DECUS

PROGRAM LIBRARY

DECUS NO.	FOCAL8-301 & 12-216
TITLE	U/W FOCAL
AUTHOR	Jim van Zee
COMPANY	University of Washington
DATE	August 1978
SOURCE LANGUAGE	PAL-8/PAL-III

FOCAL8-301
12-216

U/W FOCAL

Vefsion: 1A (August 1978)

Author: Jim van Zee
University of Washington, Seattle, WA

Operating System: OS/8; OS/12

Source Language: PAL-8/PAL-III

Memory Required: 8K

U/W FOCAL is an expanded version PS/8 FOCAL which offers 13 new commands (including 2 unused ones), 15 more function entries (30 altogetner), and many other improvements, all in the same amount of core space! Among the new features are FOCAL Statement Functions, double subscipting, variable file names, decrementing loops, the constant PI, new EAE routines for the 8/E (and older machines too), several improved functions, a command for printing the date and a way to use the teletype as a giant switch register. This version of FOCAL offers exceptional flexibility for laboratory applications as well as greatly enhanced performance for purely numerical problems. 10-digit precision (a unique feature of FOCAL) is standard.

Restrictions: 1-page I/O Handlers

Note: The date fix (see write-up addendum) is required only for the paper tape version.

DECUS

TABLE OF CONTENTS

COMMANDS (p.19)

ELEMENTARY COMMANDS

ASK
BREAK
COMMENT
DO
ERASE
FOR
GO
HESITATE
IF
JUMP
KONTROL
MODIFY
MOVE
NEXT
ON
PLOT
QUIT
RETURN
SET
TYPE
USER
VIEW
WRITE
XECUTE
ZERO
and others

LIBRARY COMMANDS (p.31)

LIBRARY CALL
LIBRARY DELETE
LIBRARY GOSUB
LIBRARY RUN
LIBRARY SAVE
LIBRARY NAME
LIBRARY LIST
LIST ALL
LIST ONLY
LOGICAL BRANCH
LOGICAL EXIT

OPERATORS (p.4)

ARITHMETIC OPERATORS

^	EXPONENTIATION
*	MULTIPLICATION
/	DIVISION
+	ADDITION
-	SUBTRACTION
()	PARENTHESES
[]	SQUARE BRACKETS
<>	ANGLE BRACKETS

SPECIAL CHARACTERS (6)

!	RETURN/LINEFEED
#	CARRIAGE RETURN
:	TABULATE
%	FORMAT CONTROL
$	SYMBOL TABLE
"	QUOTATION MARKS
?	TRACE
E	POWER OF TEN
,	COMMA
;	SEMICOLON
_	DELETE INPUT
	RUBOUT KEY
	LINE FEED KEY
	ALTMODE KEY
	ESCAPE KEY
	RETURN KEY
	SPACE KEY
	CTRL/C
	CTRL/F
	CTRL/G [BELL]
	CTRL/L
	CTRL/Z

GRAPHICS (p.40)

FUNCTIONS (p.52)

FABS( )	ABSOLUTE VALUE
FADC( )	ANALOG - DIGITAL
FATN( )	ARCTANGENT
FBLK( )	STARTING BLOCK
FCOM( )	STORAGE
FCOS( )	COSINE
FDAY( )	CHANGE DATE
FEXP( )	EXPONENTIAL
FIN( )	CHARACTER INPUT
FIND( )	CHARACTER SEARCH
FITR( )	INTEGER PART
FLEN( )	FILE LENGTH
FLOG( )	LOG (BASE E)
FMQ( )	DISPLAY IN MQ
FOUT( )	CHARACTER OUTPUT
FRA( )	RANDOM ACCESS
FRAC( )	FRACTIONAL PART
FRAN( )	RANDOM NUMBER
FSGN( )	SIGN PART
FSIN( )	SINE (RADIANS)
FSQT( )	SQUARE ROOT
FSR( )	READ SWITCH REG.
FTRM( )	INPUT TERMINATOR

and others

USER FUNCTIONS (p.68)

FILE COMMANDS (p.36)

OPEN INPUT
OPEN OUTPUT
OPEN RESTORE INPUT
OPEN RESTORE OUTPUT
OPEN RESTART READ
OUTPUT ABORT
OUTPUT BUFFER
OUTPUT CLOSE
OUTPUT DATE
ONLY LIST

index at the end of the manual (p.92)

Changes from OMSI PS/8 FOCAL and FOCAL-69, FOCAL8 include:

1) Extended library features with device-independent chaining and subroutine calls between programs.

2) File reading and writing commands, 10 digit precision, expandable to 32k memory, 36 possible functions, 26 possible command letters.

3) Computed line numbers and unlimited line lengths.

4) Tabulation on output, format control for scientific notation.

5) Double subscripting allowed.

6) Negative exponentiation operators permitted.

7) FLOG,FEXP,FATN,FSIN,FCOS,FITR,FSQT rewritten for 10-digit accuracy.

8) Character manipulations handled with 'FIN()','FOUT()' and 'FIND()'.

9) FSGN(0)=0 [FSGN(0)=1 in FOCAL,1969]
FOUT(A)=0 [FOUT(A)=A in PS/8 FOCAL]

10) Plot functions or commands available for PDP-12 scope, Tektronix terminals (including joystick), incremental plotters and LAB 8/e.

11) 6 Special variables are protected from the 'ZERO' command. --- 'PI', '!', '"', '$', '%' and '#'. PI is initialized as 3.141592654

12) The number of variables is 120(8k), 207(12k) and 676(16k).

13) Text buffer expanded to 15 blocks on 12k or larger systems.

14) Two-page handlers permitted with 12k (and larger) systems.

15) Program and file names are wholly programmable; file size may be specified. OS/8 block numbers may be used in place of file names.

16) 'Open' and 'Delete' commands can have programmed error returns.

17) Improved distribution and random initialization of 'FRAN()'.

18) 'ERASE', 'MODIFY' and 'MOVE' do not erase variables.

19) 'WRITE' doesn't terminate a line; 'MODIFY' doesn't echo form feed.

20) U/W-FOCAL's starting address is 100 (Field 0) or 10200 (Field 1).

---> <u>CTRL/F IS THE BREAK CHARACTER</u> <----

note: U/W-FOCAL data files are compatible with EDIT and TECO8; however, U/W-FOCAL program files are saved as core images.

Program changes required by U/W-FOCAL:

Programs written in FOCAL,1969 should require only two changes to run under U/W-FOCAL

1) the input device is switched to the high speed paper tape reader with the 'OPEN INPUT PTR:' command and switched back to the terminal with 'OPEN INPUT,ECHO' instead of with the '*' command of FOCAL, 1969.
2) the 'ERASE' command must be changed to 'ZERO'.

Programs written in OMSI PS/8 FOCAL can be used after completing the following simple procedure, which converts the core image files into ASCII data files:

```
1) load PS/8 FOCAL
2) LIBRARY CALL DEVICE:FILENAME
3) OPEN OUTPUT DEVICE:FILENAME;WRITE
4) OUTPUT CLOSE
```

The last command will not echo and must be given when 'nothing seems to be happening'. Repeat 2-4 for each program being converted; a 'direct command file' may be created to perform these steps automatically. See page 9 of the OMSI manual.

```
5) load U/W-FOCAL
6) OPEN INPUT DEVICE:FILENAME (wait for a '_' to appear)
7) LIBRARY SAVE DEVICE:FILENAME;
   LIBRARY DELETE DEVICE:FILENAME.FD
8) ERASE (could go at the end of the previous step)
```

Repeat 6-8 for each program. If the 8/e version is used, steps 6-8 can also be automated by preparing a direct command papertape and placing it in the low speed reader.

