        ;         (7 SCREENS) IN ORDER TO SKIP THE BOOT BLOCK AND AN
                        ;         RT-11 DIRECTORY.  THIS CAUSES THE SCREEN POSITIONS TO BE THE
                        ;         SAME FOR STAND-ALONE AND FOR RT-11 (WHICH ACCESSES THE FILE
                        ;         'FORTH.DAT').  YOU CAN CHANGE THIS BY CHANGING THE VALUE OF
                        ;         THE VARIABLES 'S-SKIP' (NUMBER OF SCREENS SKIPPED) AND
                        ;         'S-USE' (NUMBER OF SCREENS USED BEFORE ACCESSING THE
                        ;         SECOND DISK).  THESE VARIABLES CAN BE CHANGED AT ANY TIME,
                        ;         SO DISK SCREENS CAN BE READ INTO BUFFERS AND THEN FLUSHED
                        ;         TO DIFFERENT LOCATIONS ON THE DISK.
                        ;       - ADVANCED USERS MAY NOTE THAT THIS SYSTEM IS DESIGNED TO
                        ;         ALLOW THE MEMORY LAYOUT - NUMBER AND LOCATION OF DISK
                        ;         BUFFERS, LOCATION OF THE STACK, ETC. - TO BE CHANGED
                        ;         DYNAMICALLY, WITHOUT REASSEMBLY.
                        ;
                        ;
                        ;  TO BRING UP THIS SYSTEM UNDER RT-11:
                        ;       - BE SURE THAT RT-11 IS SELECTED BELOW.  THE LINES DEFINING
                        ;         'RSX11M' AND 'ALONE' SHOULD BE COMMENTED OUT; 'RT11' SHOULD
                        ;         NOT BE.  NOTE THAT THIS DISK IS DISTRIBUTED READY FOR RT-11
                        ;         ASSEMBLY (EVEN THOUGH THIS LISTING IS FOR STAND-ALONE).
                        ;       - IF YOU HAVE HARDWARE MULTIPLY/DIVIDE, ALSO REMOVE THE
                        ;         SEMICOLON FROM THE LINE DEFINING 'EIS'.
                        ;       - IF YOU ARE USING AN OLDER VERSION OF RT-11 (VERSION 2),
                        ;         YOU MAY NEED TO USE THE MACROS '..V2..' AND '.REQDEF'.
                        ;       - ASSEMBLE, LINK, AND RUN.  THE SYSTEM SHOULD COME UP AND
                        ;         TYPE 'FIG-FORTH' AND THE VERSION NUMBER.
                        ;       - TEST THAT IT IS UP BY TRYING SOME ARITHMETIC OR DEFINITIONS, E.G.
                        ;             88 88 * .                (NOTE THAT THE '.' MEANS PRINT)
                        ;             : SQUARE DUP * ;
                        ;             25 SQUARE .
                        ;         OR TYPE 'VLIST' FOR A LIST OF ALL THE FORTH OPERATIONS IN THE
                        ;         DICTIONARY.
                        ;       - THE DISK SHOULD WORK IF THE DISKETTE IS IN DRIVE 'DK'.
                        ;         MAKE SURE THAT 'DK' IS ASSIGNED TO WHATEVER  PHYSICAL
                        ;         DRIVE YOU ARE USING - OR CHANGE LINE 'RTFILE:' IN
                        ;         'FORTH.MAC'.  TEST THE DISK BY TYPING
                        ;              1 LIST
                        ;         WHICH SHOULD LIST THE SCREEN WHICH LOADS THE EDITOR,
                        ;         ASSEMBLER, AND STRING ROUTINES.
                        ;       - IN CASE YOU NEED TO GET A LISTING FROM THE ASSEMBLY OF
                        ;         'FORTH.MAC' (NOT USUALLY NECESSARY), AND YOUR SYSTEM HAS
                        ;         ONLY DISKETTES (NO LARGER DISKS), THE 'ALLOCATE' OPTION
                        ;         IS NECESSARY BECAUSE OF THE SIZE OF THE '.LST' FILE
                        ;         (AROUND 230 BLOCKS).  FIRST COPY 'FORTH.MAC' ONTO A
                        ;         SEPARATE DISKETTE BY ITSELF.  THEN EXECUTE
                        ;               .MACRO /LIST:FORTH.LST /ALLOCATE:300. /NOOBJECT
                        ;         AND REPLY 'FORTH.MAC' WHEN ASKED FOR 'FILES?'.
                        ;
                        ;
```

```
                        ;
                        ;  TO BRING UP THE SYSTEM UNDER RSX-11M:
                        ;       - THE DISKETTE PROVIDED IS IN RT-11 FILE FORMAT.  THE TWO FILES
                        ;         MUST BE COPIED OFF THE DISKETTE INTO AN RSX DIRECTORY.  THE
                        ;         'FORTH.DAT' FILE MUST BE COPIED IN IMAGE MODE.  ANY RSX
                        ;         DIRECTORY MAY BE USED.  ASSUMING THE DISKETTE IS IN DRIVE 0,
                        ;         USE THE RSX COMMANDS:
                        ;               >FLX =DX:FORTH.MAC/RT
                        ;               >FLX =DX:FORTH.DAT/RT/IM
                        ;         INCIDENTALLY, 'FORTH.DAT' IS THE SYSTEM'S 'VIRTUAL MEMORY'
                        ;         FILE, USED FOR DISK I/O.  THE REST OF THE SYSTEM (THIS
                        ;         PROGRAM ALONE) CAN RUN INDEPENDENTLY, EVEN IF 'FORTH.DAT'
                        ;         IS NOT AVAILABLE.
                        ;       - EDIT 'FORTH.MAC' TO SELECT RSX ASSEMBLY.  CHANGE THE SEMICOLON
                        ;         TO COMMENT OUT 'RT11' NOT 'RSX11'.  LET 'EIS' BE DEFINED IF
                        ;         YOU HAVE HARDWARE MULTIPLY/DIVIDE.
                        ;       - ASSEMBLE, TASK BUILD, AND RUN.  TEST AS WITH RT11 ABOVE.
                        ;       - THE DISK I/O SHOULD WORK IF 'FORTH.DAT' IS IN THE DEFAULT
                        ;         DEVICE AND DIRECTORY.  TEST AS ABOVE.
                        ;
                        ;
                        ;
                        ;  THE SYSTEM AS SUPPLIED RESERVES 8000. BYTES FOR YOUR FORTH
                        ;  PROGRAMMING AND STACK.  THIS IS ENOUGH FOR SUBSTANTIAL PROJECTS.
                        ;  (NOTE THAT THE EDITOR, ASSEMBLER, AND STRING PACKAGE, IF LOADED,
                        ;  USE MORE THAN 5K OF THIS.)  TO CHANGE THIS MEMORY SIZE, CHANGE
                        ;  THE '8000.' WHICH IS IN THE LINES FOLLOWING THE LABEL 'DP:',
                        ;  NEAR THE END OF THIS PROGRAM.  INCIDENTALLY, VERY FEW JOBS
                        ;  (E.G. RECURSION) WILL EVER USE MORE THAN 100 WORDS OF THIS SPACE
                        ;  FOR THE STACK; THE REST OF THE SPACE IS AVAILABLE FOR A STRING
                        ;  STACK (IF USED) OR FOR YOUR PROGRAMS - AND FORTH OBJECT CODE IS
                        ;  CONSIDERABLY MORE COMPACT THAN ASSEMBLY.
                        ;
                        ;
                        ;
                        ;  THE FORTH VIRTUAL FILE 'FORTH.DAT' IS USED FOR STORING SOURCE
                        ;  PROGRAMS (OR DATA).  THIS FILE HAS 70 1-K SCREENS (1-70),
                        ;  I.E. 140 PDP-11 DISK BLOCKS.  SCREENS 4 AND 5 ARE USED BY THE
                        ;  SYSTEM FOR STORING ERROR AND WARNING MESSAGES.  SCREENS 6-30
                        ;  CONTAIN A TEXT EDITOR, ASSEMBLER, STRING PACKAGE, AND MISCELLANEOUS
                        ;  EXAMPLES.  SCREENS 40 THROUGH 47 CONTAIN A BINARY STAND-ALONE
                        ;  SYSTEM (NOT USED UNDER RT-11 OR RSX-11M).  USERS MAY WANT
                        ;  TO SAVE THEIR SOURCE PROGRAMS AND DATA IN THE BLANK SCREENS.
                        ;  THE SIZE OF THIS FORTH SCREENS FILE ('FORTH.DAT') CAN BE INCREASED
                        ;  IF NEEDED.  IF THE SYSTEM IS TO BE BOOTED STAND-ALONE, THE LOCATION
                        ;  OF THE SYSTEM BINARY IMAGE ON THE DISK MUST NOT BE CHANGED;
                        ;  THEREFORE, IF THE DISK IS TO BE USED TO RUN STAND-ALONE, DO NOT
                        ;  USE RT-11 TO MOVE 'FORTH.DAT' TO ANOTHER PLACE ON THE DISK.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;  NOTE THAT THE RT-11 AND RSX-11M SYSTEMS DO NOT ECHO CHARACTERS
                        ;  WHICH ARE INPUT FROM THE TERMINAL.  INSTEAD, THEY LET THE OPERATING
                        ;  SYSTEM (RT-11 OR RSX-11M) ECHO THEM.  THIS IS DONE SO THAT TYPING
                        ;  CONVENTIONS WILL BE THE SAME AS THE USER IS FAMILIAR WITH.  ALSO,
```

```
                        ;  TO AVOID SWAPPING DELAYS, THE RSX VERSION OF 'KEY' READS A LINE OF
                        ;  CHARACTERS AT A TIME.
                        ;
                        ;
                        ;
                        ;
                        ;  CHANGE THESE LINES TO CONTROL CONDITIONAL ASSEMBLY:
                        ;
                        ;RT11=1          ; COMMENTED OUT UNLESS RT-11
                        ;RSX11=1         ; COMMENTED OUT UNLESS RSX11M
000001                  ALONE=1          ; COMMENTED OUT UNLESS STAND-ALONE
                        ;EIS=1           ; COMMENTED OUT UNLESS HARDWARE MULTIPLY-DIVIDE
                        ;LINKS=1         ; COMMENTED OUT UNLESS SUBROUTINE LINKAGE FROM
                        ;                        FORTH TO OTHER LANGUAGES
                        ;
                        ;.PAGE
                        ; ****************************************************************
                        ;
                        ;               VARIATIONS FROM F.I.Q. MODEL
                        ;
                        ; ****************************************************************
```

X. Documentation Hints

Programs in Forth or other extensible languages especially need documentation if other programmers are going to maintain them. Forth coding includes new operations which are not part of the standard language; these must be described. Also, they should be designed to form coherent application-related groups.

The user needs to know what arguments an operation takes from the stack, and what results it returns. This information can be abbreviated in a comment in source code on disk. E.g. see Screen #6 in the listing of FORTH.DAT below. The comment in line 3 '(→ N)' indicates that the operation '#ARGS' takes no arguments from the stack, and returns a number to it. 'GETWORD' takes no arguments and returns no result; 'CURSADDR' takes no arguments and returns an address; 'NLINE' returns a line #, etc. Since these words are only used internally in the editor, they aren't described in the editor user documentation.

The next level of documentation is the glossary. The glossary should repeat the stack information, and tell what the word does. Often a couple sentences is enough. The Forth Interest Group glossary reproduced below shows that higher-level words needn't take any more space than lower-level ones. This is probably because words are selected for purposes

chosen by human users, but whatever the reason, a consequence is that program complexity tends to grow linearly with size.

One more requirement for understandable programs is the design of coherent application-oriented groupings or vocabularies, operation sets which in effect make Forth into a special application language. Large applications should have a hierarchical structure of various levels of language. At any level there will probably be special-purpose operations which are only used locally to define other words, not used later. Forth has facilities to hide these words from the user, but that may not be worth the trouble, as the user is allowed to redefine those names as something else with no more penalty than a "not unique" warning message. User glossaries should omit those words or list them separately, however.

Documentation for maintenance programmers can be a narrative walk-through explaining the purpose of coding decisions and the working of any unusual code. Also, any special-purpose operations omitted from the user's glossary should be described here.

XI. FORTH.DAT listing

Many of the following source screens are explained elsewhere in the text.

```
1 39 INDEX


  1 ( LOAD SCREEN)
  2
  3
  4 (  ERROR, WARNING, AND OTHER MESSAGES - SCREENS 4 AND 5 )
  5 (  ERROR MESSAGES, CONTINUED  )
  6 ( EDITOR - SET-UP)
  7 ( EDITOR - OPERATIONS)
  8 ( EDITOR, SCREEN 3)
  9 ( EDITOR, SCREEN 4)
 10 ( ASSEMBLER)                                OCTAL
 11 ( ASSEMBLER, CONT.)                         OCTAL
 12 ( ASSEMBLER - INSTRUCTION TABLE)            OCTAL
 13 ( ASSEMBLER - CONT.)                        OCTAL
 14 ( ASSEMBLER - REGISTERS, MODES, AND CONDITIONS)   OCTAL
 15 ( ASSEMBLER - STRUCTURED CONDITIONALS)  OCTAL
 16
 17 ( ASSEMBLER - EXAMPLES)
 18
 19 ( STRING ROUTINES)          DECIMAL
 20 ( STRINGS - CONTINUED)
 21 ( STRINGS - CONTINUED)
 22 ( STRINGS - CONTINUED )
 23
 24 ( TRIG LOOKUP ROUTINES - WITH SINE *10000 TABLE)
 25
 26 ( FORTRAN LINKAGE, RSX)
 27
 28 ( RT-11 SYSTEM-CALL EXAMPLE - DATE)
 29 ( RSX-11M SYSTEM-CALL EXAMPLE - DATE)
 30 ( RSX-11M SYSTEM-CALL EXAMPLE - TERMINAL I/O)
 31
 32 ( EXAMPLES - RANDOM #S, VIRTUAL ARRAY, RECURSIVE CALL)
 33
 34 ( CREATE BOOTABLE IMAGE ON SCREENS 40-47.  FOR STAND-ALONE.)
 35 ( CREATE A BINARY IMAGE ON SCREENS 40 - 47  )
 36 ( CREATE BOOT LOADER.  NOTE - DOES NOT WRITE BOOT BLOCK)
 37 ( CREATE BOOT LOADER, CONT.)      OCTAL
 38 ( DISK COPY FROM SYSTEM DISK TO DX1)
 39 ( ** CAUTION **   BINARY IMAGE IN SCREENS 40-47) OK
```

```
SCR # 1
  Ø ( LOAD SCREEN)
  1 DECIMAL
  2 1 WARNING ! ( GET ERR MSGS, NOT #S)
  3
  4 CR ." LOADING EDITOR... "  6 LOAD 7 LOAD 8 LOAD 9 LOAD
  5 CR ." LOADING ASSEMBLER... "  1Ø LOAD 11 LOAD 12 LOAD 13 LOAD
  6          14 LOAD 15 LOAD
  7 CR ." LOADING STRING PACKAGE... "  19 LOAD 2Ø LOAD 21 LOAD
  8          22 LOAD
  9 CR
 1Ø : BYE FLUSH CR ." LEAVING FORTH.  HAVE A GOOD DAY." CR BYE ;
 11 CR
 12
 13
 14
 15
```

```
SCR # 2
  Ø
  1
  2
  3
  4
  5
  6
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

```
SCR # 3
  Ø
  1
  2
  3
  4
  5
  6
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

```
SCR # 4
  Ø (  ERROR, WARNING, AND OTHER MESSAGES - SCREENS 4 AND 5 )
  1 EMPTY STACK
  2 STACK OR DICTIONARY FULL
  3 HAS INCORRECT ADDRESS MODE
  4 ISN'T UNIQUE
  5
  6 DISC RANGE
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15 FORTH INTEREST GROUP                              MAY 1979
```

```
SCR # 5
  Ø (  ERROR MESSAGES, CONTINUED  )
  1 COMPILATION ONLY, USE IN DEFINITION
  2 EXECUTION ONLY
  3 CONDITIONALS NOT PAIRED
  4 DEFINITION NOT FINISHED
  5 IN PROTECTED DICTIONARY
  6 USE ONLY WHEN LOADING
  7
  8 DECLARE VOCABULARY
  9
 1Ø
 11
 12
 13
 14
 15
```

```
SCR # 6
  Ø ( EDITOR - SET-UP)
  1 VOCABULARY EDITOR IMMEDIATE      Ø VARIABLE ESCR    DECIMAL
  2 Ø VARIABLE CURSOR   Ø VARIABLE STACKPTR  : STK SP@ STACKPTR ! ;
  3 : #ARGS ( ->N ) SP@ STACKPTR @ SWAP - 2 / Ø MAX ;
  4 : E ESCR @ BLOCK DROP [COMPILE] EDITOR STK ;
  5 : EDIT -DUP IF ESCR ! Ø CURSOR ! E ELSE ."  ERR Ø ARG" ENDIF ;
  6 EDITOR DEFINITIONS
  7 : EX FLUSH [COMPILE] FORTH ;
  8 ( THE FOLLOWING ARE UTILITY ROUTINES FOR LATER DEFINITIONS.)
  9 : GETWORD ( -> )  1 WORD   HERE 1+ C@ Ø= IF Ø HERE C! ENDIF ;
 1Ø : GETPAD ( -> )  GETWORD   HERE PAD 65 CMOVE ;
 11 : RANGE ( -> ) CURSOR @ Ø MAX 1Ø23 MIN CURSOR ! ;
 12 : CURSADDR ( -> ADDR ) ESCR @ BLOCK UPDATE CURSOR @ + ;
 13 : CLINE ( -> POSITION ) CURSOR @ 64 MOD ;
 14 : LINEADDR ( -> ADDR ) CURSADDR CLINE - ;
 15 : NLINE ( -> LINE# ) CURSOR @ 64 / ;
```

```
SCR # 7
  Ø ( EDITOR - OPERATIONS)
  1 : LDEFAULT ( N? -> N ) #ARGS IF 64 * CURSOR ! RANGE ENDIF ;
  2 : 1DEFAULT ( N? -> N ) #ARGS Ø= IF 1 ENDIF ;
  3 : NEW LDEFAULT 16 NLINE DO CR QUERY GETPAD PAD 1+ C@ 32 <
  4    IF LEAVE ELSE LINEADDR 64 BLANKS PAD 1+ LINEADDR PAD C@
  5    64 MIN CMOVE 64 CURSOR +! RANGE ENDIF LOOP STK ;
  6 : T LDEFAULT CR  LINEADDR PAD 64 CMOVE CURSADDR PAD CLINE + 1+
  7    64 CLINE  - CMOVE 95 PAD CLINE + C! NLINE 3 .R SPACE
  8    PAD 65 TYPE STK ;
  9 : R LDEFAULT GETPAD PAD 1+ CURSADDR PAD C@
 10    64 MIN CMOVE PAD C@ CURSOR +! RANGE  T STK ;
 11 : L  ESCR @ FORTH LIST EDITOR CR T STK ;
 12 : M 1DEFAULT CURSOR +! RANGE T STK ;
 13 : TRADE ( M,N---) 2 Ø DO 64 * CURSOR ! RANGE LINEADDR SWAP LOOP
 14    DUP PAD 64 CMOVE OVER SWAP 64 CMOVE PAD SWAP 64 CMOVE STK ;
 15
```

```
SCR # 8
  Ø ( EDITOR, SCREEN 3)
  1 : D-+ONLY DUP CURSADDR + CURSADDR 64 CLINE - CMOVE
  2    LINEADDR 64 + OVER - SWAP BLANKS T ;
  3 : D ( ADJUST ARG IF NEG, DEFAULT, OUT OF LINE) 1DEFAULT  DUP Ø<
  4    IF ( NEGATIVE ARG) CLINE MINUS MAX DUP CURSOR +! ABS
  5    ELSE 64 CLINE - MIN ENDIF   -DUP IF D-+ONLY ENDIF   STK ;
  6 : I LDEFAULT CURSADDR PAD 64 CMOVE   GETWORD   HERE 1+ CURSADDR
  7    HERE C@ CMOVE   PAD  CURSADDR HERE C@ + LINEADDR 64 + OVER -
  8    Ø MAX CMOVE HERE C@ CURSOR +! RANGE T STK ;
  9 : COMP ( ADDR ADDR LEN -> BOOL.  TEST FOR STRINGS EQUAL)
 10    OVER + SWAP DO DUP C@ FORTH I C@ -
 11       IF ( UNEQUAL) DROP Ø LEAVE ELSE 1+ ENDIF   LOOP ;
 12 : SEARCH ( ADDR LEN -> ADDR-OR-Ø )   HERE C@ -   1 MAX
 13    OVER + SWAP   Ø ROT ROT   DO FORTH I HERE 1+ HERE C@ COMP
 14    IF DROP FORTH I LEAVE ENDIF LOOP ;
 15
```