Note: at some point - probably between steps 6 and 7 - one must convert all 'ERASE' commands into 'ZERO' commands. Usually these will be found on the first line of the program. Those which are not found will 'find themselves' when the program is run because they will clear the entire text buffer (ERASE is the same as ERASE ALL). Since the program is saved, having it 'self-destruct' is not a complete disaster and it may be called in again and examined more carefully.

A R I T H M E T I C   O P E R A T O R S

EXPONENTIATION	[^] (negative or positive integer exponent)
TYPE 3^2	(outputs a '9' --- 3 to the second power)
SET X=Y^Z	(sets X equal to Y raised to the integral Z power)
TYPE 10^(-2)	(outputs '0.01' --- the same as 'TYPE 1/10^2')
SET X=2^3.5	(sets X equal to '8' --- the .5 is dropped)
	(use the LOG/EXP routines for non-integer powers)
MULTIPLICATION	[*]
TYPE 2*4	(outputs an '8')
SET X=Y*Z	(sets X equal to Y times Z)
DIVISION	[/]
TYPE 6/2	(outputs a '3')
SET X=Y/Z	(sets X equal to Y divided by Z)
ADDITION	[+]
TYPE 2+2	(outputs a '4')
SET X=Y+Z	(sets X equal to Y plus Z)
SUBTRACTION	[-]
TYPE 3-X	(outputs the value of 3-X)
SET X=Y-Z	(sets X equal to Y minus Z)
ENCLOSURES	(),[], and <> may be used interchangeably in matched pairs to enclose quantities which are to be operated on as a unit.
HIERARCHY	^ * / - +
	note especially that multiply and divide are not equal. Thus A/B*C=A/(B*C)

A S K / T Y P E   O P E R A T O R S

RETURN/LINE FEED   [!]

```
TYPE !!!!,X,!!       (outputs 4 carriage return/line feeds, the value
                     of X, and then 2 more carriage return/line feeds.
```

CARRIAGE RETURN   [#]

```
TYPE !!"="#"/"!      (outputs two carriage return/line feeds,
                     prints = and / on top of each other,
                     then types another carriage return/line feed)
```

TAB   [:]

```
TYPE #:20 :-10 :45    spaces to column 19, inputs 10 characters,
                      then spaces to column 44
```

OUTPUT FORMATTER   [%]

See what the '%' does to line 12.30 first written then executed:

```
12.30 SET X=123.456;TYPE %6.04 X,!%4.02 X,!%3 X,!%-3 X,!%,X,!
  123.456
  123.5
  123
  1.23E+02
  1.234560000E+02
```

'%4.02' permits four digits to be output including up to two decimal places; 'TYPE %' is equivalent to 'TYPE %-10'
See additional information with the 'TYPE' command.

SYMBOL TABLE   [$]

```
TYPE $               (outputs list of defined variables, 3 in a row)
TYPE $5              (prints 5 in a row and changes the default value)
```

QUOTATION MARKS   ["]

Text may be typed by enclosing it in quotes.
See what the '"' does to line 14.65 first written then executed:

```
14.65 TYPE "THE ANSWER IS ",3+8,!!!
THE ANSWER IS    11.0000
```

S P E C I A L   C H A R A C T E R S

TRACE [?]

The first time FOCAL reads a '?' (except within quotes) it will start outputting the program while it is executing it; the next time it encounters a '?' (or upon return to command mode) it will stop typing out the program. See what the '?' does to the line below:

```
15.60 SET X=3;? SET Y=5;TYPE X/Y,!?
 SET Y=5;TYPE X/Y,   0.6000
```

A 'GO ?' command will trace the entire program. The trace feature is also useful for input prompting: 'A ?X ?' will print 'X ' and wait for a response.

POWER OF TEN [E]
See what the 'E' does to line 6.80 first written then executed:

```
06.80 TYPE %9.04,1E1,7E3,1.23E-2,3.76E217,!
  10.0000  7000.00   0.0123  3.760000E+217
```

CTRL/C (pressing 'CTRL' and 'C' at the same time)

Will return user to the OS/8 monitor. CTRL/F is the break character. --- the 'LOGICAL EXIT' command is the program equivalent of 'CTRL/C'.

CTRL/F (pressing 'CTRL' and 'F' at the same time)

U/W-FOCAL's break character --- stops program execution and returns I/O to the terminal.

'BELL' (pressing 'CTRL' and 'G' at the same time)

Used with the 'MODIFY' command to change the search character.

CTRL/L (pressing 'CTRL' and 'L' at the same time)

Used with 'MODIFY' to skip to the next search character.

CTRL/Z (pressing 'CTRL' and 'Z' at the same time)

Is the last character in a U/W-FOCAL data file. Attempts to read past the end-of-file will cause a '_' to be typed on the terminal and will restore the terminal as the input/output device.

LINE FEED

Used with the 'MODIFY' command to retain remainder of modified line. Also used during command input to retype the input line just as the OS/8 monitor and command decoder do.

DELETE INPUT   [_] (back arrow, shift/J, underline sign)

When writing a program, '_' deletes everything over to left margin. In response to an 'ASK' command, '_' kills the number being entered. Printed on the terminal when attempting input beyond the 'end of file'.

RETURN KEY

Used with the 'MODIFY' command to delete remainder of modified line. 'RETURN' is a legal symbol for separating data in response to an 'ASK' command. RETURN is a required terminator for all command input.

RUBOUT KEY (or 'DELETE' on some terminals)

When writing a program, one character will be erased each time the RUBOUT key is struck; RUBOUT will show as a '\' --- thus: 'PLWEA\\\EASE' becomes 'PLEASE'. After many rubouts it is a good idea to hit LINEFEED to see what is left.

ALT MODE KEY ('ESCAPE' or 'PREFIX' on some terminals)

In response to an 'ASK' command, ALT MODE retains the previous value of the variable. (A patch is needed for teletypes with a 'PREFIX'-key; see Appendix III)

SPACE KEY

SPACE is a legal symbol for separating variables and expressions in many commands, and for delimiting input during an ASK. It is required for separating command words from their operands.

COMMA   [,]

A comma is a legal symbol for separating data in ASK and TYPE commands. It is a required separator between line numbers and other arithmetic expressions in many commands or between multiple function arguments. See the effect of the ',' in line 26.40 below:

```
26.40 TYPE 2,3,5/6,12^3,!!!

   2.0000   3.0000   0.8333  1728.00
```

SEMICOLON [;]

Separates commands when placed together on one line. See what the ';' does to line 27.42 first written then executed:

```
27.42 SET X=17;TYPE 3*X;SET X=3/7; TYPE "   ",X+3,!
  51.0000      3.4286
```

TERMINATORS:

NUMERIC INPUT:	Anything except 0-9, A-Z; only one decimal point or 'E' is allowed, leading spaces are ignored.
EXPRESSIONS:	Comma, semicolon, CR, trailing spaces
VARIABLES:	Space, comma, left parenthesis, arithmetic operator
COMMANDS:	Space, comma, semicolon, carriage return

MISCELLANEOUS:

EXPRESSIONS --- an evaluatable group of numbers, variables and/or functions: (2+3*X) or [2*F(3.6)]/FSR().