```
SCR # 9
  Ø ( EDITOR, SCREEN 4)
  1 Ø VARIABLE SAVESTRING  64 ALLOT  ( TO STORE SEARCH STRING)
  2 : SAVEARG ( ->.  SAVE OR RESTORE SEARCH STRING ARGUMENT)
  3    HERE 1+ C@ IF ( NOT NULL) HERE SAVESTRING HERE C@ 1+ CMOVE
  4    ELSE ( NULL) SAVESTRING HERE SAVESTRING C@ 1+ CMOVE ENDIF ;
  5 : S ( -> ) LDEFAULT  1 WORD   SAVEARG   CURSADDR 1Ø24 CURSOR @
  6    - SEARCH -DUP IF CURSADDR - HERE C@ + CURSOR +! RANGE ENDIF
  7    T STK ;
  8 : -R   SAVESTRING C@ MINUS D I ;
  9 : SCRATCH EMPTY-BUFFERS EX ;
 10 : SPREAD ( N -> )  LDEFAULT NLINE DUP 14 > IF ." CAN'T SPREAD"
 11    CR   ELSE Ø MAX DUP 1 - 14 DO FORTH I EDITOR DUP 1+ TRADE
 12    -1 +LOOP   64 * ESCR @ BLOCK +  64 BLANKS  ENDIF ;
 13 FORTH DEFINITIONS
 14 : SCREENMOVE ( FROM TO -> ) FLUSH SWAP BLOCK SWAP BLOCK UPDATE
 15    1024 CMOVE ;
```

```
SCR # 10
  0 ( ASSEMBLER)                                    OCTAL
  1 VOCABULARY ASSEMBLER IMMEDIATE              0 VARIABLE OLDBASE
  2 : ENTERCODE [COMPILE] ASSEMBLER  BASE @ OLDBASE ! OCTAL  SP@ ;
  3 : CODE CREATE ENTERCODE ;
  4 ASSEMBLER DEFINITIONS
  5  ' ENTERCODE 2 -    ' ;CODE 10 +    !     ( PATCH ';CODE')
  6 : FIXMODE ( COMPLETE THE MODE PACKET)
  7    DUP -1 = IF DROP ELSE DUP 10 SWAP U< IF 67 ENDIF ENDIF ;
  8 : OP <BUILDS , DOES> @ , ;
  9 : ORMODE ( MODE ADDR -> .   SET MODE INTO INSTR.)
 10    SWAP OVER @ OR SWAP ! ;
 11 : ,OPERAND ( ?OPERAND MODE -> ) DUP 67 = OVER 77 = OR IF ( PC)
 12    SWAP HERE 2 + - SWAP ENDIF  DUP 27 = OVER 37 = OR ( LITERAL)
 13    SWAP 177760 AND 60 = OR ( RELATIVE) IF , ENDIF ;
 14 : 1OP <BUILDS , DOES> @ , FIXMODE  DUP HERE 2 -
 15    ORMODE ,OPERAND ;                           DECIMAL
```

```
SCR # 11
  0 ( ASSEMBLER, CONT.)                             OCTAL
  1 : SWAPOP ( -> .  EXCHANGE OPERANDS OF 3-WORD INSTR, ADJ. PC-REL)
  2    HERE 2 - @ HERE 6 - @ 6700 AND 6700 = IF ( PC-REL) 2 + ENDIF
  3    HERE 4 - @ HERE 6 - @ 67 AND 67 = IF ( PC-REL) 2 - ENDIF
  4    HERE 2 - ! HERE 4 - ! ;
  5 : 2OP <BUILDS , DOES> @ ,
  6    FIXMODE  DUP HERE 2 -  DUP >R ORMODE ,OPERAND
  7    FIXMODE  DUP 100 * R ORMODE  ,OPERAND  HERE R> - 6 =
  8    IF SWAPOP ENDIF ;
  9 : ROP <BUILDS , DOES> @ , FIXMODE DUP HERE 2 - DUP >R ORMODE
 10    ,OPERAND DUP 7 SWAP U< IF ."   ERR-REG-B " ENDIF
 11    100 * R> ORMODE ;
 12 : BOP <BUILDS , DOES> @ ,  HERE -   DUP 376 >
 13    IF ."   ERR-BR+ " . ENDIF   DUP -400 < IF ."   ERR-BR- " .
 14    ENDIF     2 / 377 AND HERE 2 - ORMODE ;
 15                                                 DECIMAL
```

```
SCR # 12
  0 ( ASSEMBLER - INSTRUCTION TABLE)                OCTAL
  1 010000 2OP MOV,     110000 2OP MOVB,     020000 2OP CMP,
  2 120000 2OP CMPB,    060000 2OP ADD,      160000 2OP SUB,
  3 030000 2OP BIT,     130000 2OP BITB,     050000 2OP BIS,
  4 150000 2OP BISB,    040000 2OP BIC,      140000 2OP BICB,
  5 005000 1OP CLR,     105000 1OP CLRB,     005100 1OP COM,
  6 105100 1OP COMB,    005200 1OP INC,      105200 1OP INCB,
  7 005300 1OP DEC,     105300 1OP DECB,     005400 1OP NEG,
  8 105400 1OP NEGB,    005700 1OP TST,      105700 1OP TSTB,
  9 006200 1OP ASR,     106200 1OP ASRB,     006300 1OP ASL,
 10 106300 1OP ASLB,    006000 1OP ROR,      106000 1OP RORB,
 11 006100 1OP ROL,     106100 1OP ROLB,     000300 1OP SWAB,
 12 005500 1OP ADC,     105500 1OP ADCB,     005600 1OP SBC,
 13 105600 1OP SBCB,    006700 1OP SXT,      000100 1OP JMP,
 14 074000 ROP XOR,     004000 ROP JSR,
 15 : RTS, 200 OR , ;                        DECIMAL
```

```
SCR # 13
  0 ( ASSEMBLER - CONT.)                          OCTAL
  1 000400 BOP BR,      001000 BOP BNE,       001400 BOP BEQ,
  2 100000 BOP BPL,     100400 BOP BMI,       102000 BOP BVC,
  3 102400 BOP BVS,     103000 BOP BCC,       103400 BOP BCS,
  4 002000 BOP BGE,     002400 BOP BLT,       003400 BOP BLE,
  5 101000 BOP BHI,     101400 BOP BLOS,      103000 BOP BHIS,
  6 103400 BOP BLO,     003000 BOP BGT,       000003 OP BPT,
  7 000004 OP IOT,      000002 OP RTI,        000006 OP RTT,
  8 000000 OP HALT,     000001 OP WAIT,       000005 OP RESET,
  9 000241 OP CLC,      000242 OP CLV,        000244 OP CLZ,
 10 000250 OP CLN,      000261 OP SEC,        000262 OP SEV,
 11 000264 OP SEZ,      000270 OP SEN,        000277 OP SCC,
 12 000257 OP CCC,      000240 OP NOP,        006400 OP MARK,
 13 : EMT, 104000 + , ;
 14
 15                                             DECIMAL
```

```
SCR # 14
  0 ( ASSEMBLER - REGISTERS, MODES, AND CONDITIONS)   OCTAL
  1 : C CONSTANT ;        0 C R0   1 C R1   2 C R2   3 C R3  4 C R4
  2    5 C R5  6 C SP  7 C PC  2 C W  3 C U  4 C IP  5 C S  6 C RP
  3 :  RTST ( R MODE -> MODE)  OVER DUP 7 > SWAP 0 < OR
  4    IF ."   NOT A REGISTER: " OVER . ENDIF + -1 ;
  5 : )+ 20 RTST ;       : -) 40 RTST ;      : I) 60 RTST ;
  6 : @)+ 30 RTST ;      : @-) 50 RTST ;     : @I) 70 RTST ;
  7 : # 27 -1 ;          : @# 37 -1 ;
  8 : () DUP 10 U< IF ( REGISTER DEFERRED) 10 + -1
  9    ELSE ( RELATIVE DEFERRED) 77 -1 ENDIF ;
 10 ( NOTE - THE FOLLOWING CONDITIONALS REVERSED FOR 'IF,', ETC. )
 11 001000 C EQ   001400 C NE      100000 C MI   100400 C PL
 12 102000 C VS   102400 C VC      103000 C CS   103400 C CC
 13 002000 C LT   002400 C GE      003000 C LE   003400 C GT
 14 101000 C LOS  101400 C HI      103000 C LO   103400 C HIS
 15                                             DECIMAL
```

```
SCR # 15
  0 ( ASSEMBLER - STRUCTURED CONDITIONALS)  OCTAL
  1
  2 : IF, ( CONDITION -> ADDR ) HERE SWAP , ;
  3 : IPATCH ( ADDR ADDR -> . )  OVER - 2 / 1 -   377 AND
  4    SWAP DUP @ ROT OR SWAP ! ;
  5 : ENDIF, ( ADDR -> ) HERE IPATCH ;  : THEN, ENDIF, ;
  6 : ELSE, ( ADDR -> ADDR ) 00400 ,   HERE IPATCH   HERE 2 - ;
  7 : BEGIN, ( -> ADDR )  HERE ;
  8 : WHILE, ( CONDITION -> ADDR )  HERE SWAP , ;
  9 : REPEAT, ( ADDR ADDR -> ) HERE 400 , ROT IPATCH HERE IPATCH ;
 10 : UNTIL, ( ADDR CONDITION -> )  , HERE 2 - SWAP IPATCH ;
 11 : C; CURRENT @ CONTEXT !   OLDBASE @ BASE !   SP@ 2+ =
 12   IF SMUDGE ELSE ."  CODE ERROR, STACK DEPTH CHANGED  " ENDIF ;
 13
 14 : NEXT, IP )+ W MOV,   W @)+ JMP, ;
 15 FORTH DEFINITIONS                           DECIMAL
```

```
SCR # 16
  Ø
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 17
  Ø ( ASSEMBLER - EXAMPLES)
  1 CODE TEST1 33ØØ6 # 33ØØØ MOV, NEXT, C;
  2 CODE TEST2 555 # 33ØØØ () MOV, NEXT, C;
  3 CODE TESTDUP S () S -) MOV, NEXT, C;
  4 CODE TESTØ RØ S -) MOV, NEXT, C;
  5 CODE TESTBYTE 33ØØ6 R1 MOVB, R1 S -) MOV, NEXT, C;
  6 CODE TEST3 33ØØØ # R1 MOV, 444 # 2Ø R1 I) MOV, NEXT, C;
  7 CODE TEST-DUP S () TST, NE IF, S () S -) MOV, ENDIF, NEXT, C;
  8 CODE TESTLP1 15 # R1 MOV, BEGIN, R1 DEC, GT WHILE, R1 S -) MOV,
  9    REPEAT, NEXT,  C;
 1Ø CODE TESTLP2 15 # R1 MOV, BEGIN, R1 S -) MOV, R1 DEC,
 11    EQ UNTIL, NEXT, C;
 12 : TESTVARIABLE CONSTANT ;CODE W S -) MOV, NEXT, C;
 13
 14
 15
```

```
SCR # 18
  Ø
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 19
  Ø ( STRING ROUTINES)          DECIMAL
  1 ( NOTE: STRING-STACK PTR, $SP, IS 3ØØ BYTES FROM STACK ORIGIN)
  2 3ØØ VARIABLE STACKSIZE      SØ @ STACKSIZE @ - VARIABLE $SP
  3 : $CLEAR ( -> )  SØ @ STACKSIZE @ - $SP ! ;      $CLEAR
  4 : $LEN ( -> LENGTH . LENGTH OF TOP OF $STACK) $SP @ DUP SØ @
  5    STACKSIZE @ - < Ø= IF ." $STACK EMPTY" QUIT ELSE @ ENDIF ;
  6 : $DROP ( ->. DROP FROM $STACK)  $LEN 2+ =CELLS $SP +! ;
  7 : $COUNT ( ADDR -> ADDR LENGTH) DUP 2+ SWAP @ ;
  8 : $. ( ->.  PRINT STRING)  $SP @ $COUNT -TRAILING TYPE $DROP ;
  9 : $?OVER ( N-> . ) HERE 256 + +   $SP @  < Ø=
 1Ø    IF ." WOULD CAUSE $OVERFLOW" QUIT ENDIF ;
 11 : $@TEXT ( ADDR CNT  ->.  MOVE TEXT INTO $STACK)  DUP 2+ =CELLS
 12    DUP $?OVER   MINUS $SP +! $SP @ ! $SP @ $COUNT CMOVE ;
 13 : $@ ( FROM-ADDR -> .  STRING INTO $STACK)  $COUNT $@TEXT ;
 14 : (") R COUNT DUP 1+ =CELLS R> + >R $@TEXT ;
 15


SCR # 2Ø
  Ø ( STRINGS - CONTINUED)
  1 : $NULL ( CREATE NULL STRING) -2 $SP +! Ø $SP @ ! ;
  2 : " ( ->.  STRING TO $STACK - COMPILE OR EXECUTE) STATE @
  3    IF COMPILE (") 34 WORD HERE C@ 1+ =CELLS ALLOT
  4    ELSE 34 WORD HERE COUNT $@TEXT ENDIF ; IMMEDIATE
  5 : $! ( TO-ADDR -> .  MOVE STRING FROM $STACK TO MEMORY.)
  6    $SP @ SWAP $LEN 2+ CMOVE $DROP ;
  7 : $DIM ( LEN -> .  CREATES STRING VARIABLE OF GIVEN LENGTH.)
  8    Ø CONSTANT   HERE HERE 2 - !   2+ =CELLS ALLOT ;
  9 : $VARIABLE ( -> .  CREATES $VAR FROM $STACK TOP.)  $LEN $DIM
 1Ø    $SP @   HERE $LEN 2+ =CELLS -   $LEN 2+   CMOVE   $DROP ;
 11 : $DUP ( -> )  $SP @ $@ ;
 12 : $SEG ( BEGIN END -> )  OVER - 1+ SWAP 1 -
 13    $SP @ 2+ + SWAP $@TEXT ;
 14 : $STR ( N -> )  S->D SWAP OVER DABS <# #S SIGN #> $@TEXT ;
 15


SCR # 21
  Ø ( STRINGS - CONTINUED)
  1 : $VAL ( -> N . POSITIVE ONLY)  HERE 33 32 FILL
  2    $SP @ 2+ HERE $LEN CMOVE Ø S->D HERE 1 - (NUMBER)
  3    DROP $DROP DROP ;
  4 : $SECOND $LEN =CELLS 2+ $SP @ + DUP SØ @ STACKSIZE @ - < Ø=
  5    IF ." ERROR, NO SECOND STRING" QUIT ENDIF ;
  6 : $OVER $SECOND $@ ;
  7 : MOVEW ( FROM TO NBYTES -> .  LIKE 'CMOVE' BUT FROM HIGH END)
  8    2 - -2 SWAP DO OVER I + @ OVER I + ! -2 +LOOP DROP DROP ;
  9 : $SWAP ( -> )   $OVER   $SP @ ( FROM)   DUP $LEN =CELLS 2+ +
 1Ø    ( TO)   $LEN =CELLS 2+ $SECOND @ =CELLS 2+ + ( # OF BYTES)
 11    MOVEW   $LEN =CELLS 2+ $SP +! ;
 12
 13
 14
 15
```

```
SCR # 22
  0 ( STRINGS - CONTINUED )
  1 0 VARIABLE TEMP
  2 : $COMP ( -> NEG OR 0 OR POS.  COMPARE STRINGS)  0 TEMP !
  3    $SECOND 2+ $SP @ 2+ ( STRING TEXT ADDRESSES)
  4    $LEN $SECOND @ MIN ( # CHARACTERS TO COMPARE)
  5    0 DO OVER I + C@ OVER I + C@ - -DUP IF LEAVE TEMP ! ENDIF
  6    LOOP   DROP DROP TEMP @ $DROP $DROP ;
  7 : $< ( -> BOOL )   $COMP 0< ;
  8 : $= ( -> BOOL )   $COMP 0= ;
  9 : $> ( -> BOOL )   $COMP 0 > ;
 10 : $+-EVEN ( -> ) $LEN    $SWAP   $SP @ ( FROM) DUP 2+
 11    $LEN =CELLS 2+ ( #) MOVEW   2 $SP +!   $LEN + $SP @ ! ;
 12 : $+ ( -> . CONCATENATE )  $LEN $SECOND @ ( SAVE LENGTHS)
 13    $+-EVEN   DUP 1 AND IF $SP @ 2+ + OVER SWAP DUP 1+ SWAP ROT
 14    CMOVE 1 AND IF $SP @ DUP 2+ $LEN 2+ MOVEW  2 $SP +!  ENDIF
 15    ELSE DROP DROP ENDIF ;

SCR # 23
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

SCR # 24
  0 ( TRIG LOOKUP ROUTINES - WITH SINE *10000 TABLE)
  1 : TABLE <BUILDS 0 DO , LOOP DOES> SWAP 2 * + @ ;
  2 10000 9998 9994 9986 9976 9962 9945 9925 9903 9877
  3  9848 9816 9781 9744 9703 9659 9613 9563 9511 9455
  4  9397 9336 9272 9205 9135 9063 8988 8910 8829 8746
  5  8660 8572 8480 8387 8290 8192 8090 7986 7880 7771
  6  7660 7547 7431 7314 7193 7071 6947 6820 6691 6561
  7  6428 6293 6157 6018 5878 5736 5592 5446 5299 5150
  8  5000 4848 4695 4540 4384 4226 4067 3907 3746 3584
  9  3420 3256 3090 2924 2756 2588 2419 2250 2079 1908
 10  1736 1564 1392 1219 1045 0872 0698 0523 0349 0175
 11  0000         91 TABLE SINTABLE
 12 : S180 DUP 90 > IF 180 SWAP - ENDIF SINTABLE ;
 13 : SIN ( N -> SIN)  360 MOD DUP 0< IF 360 + ENDIF DUP 180 >
 14    IF 180 - S180 MINUS ELSE S180 ENDIF ;
 15 : COS ( N -> COS) 90 + SIN ;
```

```
SCR # 25
  Ø
  1
  2
  3
  4
  5
  6
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

```
SCR # 26
  Ø ( FORTRAN LINKAGE, RSX)
  1 CODE ACALL ( ARGS... N ADDR -> .  CALL FORTRAN, ETC.)
  2    S )+ R2 MOV,              ( SAVE ENTRY ADDRESS IN REGISTER)
  3    R3 RP -) MOV, R4 RP -) MOV, R5 RP -) MOV,    ( SAVE R3,R4,R5)
  4    S R5 MOV,                 ( THE STACK WILL BE THE ARG. LIST)
  5    PC R2 () JSR,                        ( LINK THROUGH R2)
  6    RP )+ R5 MOV, RP )+ R4 MOV, RP )+ R3 MOV, ( RESTR R3,R4,R5)
  7    S )+ R2 MOV, R2 R2 ADD, R2 S ADD, ( DROP THE ARGS)
  8    NEXT, C;
  9
 1Ø ( THIS IS AN EXAMPLE - WRITE LINES ON AN RSX FILE)
 11 Ø VARIABLE NFORT
 12 : FILECALL 2 VLINK @ ACALL ;
 13 : OPEN 1 NFORT !  Ø NFORT FILECALL ;
 14 : CLOSE 3 NFORT !  Ø NFORT FILECALL ;
 15 : WRITE ( ADDR ->.  WRITE A LINE) 2 NFORT !  NFORT FILECALL ;
```

```
SCR # 27
  Ø
  1
  2
  3
  4
  5
  6
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

```
SCR # 28
  0 ( RT-11 SYSTEM-CALL EXAMPLE - DATE)
  1 CODE DATE 12 400 * # R0 MOV, 374 EMT, R0 S -) MOV, NEXT, C;
  2 : YEAR ( -> N )  DATE 31 AND 72 + ;
  3 : DAY ( -> N )  DATE 32 / 31 AND ;
  4 : MONTH ( -> N)  DATE 1024 / 15 AND ;
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 29
  0 ( RSX-11M SYSTEM-CALL EXAMPLE - DATE)
  1 DECIMAL
  2 0 VARIABLE TBUFF 14 ALLOT
  3 CODE TIME TBUFF # SP -) MOV,  2 400 * 75 +  # SP -) MOV,
  4    377 EMT, NEXT, C;
  5 : YEAR ( -> N )  TIME TBUFF @ ;
  6 : MONTH ( -> N )  TIME TBUFF 2+ @ ;
  7 : DAY ( -> N )  TIME TBUFF 4 + @  ;
  8 : HOUR ( -> N )  TIME TBUFF 6 + @  ;
  9 : MINUTE ( -> N )  TIME TBUFF 8 + @  ;
 10 : SECOND ( -> N )  TIME TBUFF 10 + @  ;
 11 : TICK ( -> N )  TIME TBUFF 12 + @  ;
 12 : TICKS/SECOND ( -> N )  TIME TBUFF 14 + @  ;
 13
 14
 15
```