Using the high speed paper tape reader to read in FOCAL programs:

1) Type 'OPEN INPUT PTR:'
2) After '^' is typed out, hit space bar to read tape
3) Wait for '_', then continue

The following 3 line program will ask for 10 numbers from the reader:

```
10.60 OPEN INPUT PTR:
10.65 FOR X=1,10;ASK A(X)
10.70 OPEN INPUT,ECHO;COMMENT --- restores the terminal for input
```

LINE NUMBERS --- may range from 1.01 through 31.99 but must not include integers. Variables and/or expressions may be used in place of line numbers (example: 'GOTO X'). Group numbers are integers from 1 - 31 and reference groups of lines with 'DO', 'ON', 'WRITE' and 'ERASE' commands and FSF's. Line or group 0 refers either to the entire program or to the next command, depending upon the context.

MERGING PROGRAMS --- Merging 'A' and 'B' is done with a series of commands which convert 'A' into a data file (OS/8 editor compatible) and then bring it in 'on top' of program 'B': Note the convenience of changing line numbers with the MOVE command in between.

```
  1) LIBRARY CALL A
  2) O O A;W;O C
  3) LIBRARY CALL B
  4) O I A
  5) (wait for the '_' to appear)
  6) L D A.FD   (remove A, if necessary)
```

The merged program will now be in your program buffer.

PROGRAM NAMES --- up to six alphameric characters. FOCAL assumes an .FC or .FD extension (program or data). The use of .FB for files in use with the random access function 'FRA' and .FH for FOCAL help files is suggested. FOCAL help files explain how to use a program. For example, start a program by asking: "DO YOU KNOW HOW TO USE THIS PROGRAM" and then do a 'L G ----.FH'.

PUSHDOWN LIST OVERFLOW --- too many 'DO' or 'LIBRARY GOSUB' commands have been given without a 'RETURN'. Remedy: reduce nested calls - look for lines which call themselves and replace 'DO's with 'GOTO's if possible.

STRINGS --- any series of characters such as:

```
HELLO
122.5
$99.95
NOW IS THE TIME FOR ALL GOOD PEOPLE ....
TEST34
```

These strings were printed out ['122.5' is also a number] by surrounding them with quotation marks in a TYPE statement as in line 22.25:

```
22.25 TYPE "HELLO"!
```

Strings may also be printed out character by character using the 'FOUT()' function; input of strings may be handled with the 'FIN()' function. The following example reads a string of characters terminated by a carriage return and then retypes them:

```
        2.1 FOR I=2,100;SET C(I)=FIN();IF (C(I)-141),2.2
        2.2 BREAK;SET C(1)=I-1;FOR I=2,C(1);X FOUT(C(I))
```

By using double subscriting (see p. 11) a 'message array' can easily be constructed with the character count of each string stored as the first element.

The ASK command may be used to input short strings for comparison:

```
22.78 COMMENT: 'YES OR NO' SUBROUTINE
22.80 ASK "ANSWER YES OR NO ?  ",AN
22.82 IF (AN-0YES)22.84,22.86
22.84 IF (AN-0NO)22.8,22.88,22.8
22.86 SET X=2;RETURN
22.88 SET X=1;RETURN

ANSWER YES OR NO ?  YEP
ANSWER YES OR NO ?  NOPE
ANSWER YES OR NO ?  YES
```

Program control would then return to the commands following the subroutine call ('DO' or 'LIBRARY GOSUB') with X equal 2. This form of string comparison is limited to responses with only one 'E'!

VARIABLES --- one or two characters such as: A, X, Z7, P2, PI, AB; If 'ABCDEFG' were used as a variable, only 'AB' would be significant. Variables may not start with an 'F' or a digit.

An infinite number of variables are permitted to be equal to zero; Thus, you should zero variables when no longer needed. (ZVR feature).

'!','"','#','$' and '%' are protected variables and cannot be 'TYPED', or 'ASKED' directly, but may be 'SET' and otherwise used as regular variables. Note the use of '!' as the dimension constant for double subscripting and the use of the others in FOCAL Statement Functions. These variables (and 'PI') retain their values following a 'ZERO' command.

Note: 'TYPE =', 'TYPE)', 'TYPE]', or 'TYPE >' output a string of zeroes. This can be interrupted by typing 'CTRL/F'.

----> C T R L / F   I S   T H E   B R E A K   C H A R A C T E R <----

***** D O U B L E   S U B S C R I P T I N G *****

One of the most significant features of U/W-FOCAL is the addition of double subscripting. This facility allows FOCAL to be used to diagonalize matrices and solve linear algebra problems which were essentially impractical before. Furthermore it permits simple translation of programs written in other languages (such as FORTRAN or BASIC) without resorting to special tricks for dealing with two-dimensional arrays. The maximum size of such arrays is obviously limited by the number of variables. For the standard version of U/W FOCAL this limit is a 10*10 matrix if all the elements are non-zero. Because of the ZVR feature however, if the array is tridiagonal, it could be as large as 43*43! Removing some of the functions and/or adding more memory can increase this size still further: The 16k version can handle a 26 x 26 matrix.

Doubly subscripted variables have the standard form: A(I,J). Either I or J (or both) may be expressions involving functions, other subscripted variables, etc. Such expressions need not have integral values (although only the integer part will be used) and in fact may be zero or even negative. (This is also true for singly subscripted variables.)

Internally, FOCAL assigns a single subscript to all variables. Thus an unsubscripted variable and one with a subscript of zero are identical: X=X(0)=X(0,1)=X(!,0). In order to convert from two subscripts to one it is necessary for FOCAL to know the number of rows in the array, that is, the maximum value of the first subscript. This number must be stored in the first secret variable ('!') prior to the first use of double subscripting. If ! is zero, or less than the largest value of the row index, the resulting subscripts will not be unique. The algorithm is index=integer part of J*!-!+I. This is equivalent to ordering the array column by column. Thus B(1,2)=B(6) if !=5, which is the way the symbol table dump would appear. Using this scheme the number of columns is not important in accordance with the common practice of appending several columns to the matrix of coefficients when solving sets of simultaneous linear equations.

The use of ! for this dimension constant is generally quite convenient since it is protected from the ZERO command and is thus easily shared between subroutines and because, being a variable, it makes an ideal loop constant. In this way matrix programs are easily written for any size array. For a simple example, consider the statement - FOR I=1,!;FOR J=1,!;TYPE A(I,J);NEXT;TYPE ! - which prints an entire !*! array with ! determined elsewhere. In this way matrix programs are easily written for any size array. For non-square arrays it is recommended that the second protected variable (") be used for the number of columns. Also, since ! is a variable, it is possible for the program to change the "dimension" for different arrays should this need ever arise. The 'examples' section shows several sample programs using double subscripting. Note in particular the 3 line GAUSSIAN matrix inversion program. This same routine requires 12-13 FORTRAN statements and about 15 lines when coded in BASIC.

**** P R O G R A M S ****

Programs are usually created directly from the terminal, but program tapes may also be created off-line (i.e. with a terminal set to 'LOCAL') or by use of a general-purpose editor and then read in later. FOCAL's built-in editor, however, is very convenient since it handles line references directly and permits duplication of lines with little effort. Once the program has been entered you will probably want to save it just in case an 'accident' occurs. The program can be saved either as an ASCII data file by WRITING it out, or, more commonly, as a 'core-image' load file by using the LIBRARY SAVE command.