```
SCR # 30
  0 ( RSX-11M SYSTEM-CALL EXAMPLE - TERMINAL I/O)
  1
  2 : PUSH ASSEMBLER SP -) MOV, FORTH ;
  3 0 VARIABLE INBUF 78 ALLOT
  4 0 VARIABLE IOSTAT 2 ALLOT
  5 CODE INPUT 0 # PUSH 0 # PUSH 0 # PUSH 0 # PUSH
  6    120 # PUSH INBUF # PUSH 0 # PUSH IOSTAT # PUSH
  7    4 # PUSH 4 # PUSH 10400 # PUSH 6003 # PUSH
  8    377 EMT, NEXT, C;
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 31
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 32
  0 ( EXAMPLES - RANDOM #S, VIRTUAL ARRAY, RECURSIVE CALL)
  1 ( RANDOM NUMBER GENERATOR.  CAUTION - EVERY 128TH RELATED.)
  2 1001 VARIABLE RSEED
  3 : URAND ( -> N, UNSIGNED 0-65K)
  4    RSEED @ 2725 U* 13947 S->D D+ DROP DUP RSEED ! ;
  5 : RAND ( N -> M, 0 TO N-1)
  6    URAND U* SWAP DROP ;
  7 ( 'VARRAY' CREATES A VIRTUAL ARRAY ON DISK SCREENS.)
  8 : VARRAY ( LRECL #RECS STARTSCREEN -> )
  9    <BUILDS , , DUP , 1024 SWAP / ,
 10    ( STARTSCREEN, #RECS, LRECL, RECS/SCREEN)
 11    DOES> >R DUP R 2 + @ < 0= OVER 0< OR
 12      IF ." ERROR, V-ARRAY RANGE " . R> DROP
 13      ELSE R 6 + @ /MOD R @ + BLOCK SWAP R> 4 + @ * + THEN ;
 14
 15 : MYSELF ( RECURSIVE CALL)   LATEST PFA CFA , ; IMMEDIATE
```

```
SCR # 33
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 34
  0 ( CREATE BOOTABLE IMAGE ON SCREENS 40-47.  FOR STAND-ALONE.)
  1 ( NOTE - THIS DOES NOT WRITE THE BOOT BLOCK OR THE OTHER FORTH)
  2 ( SCREENS.  IF YOU START WITH A BLANK DISK, FIRST USE THE COPY)
  3 ( PROGRAM ON SCREEN 38, AND MOVE THE COPY TO DX0. THEN EXECUTE)
  4 ( 'DECIMAL 34 LOAD'.  THE BOOT LOADER WILL ONLY HANDLE)
  5 ( IMAGES UP TO 7.9K BYTES.  THIS LEAVES SEVERAL HUNDRED)
  6 ( BYTES FOR NEW OPERATIONS, AND THESE COULD LOAD MORE.)
  7 DECIMAL   : SIZETEST 1024 8 * 256 -  HERE U< IF ." TOO BIG"
  8    QUIT THEN ;   SIZETEST   FORGET SIZETEST
  9 OCTAL     ( NEXT LINE RESETS THE START-UP TABLE.)
 10 LATEST 14 +ORIGIN !   HERE 36 +ORIGIN !  HERE 34 +ORIGIN !
 11 DECIMAL   35 LOAD   CREATE-BINARY-IMAGE  ( WRITE SYSTEM)
 12 10 LOAD 11 LOAD 12 LOAD 13 LOAD 14 LOAD 15 LOAD  ( ASSEMBLER)
 13 36 LOAD  ( WRITES BOOT LOADER AT END OF SCREEN 47)
 14 COLD  ( COLD START OF NEW SYSTEM - GET RID OF ASSEMBLER ETC.)
 15
```

```
SCR # 35
  0 ( CREATE A BINARY IMAGE ON SCREENS 40 - 47  )
  1 ( START AT ZERO)
  2 : CREATE-BINARY-IMAGE  48 40 DO
  3    I 40 - 1024 *  ( ADDRESS TO MOVE FROM)
  4    I BLOCK ( ADDRESS TO MOVE TO)
  5    1024 CMOVE     UPDATE    LOOP    FLUSH ;
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 36
  0 ( CREATE BOOT LOADER.  NOTE - DOES NOT WRITE BOOT BLOCK)
  1 ASSEMBLER DEFINITIONS OCTAL
  2 : INIT, 1000 # R0 MOV,   00000 # R1 MOV,
  3    177170 # R4 MOV,   200 # R3 MOV, ;
  4 : ?TERM, R1 () TSTB,   LE IF, 1000 @# JMP, ENDIF, ;
  5 : WAITT, BEGIN, R3 R4 () BIT, NE UNTIL, ;
  6 : WAITD, BEGIN, 40 # R4 () BIT, NE UNTIL, ;
  7 : ?ERR, R4 () TST, LE IF, HALT, ENDIF, ;
  8 : BLOOP, R3 R2 MOV,
  9    BEGIN, WAITT, 2 R4 I) R0 )+ MOVB,   R2 DEC,   EQ UNTIL, ;
 10 : NEXTTAB, 1 R1 I) R5 MOVB,   R5 INC, R5 INC,
 11    R5 32 # CMP, GT IF, 32 # R5 SUB, THEN,
 12    R5 1 R1 I) MOVB,   1 R1 I) 2 R1 I) CMPB,
 13    EQ IF, 3 # R1 ADD, ENDIF,  ;
 14
 15    DECIMAL 37 LOAD
```

```
SCR # 37
  Ø ( CREATE BOOT LOADER, CONT.)       OCTAL
  1 : TRACK, R1 () R5 MOVB, R5 2 R4 I) MOV, ;
  2 : SECTOR, 1 R1 I) R5 MOVB,  R5 2 R4 I) MOV, ;
  3 : MAINL, BEGIN, ?TERM, 7 # R4 () MOV, WAITT, SECTOR, WAITT,
  4    TRACK, WAITD, ?ERR,  3 # R4 () MOV,  BLOOP, NEXTTAB,
  5    4ØØ UNTIL, ;
  6 : 2, 4ØØ * + , ;
  7 : TABLE,   17 27 2,   7 17 2,   1Ø 1Ø 2,
  8    2Ø 15 2,   15 2Ø 2,   16 16 2,
  9    21 23 2,   23 21 2,   24 26 2,   Ø Ø 2,  ;
 1Ø
 11 CODE BOOT 35ØØØ JMP, C;
 12 : TASK ;
 13 35ØØØ DP ! HERE 6 + INIT, WAITD, MAINL, HERE SWAP ! TABLE,
 14 FORGET TASK
 15 17572 35ØØ6 !   35ØØØ 21 26 WTS  352Ø0 21 3Ø WTS
```

```
SCR # 38
  Ø ( DISK COPY FROM SYSTEM DISK TO DX1)
  1 DECIMAL   2ØØØØ CONSTANT C
  2 : GET 26 Ø DO   C I 128 * +   OVER   I 3 * 26 MOD 1+   RTS
  3    LOOP ;
  4 : PUT 26 Ø DO   C I 128 * +   OVER 77 +   I 3 * 26 MOD 1+   WTS
  5    LOOP ;
  6 : COPY 77 Ø DO   I GET DROP   I PUT DROP   LOOP ;
  7
  8
  9
 1Ø
 11
 12
 13
 14
 15
```