U/W-FOCAL provides a number of 'LIBRARY' commands (inherited from OMSI's PS/8 FOCAL) for saving and recalling programs. These are described in detail later on, but here is a quick summary:

LIBRARY SAVE	-saves the current program
LIBRARY CALL	-reloads a program
LIBRARY RUN	-loads and starts a program
LIRBARY GOSUB	-runs a program as a subroutine

The last command is especially potent since it permits coding a more-or-less general purpose subroutine which can then be called by several different programs. In contrast to the OPEN commands described next, the LIBRARY commands use 'core-image' files which are not compatible with other OS/8 system programs. They are normally given an extension of '.FC' to identify them as FOCAL programs. Programs saved in this way are 'ready-to-run' when loaded into core and may thus be used as subroutines without further processing. They contain the program name and the date they were saved in the header line as well as information about the amount of core storage required. Each system block provides room for approximately 500 characters of text storage. In order to pack programs as near to the beginning of the storage device as possible, the SAVE command computes the amount of file space actually needed and then tries to fit the program into the nearest 'hole' of this size. 'Deleting' an old copy of the program first will thus usually restore the new version in the same place - unless it has gotten too big to fit!

The LIBRARY LIST command will print out a listing of all the .FC programs on any device and LIST ONLY will quickly verify whether a specific program is available. It is a good practice to check for the existence of a program with the same name -BEFORE- you save something since otherwise the OS/8 system will delete the other program without telling you about it! While the 'L' commands assume an extension of .FC, any other extension can be given explicitly. The LIST command, for example, can be made to list all programs with a .PA extension by using a command like 'L L .PA'. This command will also list all files with the same name, regardless of the extension: 'L L FLUNKY.' will list all 'FLUNKYs'!

**** I N P U T / O U T P U T ****

U/W-FOCAL normally uses the terminal for all I/O operations. Thus ASK and TYPE receive input and produce output on the terminal. To allow the use of other I/O devices, such as the paper tape reader or line printer, FOCAL calls upon the OS/8 device handlers which are a part of the operating system. These handlers are loaded into core by a series of 'OPEN' commands which can thus switch the input and/or output 'channels' to any device for which a standard system handler has been written. The 'OPEN INPUT PTR:' command, for instance, switches the input channel to the PTR: (see the list of OS/8 device names on page 15). All input operations such as ASK or FIND() then use the tape reader rather than the terminal. This also includes the text input routines, thus when FOCAL is in command mode you can type in the command above, place a program tape in the reader, and quickly read in a new program. The terminal will be restored as the input device when the reader runs out of tape (see below).

Likewise, the command 'OPEN OUTPUT LPT:' can be used to direct output to the line printer rather than to the terminal. Data and programs can also be kept in permanent files on mass storage devices such as LINCtapes or DECtapes or floppy disks. All files accessed by the 'OPEN' commands are formatted in 8-bit packed ASCII and may be read or written by other OS/8 system programs such as PIP, EDIT, TECO, etc. These files contain 384 characters per OS/8 block.

The form of the I/O device/filename specification used by the 'OPEN' commands is similar to that used by the keyboard monitor or the command decoder, i.e. device names are delimited by a colon (:) and file names (if any) may include an explicit extension code separated by a period. FOCAL does not allow embedded spaces in the name since the space character is sensed as a separator for the error trap option described below. The default device is 'DSK:' if none is specified, and the default extension for the 'OPEN' commands is '.FD'. (Default extensions for other commands are described elsewhere.) A command such as 'O O DTA1:MYDATA' would thus set up a temporary file on DECtape 1 with the name 'MYDATA.FD'.

The 'OPEN' commands also provide an 'ECHO' option. This permits connecting the Input and Output devices together, just as the keyboard is normally connected to the terminal printer. This linkage is specified by adding the phrase ',ECHO' (or just ',E') after the device/filename specification. Thus 'O I MYDATA,E' will switch input to the file 'MYDATA.FD' on the default device 'DSK:' and echo each character as it is read in. The 'echo' is sent to the selected OUTPUT DEVICE, not necessarily to the terminal!

The ECHO option is also available during output. In this case characters sent to the output buffer are also printed on the terminal so that you can monitor the otherwise invisible results. Beginning programmers often find the ECHO options somewhat confusing, so if the preceeding discussion seems unclear do not despair but read on, look at the examples, and as soon as possible attempt to actually try things out.

It is frequently necessary to switch I/O from the terminal to a file and back again. To select the teletype for input just use the 'OPEN INPUT TTY:,ECHO' command. This will return input to the keyboard, and similarly 'O O TTY:' will return all output (including the keyboard echo) to the terminal printer. Since it is a nuisance to have to write out 'TTY:' all the time, the terminal is assumed if no other device or file name is given. Thus most programmers would just write 'O I,E;O O' for the commands above.

To switch back to file input or output you use the OPEN RESTORE commands: O R I or O R O. Again the echo option is appended, as in 'O R I,E', etc. The 'RESTORE' commands simply pick up from wherever the current file pointer is, just as though no other I/O operations had intervened. In the case of an input file, if the file pointer is already at the 'End-of-File' character (CTRL/Z is used to mark the EOF), input is automatically returned to the terminal and an error message is printed.

Since it is usually nice to avoid error messages, the OPEN commands provide, in addition to the 'echo' option, an error trap feature. This is included at the end of the command and consists of a line number, separated by one or more spaces, indicating where to branch if an error occurs. Thus 'O R I,E 12.1' will attempt to return to input from a file, but if the file is empty the program will branch to line 12.1. Similarly the command 'O O DUCKY 19.7' will branch to line 19.7 if there is no room for a new file on 'DSK:' or if there is already an active output file. (Only one OS/8 output file can be open at a time. However, users may add their own internal device handlers to get around this problem. The PLTR: handler discussed in the graphics section provides an example of this approach.)

Output files are normally terminated by an 'OUTPUT CLOSE' command. This command empties the output buffer onto the device and saves the file if the device is file structured (disk or magnetic tape). The terminal is then restored as the output device.

The 'OUTPUT BUFFER' command simply dumps the current contents of the buffer without closing the file or restoring output to the terminal. This permits 'line-by-line' output on devices such as the TV: or LPT:.

Finally, the 'OUTPUT ABORT' command discards the output file and returns output to the terminal. All 3 'closing' commands are ignored if there is not an active output file.

The OPEN RE READ command is useful for restarting input from the beginning of a file after a program interruption. It saves the overhead of rewinding a tape all the way back to the directory in order to relocate the file. It is also useful in sorting programs which require many passes through the same data.

All program errors immediately restore the terminal as the I/O device (as does typing CTRL/F). The RESTORE commands can be used to continue from the point of interruption or else the output file can be aborted and the input file re-started as indicated above.

Note: When writing FOCAL data files, it is necessary to include a space, comma, carriage return or other delimiter preceding any minus signs, otherwise the number will appear positive when 'asked'. This is the reason for the initial leading space produced by the TYPE command.

Several extensions to the usual OS/8 device and file name expressions are discussed on the next few pages. These are especially convenient for processing data files such as might be obtained in experimental work. A summary at the end indicates the many different kinds of file name expressions which can be used in the OPEN commands.