```
SCR # 39
  Ø ( ** CAUTION **   BINARY IMAGE IN SCREENS 4Ø-47)
  1 ( SCREENS 4Ø - 47  CONTAIN THE BOOTABLE STAND-ALONE SYSTEM.)
  2 (   THE LAST 256 BYTES OF THIS SYSTEM IMAGE CONTAIN A LOADER.)
  3 ( ADVISE RESERVING SCREENS 48 - 59 FOR BINARY PROGRAM OVERLAYS.)
  4 (   FOR NOW, 5Ø-56 CONTAIN AN EXAMPLE OF FORTH PROGRAMMING FOR)
  5 (   A FLOPPY DRIVER.  THESE SCREENS ARE NO LONGER USED BY THE)
  6 (   SYSTEM.  THEY ARE FOR ILLUSTRATION ONLY, AND MAY BE)
  7 (   DELETED.)
  8
  9
 1Ø
 11
 12
 13
 14
 15
 OK
```

XII. Glossary

The Forth Implementation Team glossary for the common language model is reprinted (from the Installation Manual). Words which are in FORTH.MAC but not in the common language glossary are listed below. Most were added for convenience on this particular system.

=CELLS	n1 --- n1 Add 1 if necessary to make even number.
?ALIGN	--- Force even address in dictionary.
BYE	--- Return to the operating system.
C/L	--- n1 Number of characters per line (F.I.G. model).
CURRENT	--- Vocabulary into which new definitions are compiled (F.I.G. model).
I/O	addr block# flag --- Read or write 512-byte block, handle errors.
OCTAL	--- Set number base to octal.
U.	n1 --- Unsigned print.
U<	n1 n2 --- flag Unsigned less-than test.
XI/O	addr block# flag --- report Read or write 512-byte block, return error report.

This glossary contains all of the word definitions in Release 1 of fig-FORTH. The definitions are presented in the order of their ascii sort.

The first line of each entry shows a symbolic description of the action of the proceedure on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes "---" indicate the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8 bit byte (i.e. hi 8 bits zero)
c	7 bit ascii character (hi 9 bits zero)
d	32 bit signed double integer, most significant portion with sign on top of stack.
f	boolean flag. 0=false, non-zero=true
ff	boolean false flag=0
n	16 bit signed integer number
u	16 bit unsigned integer
tf	boolean true flag=non-zero

The capital letters on the right show definition characteristics:

C	May only be used within a colon definition. A digit indicates number of memory addresses used, if other than one.
E	Intended for execution only.
L0	Level Zero definition of FORTH-78
L1	Level One definition of FORTH-78
P	Has precedence bit set. Will execute even when compiling.
U	A user variable.

Unless otherwise noted, all references to numbers are for 16 bit signed integers. On 8 bit data bus computers, the high byte of a number is on top of the stack, with the sign in the leftmost bit. For 32 bit signed double numbers, the most significant part (with the sign) is on top.

All arithemetic is implicitly 16 bit signed integer math, with error and under-flow indication unspecified.

! n addr --- L0
Store 16 bits of n at address. Pronounced "store".

!CSP
Save the stack position in CSP. Used as part of the compiler security.

d1 --- d2 L0
Generate from a double number d1, the next ascii character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #>. See #S.

#> d --- addr count L0
Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE.

#S d1 --- d2 L0
Generates ascii text in the text output buffer, by the use of #, until a zero double number n2 results. Used between <# and #>.

' --- addr P,L0
Used in the form:
' nnnn
Leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes in a colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

(P,L0
Used in the form:
(cccc)
Ignore a comment that will be delimited by a right parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

(.") C+
The run-time proceedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

(;CODE) C
The run-time proceedure, compiled by ;CODE, that rewrites the code field of the most recently defined word to point to the following machine code sequence. See ;CODE.

(+LOOP) n --- C2
The run-time proceedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(ABORT)
Executes after an error when WARNING is -1. This word normally executes ABORT, but may be altered (with care) to a user's alternative proceedure.

(DO) C
The run-time proceedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2 --- pfa b tf (ok)
addr1 addr2 --- ff (bad)
Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(LINE) n1 n2 --- addr count
Convert the line number n1 and the screen n2 to the disc buffer address containing the data. A count of 64 indicates the full line text length.

(LOOP) C2
The run-time proceedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP.

(NUMBER) d1 addr1 --- d2 addr2
Convert the ascii text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. Addr2 is the address of the first unconvertable digit. Used by NUMBER.

* n1 n2 --- prod L0
Leave the signed product of two signed numbers.

*/ n1 n2 n3 --- n4 L0
Leave the ratio n4 = n1*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence:
n1 n2 * n3 /

*/MOD n1 n2 n3 --- n4 n5 L0
Leave the quotient n5 and remainder n4 of the operation n1*n2/n3
A 31 bit intermediate product is used as for */.

+ n1 n2 --- sum L0
Leave the sum of n1+n2.

+! n addr --- L0
Add n to the value at the address. Pronounced "plus-store".

+- n1 n2 --- n3
Apply the sign of n2 to n1, which is left as n3.

+BUF add1 --- addr2 f
Advance the disc buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

+LOOP n1 --- (run)
addr n2 --- (compile) P,C2,L0
Used in a colon-definition in the form:
DO ... n1 +LOOP
At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0), or until the new index is equal to or less than the limit (n1<0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN n --- addr
Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

, n --- L0
Store n into the next available dictionary memory cell, advancing the dictionary pointer. (comma)

- n1 n2 --- diff L0
Leave the difference of n1-n2.

--> P,L0
Continue interpretation with the next disc screen. (pronounced next-screen).

-DUP n1 -- n1 (if zero)
n1 -- n1 n1 (non-zero) L0
Reproduce n1 only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop it.

-FIND --- pfa b tf (found)
--- ff (not found)
Accepts the next text word (delimited by blanks) in the input strean to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left.

-TRAILING addr n1 --- addr n2
Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. i.e. the characters at addr+n1 to addr+n2 are blanks.

. n --- L0
Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blanks follows. Pronounced "dot".

." P,L0
Used in the form:
." cccc"
Compiles an in-line string cccc (delimited by the trailing ") with an execution proceedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final ". The maximum number of characters may be an installation dependent value. See (.").

.LINE line scr ---
Print on the terminal device, a line of text from the disc by its line and screen number. Trailing blanks are suppressed.

.R n1 n2 ---
Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

/ n1 n2 --- quot L0
Leave the signed quotient of n1/n2.

/MOD n1 n2 --- rem quot L0
Leave the remainder and signed quotient of n1/n2. The remainder has the sign of the dividend.

0 1 2 3 --- n

These small numbers are used so often that is is attractive to define them by name in the dictionary as constants.

0< n --- f L0

Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= n --- f L0

Leave a true flag is the number is equal to zero, otherwise leave a false flag.

OBRANCH f --- C2

The run-time proceedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+ n1 --- n2 L1

Increment n1 by 1.

2+ n1 --- n2

Leave n1 incremented by 2.

: P,E,L0

Used in the form called a colon-definition:

: cccc ... ;

Creates a dictionary entry defining cccc as equivalent to the following sequence of Forth word definitions '...' until the next ';' or ';CODE'. The compiling process is done by the text interpreter as long as STATE is non-zero. Other details are that the CONTEXT vocabulary is set to the CURRENT vocabulary and that words with the precedence bit set (P) are executed rather than being compiled.

; P,C,L0

Terminate a colon-definition and stop further compilation. Compiles the run-time ;S.

;CODE P,C,L0

Used in the form:

: cccc .... ;CODE
assembly mnemonics

Stop compilation and terminate a new defining word cccc by compiling (;CODE). Set the CONTEXT vocabulary to ASSEMBER, assembling to machine code the following mnemonics.

When cccc later executes in the form:

cccc nnnn

the word nnnn will be created with its execution proceedure given by by the machine code following cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn. An existing defining word must exist in cccc prior to ;CODE.

;S P,L0

Stop interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling proceedure.

< n1 n2 --- f L0

Leave a true flag if n1 is less than n2; otherwise leave a false flag.

<# L0

Setup for pictured numeric output formatting using the words:

<# # #S SIGN #>

The conversion is done on a double number producing text at PAD.

<BUILDS C,L0

Used within a colon-definition:

: cccc <BUILDS ...
DOES> ... ;

Each time cccc is executed, <BUILDS defines a new word with a high-level execution proceedure. Executing cccc in the form:

cccc nnnn

uses <BUILDS to create a dictionary entry for nnnn with a call to the DOES> part for nnnn. When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after DOES> in cccc. <BUILDS and DOES> allow run-time proceedures to written in high-level rather than in assembler code (as required by ;CODE).

= n1 n2 --- f L0

Leave a true flag if n1=n2; otherwise leave a false flag.

> n1 n2 --- f L0

Leave a true flag if n1 is greater than n2; otherwise a false flag.

>R n --- C,L0

Remove a number from the computation stack and place as the most accessable on the return stack. Use should be balanced with R> in the same definition.

? addr -- L0

Print the value contained at the address in free format according to the current base.

?COMP

Issue error message if not compiling.

?CSP

Issue error message if stack position differs from value saved in CSP.

?ERROR f n ---
Issue an error message number n, if the boolean flag is true.

?EXEC
Issue an error message if not executing.

?LOADING
Issue an error message if not loading

?PAIRS n1 n2 ---
Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK
Issue an error message is the stack is out of bounds. This definition may be installation dependent.

?TERMINAL --- f
Perform a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation. This definition is installation dependent.

@ addr --- n L0
Leave the 16 bit contents of address.

ABORT L0
Clear the stacks and enter the execution state. Return control to the operators terminal, printing a message appropriate to the installation.

ABS n --- u L0
Leave the absolute value of n as u.

AGAIN addr n --- (compiling) P,C2,L0
Used in a colon-definion in the form:
BEGIN ... AGAIN
At run-time, AGAIN forces execution to return to corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below).

At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

ALLOT n --- L0
Add the signed number to the dictionary pointer DP. May be used to reserve dictionary space or re-origin memory. n is with regard to computer address type (byte or word).

AND n1 n2 --- n2 L0
Leave the bitwise logical and of n1 and n2 as n3.

B/BUF --- n
This constant leaves the number of bytes per disc buffer, the byte count read from disc by BLOCK.

B/SCR --- n
This constant leaves the number of blocks per editing screen. By convention, an editing screen is 1024 bytes organized as 16 lines of 64 characters each.

BACK addr ---
Calculate the backward branch offset from HERE to addr and compile into the next available dictionary memory address.

BASE --- addr U,L0
A user variable contaning the current number base used for input and output conversion.

BEGIN --- addr n (compiling) P,L0
Occurs in a colon-definition in form:
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT
At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the correspoinding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.

At compile time BEGIN leaves its return address and n for compiler error checking.

BL --- c
A constant that leaves the ascii value for "blank".

BLANKS addr count ---
Fill an area of memory begining at addr with blanks.

BLK --- addr U,L0
A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLOCK n --- addr L0
Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disc to which ever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is rewritten to disc before block n is read into the buffer. See also BUFFER, R/W UPDATE FLUSH

BLOCK-READ
BLOCK-WRITE These are the preferred names for the installation dependent code to read and write one block to the disc.

BRANCH C2,LO
The run-time proceedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT.

BUFFER n --- addr
Obtain the next memory buffer, assigning it to block n. If the contents of the buffer is marked as updated, it is written to the disc The block is not read from the disc. The address left is the first cell within the buffer for data storage.

C! b addr ---
Store 8 bits at address. On word addressing computers, further specification is necessary regarding byte addressing.

C, b ---
Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer. This is only available on byte addressing computers, and should be used with caution on byte addressing minicomputers.

C@ addr --- b
Leave the 8 bit contents of memory address. On word addressing computers, further specification is needed regarding byte addressing.

CFA pfa --- cfa
Convert the parameter field address of a definition to its code field address.

CMOVE from to count ---
Move the specified quantity of bytes beginning at address from to address to. The contents of address from is moved first proceeding toward high memory. Further specification is necessary on word addressing computers.

COLD
The cold start proceedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.

COMPILE C2
When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in additon to simply compling an execution address (which the interpreter already does).

CONSTANT n --- LO
A defining word used in the form:
n CONSTANT cccc
to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n to the stack.

CONTEXT --- addr U,LO
A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr1 --- addr2 n LO
Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically COUNT is followed by TYPE.

CR LO
Transmit a carriage return and line feed to the selected output device.

CREATE
A defining word used in the form:
CREATE cccc
by such words as CODE and CONSTANT to create a dictionary header for a Forth definition. The code field contains the address of the words parameter field. The new word is created in the CURRENT vocablary.

CSP ---- addr U
A user variable temporarily storing the stack pointer position, for compilation error checking.

D+ d1 d2 --- dsum
Leave the double number sum of two double numbers.

D+- d1 n --- d2
Apply the sign of n to the double number d1, leaving it as d2.

D. d --- L1
Print a signed double number from a 32 bit two's complement value. The high-order 16 bits are most accessable on the stack. Conversion is performed according to the current BASE. A blank follows. Pronounced D-dot.

D.R d n ---
Print a signed double number d right aligned in a field n characters wide.

DABS d --- ud
Leave the absolute value ud of a double number.

DECIMAL L0
Set the numeric conversion BASE for decimal input-output.

DEFINITIONS L1
Used in the form:
cccc DEFINITIONS
Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name cccc made it the CONTEXT vocabulary and executing DEFINITIONS made both specify vocabulary cccc.

DIGIT c n1 --- n2 tf (ok)
c n1 --- ff (bad)
Converts the ascii character c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, leaves only a false flag.

DLIST
List the names of the dictionary entries in the CONTEXT vocabulary.

DLITERAL d --- d (executing)
d --- (compiling) P
If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

DMINUS d1 --- d2
Convert d1 to its double number two's complement.

DO n1 n2 --- (execute)
addr n --- (compile) P,C2,L0
Occurs in a colon-definition in form:
DO ... LOOP
DO ... +LOOP
At run time, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after DO; otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at run-time and may be the result of other operations. Within a loop 'I' will copy the

urrent value of the index to the stack. See I, LOOP, +LOOP, LEAVE.

When compiling within the colon-definition, DO compiles (DO), leaves the following address addr and n for later error checking.

DOES> L0
A word which defines the run-time action within a high-level defining word. DOES> alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following DOES>. Used in combination with <BUILDS. When the DOES> part executes it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the Forth assembler, multi-diminsional arrays, and compiler generation.

DP ---- addr U,L
A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT.

DPL ---- addr U,L0
A user variable containing the number of digits to the right of the decimal on double integer input. It may also be used hold output column location of a decimal point, in user generated formating. The default value on single number input is -1.

DR0
DR1 Installation dependent commands to select disc drives, by preseting OFFSET. The contents of OFFSET is added to the block number in BLOCK to allow for this selection. Offset is supressed for error text so that is may always originate from drive 0.

DROP n --- L0
Drop the number from the stack.

DUMP addr n --- L0
Print the contents of n memory locations beginning at addr. Both addresses and contents are shown in the current numeric base.

DUP n --- n n L0
Duplicate the value on the stack.

ELSE addr1 n1 --- addr2 n2
(compiling) P,C2,L0
Occurs within a colon-definition in the form:
IF ... ELSE ... ENDIF
At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. It has no stack effect.

At compile-time ELSE emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch from IF by calculating the offset from addr1 to HERE and storing at addr1.

EMIT c --- L0
Transmit ascii character c to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS L0
Mark all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disc. This is also an initialization proceedure before first use of the disc.

ENCLOSE addr1 c ---
ddr1 n1 n2 n3
The text scanning primitive used by WORD. From the text address addr1 and an ascii delimiting character c, is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This proceedure will not process past an ascii 'null', treating it as an unconditional delimiter.

END P,C2,L0
This is an 'alias' or duplicate definition for UNTIL.

ENDIF addr n --- (compile) P,C0,L0
Occurs in a colon-definition in form:
IF ... ENDIF
IF ... ELSE ... ENDIF
At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported in fig-FORTH. See also IF and ELSE.

At compile-time, ENDIF computes the forward branch offset from addr to HERE and stores it at addr. n is used for error tests.

ERASE addr n ---
Clear a region of memory to zero from addr over n addresses.

ERROR line --- in blk
Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 4 of drive 0 is printed. This line number may be positive or negative, and beyond just screen 4. If WARNING=0, n is just printed as a message number (non disc installation). If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT). fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

EXECUTE addr --
Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

EXPECT addr count --- L0
Transfer characters from the terminal to address, until a "return" or the count of characters have been received. One or more nulls are added at the end of the text.

FENCE --- addr U
A user variable containing an address below which FORGETting is trapped. To forget below this point the user must alter the contents of FENCE.

FILL addr quan b ---
Fill memory at the address with the specified quantity of bytes b.

FIRST --- n
A constant that leaves the address of the first (lowest) block buffer.

FLD --- addr U

A user variable for control of number output field width. Presently unused in fig-FORTH.

FORGET E,L0

Executed in the form:
FORGET cccc
Deletes definition named cccc from the dictionary with all entries physically following it. In fig-FORTH, an error message will occur if the CURRENT and CONTEXT vocabularies are not currently the same.

FORTH P,L1

The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile time.

HERE --- addr L0

Leave the address of the next available dictionary location.

HEX L0

Set the numeric conversion base to sixteen (hexadecimal).

HLD --- addr L0

A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c --- L0

Used between <# and #> to insert an ascii character into a pictured numeric output string.
e.g. 2E HOLD will place a decimal point.

I --- n C,L0

Used within a DO-LOOP to copy the loop index to the stack. Other use is implementation dependent.
See R.

ID. addr ---

Print a definition's name from its name field address.

IF f --- (run-time)
--- addr n (compile) P,C2,L0

Occurs is a colon-definition in form:
IF (tp) ... ENDIF
IF (tp) ... ELSE (fp) ... ENDIF
At run-time, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead thru the true part. If f is false (zero), execution skips till just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional.; if missing, false execution skips to just after ENDIF.

At compile-time IF compiles OBRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset and error testing.

IMMEDIATE

Mark the most resently made definition so that when encountered at compile time, it will be executed rather than being compiled. i.e. the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceeding it with [COMPILE].

IN --- addr L0

A user variable containing the byte offset within the current input text buffer (terminal or disc) from which the next text will be accepted. WORD uses and moves the value of IN.

INDEX from to ---

Print the first line of each screen over the range from, to. This is used to view the comment lines of an area of text on disc screens.

INTERPRET

The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or disc) depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT it is converted to a number according to the current base. That also failing, an error message echoing the name with a " ?" will be given. Text input will be taken according to the convention for WORD. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action.
See NUMBER.

KEY --- c L0
Leave the ascii value of the next terminal key struck.

LATEST --- addr
Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE C,L0
Force termination of a DO-LOOP at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution prodeeds normally until LOOP or +LOOP is encountered.

LFA pfa --- lfa
Convert the parameter field address of a dictionary definition to its link field address.

LIMIT ---- n
A constant leaving the address just above the highest memory available for a disc buffer. Usually this is the highest system memory.

LIST n --- L0
Display the ascii text of screen n on the selected output device. SCR contains the screen number during and after this process.

LIT --- n C2,L0
Within a colon-definition, LIT is automatically compiled before each 16 bit literal number encountered in input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.

LITERAL n --- (compiling) P,C2,L0
If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