- - - - - - -

*** OS/8 DEVICE NAMES: ***

SYS:	System device (DSK: in disk system; DTA0: in DECtape system)
DSK:	The disk in disk systems; DECtape #1 in DECtape systems ('DSK:' is assumed if a device is not specified)
DTA0: - DTA7:	DECtape drives
LTA0: - LTA7:	LINCtape drives
MTA0: - MTA7:	Magtape drives
RKA0: - RKB1:	Removable disk packs
RXA0: - RXO1:	Floppy disk drives
DF:	Small fixed head disk
RF:	Large fixed head disk
PTR:	Paper tape reader
PTP:	Paper tape punch
LPT:	Line printer
TTY:	Terminal (may be used with other devices through 'ECHO')
CDR:	Card reader (2 page handler)
TV:	PDP-12 scope (2 page handler)
DL:	DIAL LINCtape handler (2 page handler)

and others, installation dependent - type 'RES' after the dot of the OS/8 monitor to find them all. 2-page handlers require a 12k system.

Note: use 'CTRL/SHIFT/P' to advance the display of the 'TV:' handler, otherwise an 'input buffer overflow' error may occur.

***** V A R I A B L E   F I L E   N A M E S *****

This feature represents a major improvement to the library and file commands by permitting programmable names in place of fixed ones. This improvement satisfies the needs of data collection and analysis programs by permitting letters and numbers to be programmed anywhere in a file name, including the extension and device codes.

Data files, for example, are commonly given a short label plus several sequence digits, e.g. DATA1, DATA2, etc. Such names may now be specified by enclosing the variable numeric part in parentheses i.e. DATA(N) where N would be programmed to have the value 1,2,3,... The quantity in parentheses need not be a single variable, any legitimate FOCAL expression may be used, including FOCAL statement functions! The decimal digits of the integer part of the expression are then used to complete the name. A typical program sequence might be: S #=#+1; O O EXPT(#). The numeric part is not restricted to the end, it may be placed anywhere; Thus 'OPEN INPUT (M)TO(N)' might call a file representing data values in the range M to N. Also both the device and the extension may be programmed: O O DTA(N):FILENO.(I). This sort of thing could be useful in conjunction with FLEN in order to select a device with the most room.

Any sort of expression may be used, but the result should be positive since negative values serve a special purpose (see below). The digits are supplied from left - to - right with no zero fill so that if two-digit numbers in the range 0-99 are desired (e.g. 01, 02 ..99) a small amount of programming is required. The simplest solution is just to program the two digits separately: F I=0,9; F J=0,9;O I FILE(I,J) would do the trick.! Another solution which might be used is the following:

```
FOR N=,.1,9.9;O I FILE(N,FRAC(N)*10.1)
```

After 6 characters (including any letters) have been filled, additional characters are ignored. Thus if A+B has the value 12345 the name "RUN(A+B)" will actually be "RUN123". Numbers less than 1 appear as a single zero. The device name is limited to 4 characters and the extension to 2. A few examples illustrating the powerful nature of the variable filename feature are shown below:

FOR J=1,N;L GOSUB PROG(J)	calls a sequence of subroutines
FOR N=,3;LIST ALL DTA(N):	Lists the entire directory of 4 tapes
LIBRARY RUN NEXT(ONE)	lets 'ONE' decide what the program will be

For experimenting with this feature, the LIBRARY NAME command is handy: FOR Q=1,10;L N TEST(Q);W produces 10 header lines (if no program!). Numbered devices may be created with the monitor assign command: .AS LPT:DEV1:

Since the conversion routine only works properly for positive expressions, it was decided to use negative values to program non-numeric characters. This is done by using the negative value of the ASCII code. Thus the expression '(-189,-201)L(-197,M,N)' is equivalent to the name 'FILE(M,N)'. Code -192 is permitted, but it should only be used to fill out a name containing less than 6 characters. 'FIN' can be used directly in the name expression, but this method does not permit any error correction in case of mis-typing. It is better to set variables with an input loop which checks for rubouts and other special characters. An example is shown below.

```
9.1 ASK "ENTER A FILE NAME: ";FOR I=6,-1,1;S C(I)=-192;C 192=NULL
9.2 SET C=FIN();I (C-255),9.3;I (C-141),9.4;S I=I+1,C(I)=-C;G 9.2
9.3 S C(I)=-192,I=I-FSGN(I);I (C)9.2;T "\";G 9.2;CHECK RUBOUTS
9.4 L N (C<1>,C<2>,C<3>,C<4>,C<5>,C<6>);!;NOW HIT LINEFEED KEY.
```

This routine uses the following ideas: (A) when the loop in 9.1 ends the index I will be one increment beyond the limit, viz. 0; (B) line 9.2 sets C(0) to a positive value and checks for rubout (255) and carriage return (141). If it is neither of these the index is advanced and the negative value stored in the character array; (C) line 9.3 handles rubouts in a very clever way: first the last character is 'erased' and then the index is backed up, but in a way (due to OMSI) which ensures that it will never go beyond zero! However, if it is already zero, then C is reset to a negative value and no backslash is echoed. Line 9.2 could be extended to test for 'LINEFEED' to retype the line, and 'BACK ARROW' to restart input from the beginning, however the simplified version shown works well for most cases. This routine can also be called first to read a device name and after saving C(1-4) as D(1-4), called again to read the file name. Alternatively the characters ':' and '.' may also be programmed as part of the name expression but with the restriction that commas may not be used to separate the character codes. This permits very simple coding for reading any OS/8 device or file name. The following somewhat awkward name expression is required, however:

```
O I (C<1>)(C<2>)(C<3>)(..)(C<13>)(C<14>)
```

Another important feature is the ability to specify the maximum expected file length when opening a file. This facility (which is automatically invoked by the SAVE command) permits placing short files in 'holes' near the beginning of a tape, thus greatly reducing the access time. The size specification is enclosed in 'square brackets' immediately following the name and before the comma which is present if the echo option is included with the name. Thus 'O O BIGGIE[50]' locates a hole as close to 50 blocks long as possible. It may be bigger than 50, you can use FLEN() to find out the exact size. When copying a file from one unit to another it is useful to know how big the input file is. This is the reason for FLEN(1). With these features it is possible to update a file 'in situ' provided it doesn't get any longer. Note the closing size option for 'OUTPUT CLOSE' to deal with this problem:

```
9.1 O I OLDONE,E;L D OLDONE.FD;O O NEWONE[FLEN(1)];...;O C;O I,E
```

*** D I R E C T   A C C E S S ***

A new 'direct access' feature has been incorporated. This feature permits block numbers to be substituted for file names and thus bypasses the directory search required to get this information. Block number expressions are enclosed in 'angle brackets' and may be followed by a file size specification as well as the echo switch. Error returns do not occur. This feature is limited to input operations for the obvious reason that it is dangerous to permit writing just anywhere on a device. This feature can be used to recover data written into a file which didn't close properly or was inadvertently deleted - perhaps because it overflowed the available space. Use of this feature can also save an enormous amount of tape motion when chaining from one program to another or when processing sequentially stored files.

In order to call programs by their block number it is necessary to first determine their location. Several methods may be used, for example by obtaining a directory listing with a 'DIRECT' command using the /B option. This will list the block numbers in octal; they can be converted to decimal by hand. The easiest way is to write a little program which asks for the file name, performs an open input (see the example on the previous page) and types or saves the values of FBLK() and FLEN(1).

Since files are read on a block by block basis only the starting block is required. However when using this feature to link programs or call subroutines it is also necessary to indicate the program size (in square brackets) since programs are read in all at once. This information can also be determined during an initialization phase from the FLEN(1) function, but for most uses it suffices to simply specify the largest program size. Information beyond the end of the program will also be read into core, but it will have no effect upon program execution (aside from interfering with the push down list in the 8k version, or the FCOM area in the 12k and 16k versions).

To summarize the many different kinds of file names and specifications now permitted by U/W-FOCAL, we list below several examples:

simple file names	L C PROG	loads 'PROG.FC'
	O I DATA	finds the file 'DATA.FD'
echo & size options	O R R,E	restarts input with the echo on
	O O TINY[1]	puts TINY.FD in a 1 block hole
branch options	L R LSTSQR 5.1	starts LSTSQR at line 5.1
	O I RESULT 29.3	branches to 29.3 if no 'RESULT'
variable names	L A DTA(N):	Lists directory of tape N
	O I FILE(I,J) 9.9	Finds FILE--, if possible

```
block number spec.     L G <S(I)>[5] G(I)    calls subroutine S(I),
                                             executing group G(I)
                       O I LTA1:<20*B+7>,E   begins input at block
                                             20*B+7 with the echo on
```

***** COMMANDS: *****

Direct commands are given while FOCAL is in command mode. They are typed without line numbers and FOCAL executes them as soon as the return key is hit. For example:

```
*TYPE 3+4,!      (FOCAL outputs the value of 3+4)
  7.0000
*
*SET X=3
*SET Y=2
*TYPE 3+X+Y,!
   8.0000
*
```

Indirect commands are used for longer programs. They are typed following line numbers between 1.01 and 31.99, not including integers and may be executed by a direct 'GO':

```
*2.1 SET X=3
*2.2 SET Y=2
*2.3 TYPE 3+X+Y,!
*GO
   8.000
*
```

-- COMMAND FORMAT --

The general form of each command is given followed by examples in which: < > enclose required terms. [ ] enclose optional terms. ( ) enclose comments. One letter abbreviations may be used for command words. X represents a variable. E1, E2 and E3 are arithmetic expressions. L1, L2 and L3 are line numbers. G1 is either a line- or a group number. L1, L2, L3 and G1 may be replaced by any arithmetic expressions.

Most commands must be followed by a space and many permit several operations to be performed with only one command. For example:

```
*2.1 SET X=3,Y=2
*3.1 WRITE 1,3,5,7,9
*4.1 DO 1.8,1.9,3
*5.1 XECUTE FMQ(12),FOUT(135)
```

```
*****  * * * * * * * * * *   * * * * * * * *  *****
*****  E L E M E N T A R Y   C O M M A N D S  *****
*****  * * * * * * * * * *   * * * * * * * *  *****
```

A S K

```
*ASK [! IDENTIFICATION][%E1][:-E2],[X]  (variable input command)

ASK Y                (expects a value to be input for Y)
A Z                  ('A' is the abbreviation for 'ASK')
A B2,X,NUMBER        (expects three values to be input)
A "HOW MANY? "M      (types 'HOW MANY' then sets M equal to the response)
ASK ?A1 ?"A2 "A2     (prints "A1 ", inputs A1; prints "A2 ", inputs A2
A !"FORMAT?"VF,%VF   (returns carriage, reads and sets a variable format)
A "SET SWITCHES":-1  (types message, waits for one char. from the TTY)
A :-8 N :-18         (skips 8 characters, reads N, skips 18 more - v.i.)
```

Several techniques are illustrated above: a) the use of the trace to provide simplified prompting; b) a way to set a variable format; c) a new addition to the tab command which permits skipping input characters by specifiying a negative column number. Thus :-10 reads and ignores the next ten characters; if the echo is on the characters will be printed, otherwise not. This feature may be used to create a single program pause by reading one character from the teletype, hitting any key will continue the program. A more sophisticated use is for re-reading an input file which contains comments and other identification. Thus the output for the last example might look like: EXPT. NO 7 DATE 4/13/73 X=... etc. The command shown skips over the labels, inputs the 7, then skips the data to get the data.

An optional patch provides an automatic ':' as a prompt character for each input item - see Appendix III. Since any character except 0-9, A-Z, RUBOUT and LINEFEED will terminate input, it is important that negative numbers stored in a file be preceeded by a space, comma or other delimiter so that the '-' sign does not serve as a terminator. The TYPE command normally supplies a leading space to avoid this problem. Remember that RUBOUT does not work for numeric input - type a '_' (back arrow, shift/O, underline) to erase a partially entered number. Use ALTMODE to leave the variable unchanged from its previous value.

ASK preserves the last terminator so the program can check for the end-of-input by looking for a special terminator. This is a feature which FOCAL has needed for some time; it is decidedly superior to checking each item for a special value since no assumptions need to be made about the range of data values. While the space bar and carriage return are common terminators, a comma, semicolon, question mark, double periods, or any other special character will do just as well. The example on next page illustrates a typical input loop:

```
1.1 TYPE "END INPUT WITH A CR - OTHERWISE USE THE SPACE BAR";ZERO N
1.2 SET N=N+1;ASK ! X(N) Y(N);IF (FTRM()-141)1.2,,1.2;T N " ITEMS"!
```

The new FTRM function reads and compares the last terminator with the value (141) specified. Note that input from a null file will be terminated by a CTRL/Z (code 154). This allows FOCAL to read data from a file without knowing in advance how many data points to expect. However, carriage returns should not be used as data separators in such files lest they, rather than CTRL/Z, terminate the last item.

B R E A K

*BREAK [L1]	(terminates a FOR loop and continues as directed)
B;CONTINUATION	(breaks are ignored unless executed within a loop)
B 14.1	(specifying a line number continues from that line)
B;B 5.1	(successive breaks are required for nested loops)

The BREAK command endows FOCAL with the ability to conditionally exit from a loop, preserving the current value of the loop index. A BREAK command may be inserted anywhere in statements executed by a FOR loop. The loop is terminated immediately upon encountering the break and the program continues from that point, or from the line specified. The following example reads and processes data points, terminating upon reaching the end of file character:

```
1.1 O I FILE
1.2 FOR I=1,1000;ASK X,Y;DO 2;ON (FTRM(154)),1.3;DO 3
1.3 O I,E;TYPE "THERE WERE"I" DATA POINTS"!;BREAK 4.1
```

Note that group 3 will not be executed after line 1.3 in spite of the use of the 'ON' test because the 'BREAK' command stops the loop immediately and (in this case) transfers the program to line 4.1. See also the NEXT command.

C O M M E N T / C O N T I N U E

*C	(lines beginning with a 'C' will be ignored)
COMMENT SORT	(comments are used to tell about a program)
CONTINUE	(dummy line)
C PRINT ROUTINE	

Comments can be added to the end of any line. No commands following the comment will be executed however.

D O

```
*DO [G1,G2,G3]          (branch - and - return subroutine call)

DO                      (begins the program - same as 'DO ALL' in FOCAL 69)
D 0,0                   (runs the entire program twice - "D," is equivalent)
D 1.3,1.4,5             (calls lines 1.3,1.4 and then group 5)
D A,B,C                 (calls three subroutines specified by the program)
D 2+#/10                (selects a single line of group 2 according to #)
```

The DO command now handles multiple calls which is especially convenient in loops as well as for direct commands. 'DO ALL' no longer exists, since the letter 'A' is no longer reserved for that use.

E R A S E

```
*ERASE [LINE OR GROUP NUMBER]  (deletes part of the indirect program)

ERASE                   (clears the text buffer - does not affect variables)
E 3                     (deletes any group 3 lines --- 3.01 through 3.99)
E A                     (removes line or group A if present)
```

The ERASE command removes the program name from the header, fills in the current date and then returns FOCAL to command mode. ERASE without an argument is the same as ERASE ALL (E A) in previous FOCALs. The 'ZERO' command must be used to clear the symbol table. Only one line or one group can be erased at a time.