```
: xxx  [ calculate ] LITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is reusumed and LITERAL compiles this value.

LOAD n --- L0
Begin interpretation of screen n. Loading will terminate at the end of the screen or at ;S. See ;S and -->.

LOOP addr n --- (compiling) P,C2,L0
Occurs in a colon-definition in form:

```
DO  ...  LOOP
```

At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

At compile-time, LOOP compiles (LOOP) and uses addr to calculate an offset to DO. n is used for error testing.

M* n1 n2 --- d
A mixed magnitude math operation which leaves the double number signed product of two signed number.

M/ d n1 --- n2 n3
A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

M/MOD ud1 u2 --- u3 ud4
An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX n1 n2 --- max L0
Leave the greater of two numbers.

MESSAGE n ---
Print on the selected output device the text of line n relative to screen 4 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc un-available).

MIN n1 n2 --- min L0
Leave the smaller of two numbers.

MINUS n1 --- n2 L0
Leave the two's complement of a number.

MOD n1 n2 --- mod L0
Leave the remainder of n1/n2, with the same sign as n1.

MON
Exit to the system monitor, leaving a re-entry to Forth, if possible.

MOVE addr1 addr2 n ---
Move the contents of n memory cells (16 bit contents) beginning at addr1 into n cells beginning at addr2. The contents of addr1 is moved first. This definition is appropriate on on word addressing computers.

NEXT
This is the inner interpreter that uses the interpretive pointer IP to execute compiled Forth definitions. It is not directly executed but is the return point for all code proceedures. It acts by fetching the address pointed by IP, storing this value in register W. It then jumps to the address pointed to by the address pointed to by W. W points to the code field of a definition which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of Forth. Locations of IP and W are computer specific.

NFA pfa --- nfa
Convert the parameter field address of a definition to its name field.

NUMBER addr --- d
Convert a character string left at addr with a preceeding count, to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

OFFSET --- addr U
A user variable which may contain a block offset to disc drives. The contents of OFFSET is added to the stack number by BLOCK. Messages by MESSAGE are independent of OFFSET. See BLOCK, DR0, DR1, MESSAGE.

OR n1 n2 -- or L0
Leave the bit-wise logical or of two 16 bit values.

OUT --- addr U
A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formating.

OVER n1 n2 --- n1 n2 n1 L0
Copy the second stack value, placing it as the new top.

PAD --- addr L0
Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA nfa --- pfa
Convert the name field address of a compiled definition to its parameter field address.

POP
The code sequence to remove a stack value and return to NEXT. POP is not directly executable, but is a Forth re-entry point after machine code.

PREV ---- addr
A variable containing the address of the disc buffer most recently referenced. The UPDATE command marks this buffer to be later written to disc.

PUSH
This code sequence pushes machine registers to the computation stack and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code.

PUT
This code sequence stores machine register contents over the topmost computation stack value and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code.

QUERY
Input 80 characters of text (or until a "return") from the operators terminal. Text is positioned at the address contained in TIB with IN set to zero.

QUIT L1
Clear the return stack, stop compilation, and return control to the operators terminal. No message is given.

R --- n
Copy the top of the return stack to the computation stack.

R# --- addr U
A user variable which may contain the location of an editing cursor, or other file related function.

R/W addr blk f ---
The fig-FORTH standard disc read-write linkage. addr specifies the source or destination block buffer, blk is the sequential number of the referenced block; and f is a flag for f=0 write and f=1 read. R/W determines the location on mass storage, performs the read-write and performs any error checking.

R> --- n L0
Remove the top value from the return stack and leave it on the computation stack. See >R and R.

R0 --- addr U
A user variable containing the initial location of the return stack. Pronounced R-zero. See RP!

REPEAT addr n --- (compiling) P,C2
Used within a colon-definition in the form:
BEGIN ... WHILE ... REPEAT
At run-time, REPEAT forces an unconditional branch back to just after the correspoinding BEGIN.

At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error testing.

ROT n1 n2 n3 --- n2 n3 n1 L0
Rotate the top three values on the stack, bringing the third to the top.

RP!
A computer dependent proceedure to initialize the return stack pointer from user variable R0.

S->D n --- d
Sign extend a single number to form a double number.

S0 --- addr U
A user variable that contains the initial value for the stack pointer. Pronounced S-zero. See SP!