F O R

```
*FOR X=E1[,E2],E3;COMMANDS    (repeats 'COMMANDS' 1+(E3-E1)/E2 times)

FOR P=1,7;TYPE 3      (outputs a '3' seven times)
FOR J=1,5;DO 17.1     (does line 17.1, five times)
F X=2,2,8;T X/5,!     (outputs .4 .8 1.2 and 1.6 )
FOR Z=1,Y;DO 7        (does group 7, Y times)
FOR I=-5,5;ASK X(I)   (will input 11 numbers called X(-5) .. X .. X(5))
F N=5,-1,-N;T X(N)    (will type 11 numbers: X(5),X(4) .. X .. X(-5))
F A=,PI/180,PI;D 3    (scans angles from 0-180 degrees in radians)
```

The 'range' of a FOR loop extends only to the end of a line, unless it is shortened by a NEXT command. Subroutine calls may be used, however, to extend the range of executable statements to include entire programs if desired. The loop index is incremented by the value of E2 (or 1 if E2 is omitted) at the end of each cycle until the value exceeds E3 in magnitude. Note that both negative and non-integer increments are permitted. Also since 'FOR' begins with a 'SET' it is possible to use the initial value of the index to compute the increment and limit. See also the NEXT and BREAK commands.

G O

```
*GO [LINE NUMBER]  (starts executing program at designated line)

GO 8.17            (starts executing program at line 8.17)
GOTO 3.1           (starts executing program at line 3.1)
GO                 (starts executing program at lowest line number)
G                  (same)
GO ?               ('trace feature' -- prints program during execution
                    until next occurance of '?')
G X                (starts executing program at line X)
```

H E S I T A T E

```
*H [E1,E2,]      (stalls the program for a specified period of time)

H 1000           (waits for 1 second - times are in milliseconds)
H -4,1000        (set time base and delay count for 1 second (PDP12))
H -5,100,0       (ditto and wait for one delay period immediately)
H,,,FOUT(135)    (wait for 3 clock flags,ring the bell, wait for 1 more)
```

This command is installation dependent (and therefore optional). The first form shown utilises a software wait loop which gives moderately accurate delays but does not compensate for program execution time. The remaining examples illustrate a version using a programmable clock such as the KW12A or DK8-EP to provide accurate delays without concern for execution time other than ensuring that it be somewhat less than the specified delay time. If a clock is used, it is not necessary to set the period, unless you wish to change it from 1 millisecond. But if you do, there must be a new delay count too. Delay counts must be less than 4095. Initially the period is set to milliseconds (-4)*. Specifying (-5)# gives a time base of 0.01 seconds, and (-6)% gives a rate determined by input # 1. If the clock flag is already set there is no wait (i.e. the program must keep up with the specified delay). To wait for at least one delay period, use the command 'H,' (=H 0,0). To ring the bell on the Teletype every 4 seconds:

```
    1.1 H 4000; C OR H -5,400
    1.2 H FOUT(135);L B 1.1;R
```

* Use -3 for the DK8-EP clock; # ditto, use -2; % ditto, use -1

I F

```
*IF (E1)[L1,L2,L3]        (branch to L1, L2 or L3 as E1 is -,0,+)

I (H-5)1.3,1.5,1.7 (IF checks the value inside the parentheses)
IF (Y)2.7,1.2,L1   (FOCAL goes to line 2.7,1.2 or L1 if Y is -,0,+)
I (3-3)2.8,7.9;    (goes to next command if (8-3) is positive)
I (Z)2.7;T 21      (if Z is less than 0, then 2.7; otherwise 'TYPE 21')
IF (#-2)A,B,C      (goes to line A, B or C if # is 1, 2 or 3)
I (Y)5.6,,5.6      (continues the program if Y=0,otherwise goes to 5.6)
I (Z),7.8;T PI     (goes to line 7.8 if Z is zero, otherwise types PI)
```

Omitted line numbers (or a value of zero) may be used to indicate a branch to the rest of the line; if no commands follow on the line, the next sequential program step will be used unless the IF command is part of a 'FOR' loop. In that case the loop index will be incremented and the program flow will be determined by the loop command. Be careful not to use excess right parentheses as they are not trapped as an error. Also avoid line-number expressions containing double subscripted variables and/or functions with more than one argument. The additional commas may cause incorrect branching. See also the 'JUMP' and 'ON' commands.

J U M P

```
*JUMP (E1)[L1,L2,L3,L4,L5,...]  (computed goto command)

JUMP (L)1.9,2.5,3.7,4.2    (line 1.9 if L=1, line 2.5 if L=2, etc.)
J (X),,3.3,2.9,,4.1,,5.7 (branches only if X=3,4,6 or 8)
J (N/2-1)4.5,,8.9,10.11    (if N=4 or 5 goto 4.5, N=8,9 -> 8.9, etc.)
J (FIN()-192)9.1,8.7,...   ('A'->9.1, 'B'->8.7, 'C'->...)
```

JUMP is much like 'IF' except that the integer value of an expression is used to select the desired branch. If the expression is less than 1 or greater than the number of branches, or if the line number is omitted, no branch occurs and the program continues with the next command. Expressions not containing commas may be used in place of fixed line numbers. In particular, a construction such as 'F(3)*0' may be used to 'DO' all of group 3 and then return to the next command. See the FSF discussion for more details.

K O N T R O L

Installation dependent command. Used for example on the PDP-12: The relays are treated as a 6 bit number, so they may be set or cleared simultaneously. Relay 0=32, 1=16, 2=8 .... 5=1, thus decimal 25 (=16+8+1) will close relays 1,2 and 5 and open the others.

M O D I F Y

*MODIFY <LINE NUMBER>   (edit line)

'MODIFY 3.72' followed by a 'RETURN' and the letter 'X' will cause line 3.72 to be typed through its first 'X'. You may then:

1) Type in rest of revised line and hit 'RETURN'
2) use 'RUB OUT' to erase single characters
3) hit 'LINE FEED' to keep remainder of old line
4) hit CTRL/L to go to next occurrence of 'X'
5) hit CTRL/BELL to change search character; then hit new character
6) hit '_' (back arrow) to delete line over to left margin
7) hit 'CR' to delete line over to the right margin
8) hit CTRL/F to abort leaving the line unchanged.

MODIFY does not echo CTRL/L, hence no form feed or screen erasure takes place. Also line numbers ending in '.63' no longer cause any problems. Typing just 'M(return,linefeed)' will print the program header. This is useful when interrupting a program which calls subroutines to see which one is in core. 'MODIFY' may be patched to print the line number at the beginning of the line - see Appendix III.

M O V E

*MOVE <L1,L2>   (edit line L1, saving it as L2)

```
MOVE 1.1,1.2(crlf) (moves line 1.1 to 1.2 with no changes)
M 1.1,2.1 (cr;_lf) (moves all but the first command in line 1.1 to 2.1)
M (crlf)           (displays the header line: program name, date saved)
M,12.9 (crlf)      (copies the header line (line 0) to line 12.9)
```

MOVE is really an expanded version of MODIFY and operates in exactly the same way in terms of searching for a character, etc. The only difference is the second line number (separated by a comma) which becomes the line number of the edited line. The original line remains as it was. Although it is possible to create whole number lines in this way (i.e. 1.00) such lines cannot be individually erased and ought to be carefully avoided. When patched to print the line number, the new number will be shown. Turning off the input echo will disable the text output. This is useful for rapidly moving many lines. In the patched version the new line number is printed anyway.