SCR --- addr U
A user variable containing the screen number most recently reference by LIST.

SIGN n d --- d L0
Stores an ascii "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must be used between <# and #>.

SMUDGE
Used during word definition to toggle the "smudge bit" in a definitions' name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SP!
A computer dependent proceedure to initialize the stack pointer from S0.

SP@ --- addr
A computer dependent proceedure to return the address of the stack position to the top of the stack, as it was before SP@ was executed. (e.g. 1 2 SP@ @ . . . would type 2 2 1)

SPACE L0
Transmit an ascii blank to the output device.

SPACES n --- L0
Transmit n ascii blanks to the output device.

STATE --- addr L0,U
A user variable containg the compilation state. A non-zero value indicates compilation. The value itself may be implementation dependent.

SWAP n1 n2 --- n2 n1 L0
Exchange the top two values on the stack.

TASK
A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety.

THEN P,C0,L0
An alias for ENDIF.

TIB --- addr U
A user variable containing the address of the terminal input buffer.

TOGGLE addr b ---
Complement the contents of addr by the bit pattern b.

TRAVERSE addr1 n --- addr2
Move across the name field of a fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward hi memory; if n=-1, the motion is toward low memory. The addr2 resulting is address of the other end of the name.

TRIAD scr ---
Display on the selected output device the three screens which include that numbered scr, begining with a screen evenly divisible by three. Output is suitable for source text records, and includes a reference line at the bottom taken from line 15 of screen4.

TYPE addr count --- LO
Transmit count characters from addr to the selected output device.

U* u1 u2 --- ud
Leave the unsigned double number product of two unsigned numbers.

U/ ud u1 --- u2 u3
Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

UNTIL f --- (run-time)
addr n --- (compile) P,C2,LO
Occurs within a colon-definition in the form:
BEGIN ... UNTIL
At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; if true, execution continues ahead.

At compile-time, UNTIL compiles (OBRANCH) and an offset from HERE to addr. n is used for error tests.

UPDATE LO
Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be transferred automatically to disc should its buffer be required for storage of a different block.

USE --- addr
A variable containing the address of the block buffer to use next, as the least recently written.

USER n --- LO
A defining word used in the form:
n USER cccc
which creates a user variable cccc. The parameter field of cccc contains n as a fixed offset relative to the user pointer register UP for this user variable. When cccc is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable.

VARIABLE E,LO
A defining word used in the form:
n VARIABLE cccc
When VARIABLE is executed, it creates the definition cccc with its parameter field initialized to n. When cccc is later executed, the address of its parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

VOC-LINK --- addr U
A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting thru multiple vocabularys.

VOCABULARY E,L
A defining word used in the form:
VOCABULARY cccc
to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary into which new definitions are placed.

In fig-FORTH, cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularys ulitmately chain to Forth. By convention, vocabulary names are to be declared IMMEDIATE. See VOC-LINK.

VLIST
List the names of the definitions in the context vocabulary. "Break" will terminate the listing.

WARNING --- addr U
A user variable containing a value controlling messages. If = 1 disc is present, and screen 4 of drive 0 is the base location for messages. If = 0, no disc is present and messages will be presented by number. If = -1, execute (ABORT) for a user specified proceedure. See MESSAGE, ERROR.

WHILE f --- (run-time)
adl nl --- adl nl ad2 n2 P,C2
Occurs in a colon-definition in the form:
BEGIN ... WHILE (tp) ... REPEAT
At run-time, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE contintues execution of the true part thru to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile time, WHILE emplaces (OBRANCH) and leaves ad2 of the reserved offset. The stack values will be resolved by REPEAT.

WIDTH --- addr U

In fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definitions' name. It must be 1 thru 31, with a default value of 31. The name character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.

WORD c --- L0

Read the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string begining at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurances of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disc block stored in BLK. See BLK, IN.

X

This is pseudonym for the "null" or dictionary entry for a name of one character of ascii null. It is the execution proceedure to terminate interpretation of a line of text from the terminal or within a disc buffer, as both buffers always have a null at the end.

XOR n1 n2 --- xor L1

Leave the bitwise logical exclusive-or of two values.

[P,L1

Used in a colon-definition in form:

```
: xxx [ words ] more ;
```

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]. See LITERAL,].

[COMPILE] P,C

Used in a colon-definition in form:

```
: xxx [COMPILE] FORTH ;
```

[COMPILE] will force the compilation of an immediate defininition, that would otherwise execute during compilation. The above example will select the FORTH vocabulary when xxx executes, rather than at compile time.

] L1

Resume compilation, to the completion of a colon-definition. See [.

XIII. ERROR AND WARNING MESSAGES

Messages are listed alphabetically below. The following conditions do not fit in the alphabetical list:

word ? The word is not defined. This will also occur if a number contains a digit not valid in the current number base, e.g. '8' in octal. Also note that editor operations are undefined unless a screen is being edited, and assembly mnemonics, etc. are undefined outside of a 'CODE' definition. Also the editor, assembler, and string package must have been loaded from disk before they can be used.

no response If there is no response from the system, first hit carriage return to see if the system responds 'OK'. If not, try semicolon and carriage return, in case the system was in compilation mode.

MSG # n Most error conditions in the Forth Interest Group model can report as either message numbers or texts. This scheme allows the message texts to be kept on disk to save memory, yet if the disk file (FORTH.DAT) is not accessible (e.g. on systems with no disk), Forth can still run and it will report errors as numbers. The system variable 'WARNING' controls this error-reporting mode; commonly the disk load

screen changes 'WARNING' to cause message texts to be output. Note that when numbers are output, they will print in whatever number base is in effect.

Error Numbers

These numbers are reported as 'MSG # n', only if the warning mode has not been set to use message texts from disk. The warning mode is set for disk when you execute '1 LOAD'. If you get one of these message numbers, look up the corresponding text in the alphabetical error and warning list below, for additional information.

MSG #	TEXT
1	EMPTY STACK
2	DICTIONARY FULL
4	ISN'T UNIQUE (This is a warning only - no problem)
17	COMPILATION ONLY, USE IN DEFINITION
18	EXECUTION ONLY
19	CONDITIONALS NOT PAIRED
20	DEFINITION NOT FINISHED
21	IN PROTECTED DICTIONARY
22	USE ONLY WHEN LOADING

These message numbers are in decimal. If the system is in octal when the message is given, the corresponding octal numbers will be printed.

Error/Warning List

Note that some conditions which do not fit well into this list are described separately above.

$STACK EMPTY
: String stack empty.

CAN'T SPREAD
: Invalid argument to Editor 'SPREAD' command.

CODE ERROR, STACK DEPTH CHANGED
: Error in using the Assembler, in a CODE definition. Usually an operand, mode symbol, or instruction has been omitted, or is extraneous. In case of difficulty spotting the error, enter a bunch of test CODE definitions, each with only one or a small number of instructions from the erroneous definition.

COMPILATION ONLY, USE IN DEFINITION
: A word such as 'DO', which can only be used inside of a colon definition, has been used otherwise.

CONDITIONALS NOT PAIRED
: 'IF...ENDIF' or other conditionals are not paired or nested correctly.

DECLARE VOCABULARY
: Attempt to FORGET when the CONTEXT and CURRENT vocabularies are not the same. Beware of 'FORGET'; the entire range of words forgotten must be in a single vocabulary, otherwise Forth may crash. 'FORGET' is seldom useful on this system anyway, as there is enough memory so that it is unnecessary to clean up the dictionary very often, and if it is necessary, it is easy to reload the system. Valuable definitions should be in source form on disk.

DEFINITION NOT FINISHED
: Erroneous structure, such as 'DO' with no corresponding 'LOOP' when semicolon ends the definition.

DISK READ ERROR # n
: Disk access error; for some reason the program is not able to read 'FORTH.DAT'. Make sure that the file is present on the proper disk and/or in the proper account number. These error numbers vary

depending on the operating system. In case of difficulty, the source listing pages 9-1 through 9-5 pinpoints each error number in the I/O process. (Note than under RSX-11M, in case Forth crashes after 'FORTH.DAT' has been accessed, that file will probably need to be unlocked.)

DISK WRITE ERROR # n
See DISK READ ERROR, above.

EMPTY STACK
Attempt to use data from the stack when it is empty. However, this condition is not checked every time, for efficiency reasons. It is checked whenever control returns to the keyboard. Also, when 'EMPTY STACK' is given, a couple of numbers are placed on the stack; these are for use in future debugging packages. Therefore 'EMPTY STACK' will not occur every time.

ERR 0 ARG
Attempt to EDIT screen zero (usually the argument of 'EDIT' was omitted). Screen numbers start at 1.

ERR-BR+ n
Attempt to branch beyond range of the 'BR' instruction, in a CODE definition. 'n' is the length of the attempted branch.

ERR-BR- n
Same, only attempt to exceed branch range in negative direction.

ERR-REG-B
Non-register argument to 'JSR,' or 'XOR,'.

ERROR, NO SECOND STRING
Attempt to use '$OVER' or some other string operation which requires two arguments, when there is only one argument on the string stack.

EXECUTION ONLY
Attempt to use a colon within a colon definition.

IN PROTECTED DICTIONARY
Attempt to FORGET a word below the FENCE, which is a safety guard to prevent accidental forgetting of the Forth system. To FORGET below the FENCE, first put a lower limit address into the system variable 'FENCE'.

ISN'T UNIQUE

This is a warning only. It means that the word just defined already exists. The consequence is that the previous definition becomes no longer accessible, which is all right if you do not intend to use the original definition later in the same program. In case of a mistaken redefinition, FORGET the mistake and re-enter the definition with a different name.

NOT A REGISTER n

In the assembler, an address-mode symbol which required a register argument was given a number 'n' which was not zero through seven.

TRAP ERROR # n addr psw

A run-time error occurred and trapped to the operating system. The trap number (which varies with operating system), the address of the trap, and the computer's PSW are given in the current number base. Common causes of traps are non-existent or otherwise invalid addresses.

USE ONLY WHEN LOADING

The operation '--> ' (which is not used in this system anyway) was executed from the keyboard, not from a screen being loaded from disk.

WOULD CAUSE $OVERFLOW

A string operation could not be performed because the string stack would have overflowed into the dictionary, or come dangerously close.

Usually the session can continue normally after one of these errors has been reported. Previous definitions should still be good. Certain errors in use of the assembler may leave the number base set to octal; 'DECIMAL' will restore it.