N E X T

```
*NEXT [LI]        (initiates the next cycle of a FOR loop)

F I=1,10;N;T "?" (types -one- question mark)
F J=,9;D 2;N 3.3 (continues with line 3.3 following completion of loop)
```

The NEXT command allows nested (or sequential) loops to be coded on the same line. It permits direct commands to include operations both before and after a loop. It facilitates the conversion of programs written in other languages by offering the FOCAL programmer an easy, flexible way to escape past commands immediately following a loop. NEXT may occur anywhere in statements executed by a loop. Commands following NEXT will not be executed until the loop is satisfied. Thus NEXT either indicates the start of the next cycle of the loop, or the location of the next command following completion of the loop, with the option of branching to a specified line at that time. A NEXT command encountered outside of a loop will simply continue the program with the -next- command, as indicated! This allows FOR loops to reference immediately sequential lines (in the style of other languages) and then skip over those lines upon completion of the loop. For example:

```
1.1 FOR I=1,10;FOR J=1,10;DO 1.2;NEXT;DO 1.3;NEXT 1.4
1.2 TYPE A(I,J)
1.3 TYPE !
1.4 CONTINUATION OF THE PROGRAM
```

The 'NEXT 1.4' command in line 1.1 could also appear at the end of line 1.3 in which case a simple 'NEXT' would suffice. Of course this example (which prints out a matrix with a carriage return after each row) could, and normally would, be printed on a single line:

```
1.1 F I=1,10;F J=1,10;T A(I,J);NJ;T !
```

Here the nesting of the inner (J) loop is more obvious and in fact the abbreviation of the NEXT command has been chosen to make this as clear as possible. The following can now be written as a direct command:

```
O O FILE;   F I=1,100;T X(I);N;   O C
```

or two sequential loops may be combined on a single line:

```
F I=1,100;S X(I)=I;N;   F I=100,-1,1;T X(I),!
```

See also the BREAK command.

O N

```
*ON (E1)[G1,G2,G3]          (call subroutine G1, G2, or G3 as E1 is -,0,+)

ON (B*B-4*A*C)1,2,3         (calls either group 1, 2 or 3; then continues)
O (P-Q)P,,Q                 (call either P or Q - whichever is smaller)
O (X(I)-MIN)2.1             (calls line 2.1 if X(I) is less than MIN)
```

'ON' functions exactly like 'IF', except that DO-type branches are used - i.e. subroutine calls. Hence after executing the appropriate subroutine the program continues with the next command following the ON command. As with the improved 'IF' command, line numbers may be omitted if no branch is desired. This command is primarily useful within loops where several tests in a row must be performed. Use of the IF command in that case would terminate the loop after the first test. Be careful not to use excess right parentheses as they are not trapped as an error, but computed as zero, and hence no branch is taken! See also the restrictions on line-number expressions discussed under 'IF'.

P L O T

The PLOT command is installation dependent. An extensive set of incremental plotter routines are available - or this command may be used to drive an analog (XY) plotter. Routines for Tektronix terminals are also available - see the graphics section.

Q U I T

```
*QUIT          (terminates program execution)
Q              (abbreviation)
```

R E T U R N

*RETURN (causes escape from a subroutine to the command following the calling 'DO', 'ON', 'LIBRARY GOSUB' or 'FSF. Otherwise it functions the same as 'QUIT' and is recommended for this purpose so that a program may be run either independently or as a subroutine without change. Comments following a 'return' will be ignored.)

R (abbreviation)

S E T

```
*SET Y=<NUMBER, VARIABLE OR EXPRESSION>          (sets variable value)

SET Y=37            (causes 'Y' to take the value 37)
S A=110/P+32        (causes 'A' to assume the value of 110/P+32)
S Y=Y+1             (sets the new value of Y = 1 plus old value)
S X=FIN()           (sets X to decimal ASCII code of next character input)
S A=8,C=5,D=FITR(U/W)   (may be used instead of " S A=8;S C=5;S D=...")
```

Strings of SET commands may now be combined by separating each equality with a comma. This permits more efficient programming and also improves the readability of the program.

T Y P E

```
*TYPE [%-E1][:E2,][E3][$]          (output variables, symbol table, etc.)

TYPE "WAITING":-I   (creates a pause by reading (and ignoring) 1 char.)
TYPE %6,-6          (types "     -6", not " -     6" as in PS/8 FOCAL)
TYPE %0 or T %,     (sets format to full precision scientific notation)
T %-5 ?PI?          (types PI  3.1416E+00 correctly rounded to 5 places)
Z,X(-1);T $         (types X (-01) ..., not X9(<7) .. as in PS/8 FOCAL)
TYPE 3              (outputs a 3 on teleprinter or other output device)
T X/Y-1             (outputs the value of the expression X/Y-1)
T "A"!"B"!"C"!      (outputs 'A', 'B' and 'C' in a vertical row)
T !                 (outputs a return/line feed)
T "=",#,"/"         (prints = and / on top of each other)
T "ANS:",26+5       (outputs 'ANS:  31')
T :8,"A"            (outputs 'A' in position 8 on line)
T $                 (outputs the symbol table, 3 variables / line)
T %10.09            (sets the output format to allow up to 10 digits
                     to be printed, of which 9 may be decimal places)
T %10               (formats output as ten digit integers)
```

Aside from the extension of the tab command to include skipping over input as discussed before, the primary changes to TYPE are related to format improvements. First, the minus sign now "floats" to the position directly before the first digit rather than always appearing at the beginning of the output. Secondly, "the power of ten" format has been converted to proper scientific notation - a very desirable change! Thirdly, the number of digits printed in this format can now be controlled, complete with roundoff, by specifying a negative integer format. And finally, the symbol table dump now correctly prints single character names as well as subscripts greater than 99 or less than 0; it also prints several variables per line.

Normally TYPE $ is set to print 3 variables per line. Specifying 'TYPE $5' will list 5 across and change the default value to 5. If variables occur in pairs (e.g. X,Y), a value of 2 is rather convenient.

Note: a 'TYPE' command containing an '=' or an unbalanced right parenthesis will output an infinite number of zeros until interrupted by CTRL/F.

V I E W

```
*VIEW X,Y [,Z]             (CRT output)
```

The standard version is written for a VC8/e display control and can be easily patched for use with the PDP12. It is not possible to drive a refreshed display with this command, but it is quite useful in conjunction with an XY plotter or a storage scope. In particular VIEW has been implemented for use with many of the Tektronix graphics terminals - see the section on graphics options.

W R I T E

```
*WRITE [GROUP OR LINE NUMBER]     (lists program)

W 2                  (outputs group 2)
WRITE                (outputs the entire stored program)
W 5,1.1,9            (prints group 5,then line 1.1, then group 9)
F X=12,31;W X        (lists all program lines from 12.01 to 31.99)
```

WRITE no longer terminates the line as it did in earlier versions of FOCAL. This command provides the means for converting 'core-image' programs into ASCII data files so that they can be merged or sent to other installations. (See p. 9)

X E C U T E

```
*XECUTE [E1][E2,][ E3]            (evaluates one or more expressions)

XECUTE FIND(154)  (searches for CTRL/Z - used to move a file)
X FOUT(135) FIN() (rings the teletype bell, waits for one character)
X FSIN(#)/FCOS(#) (leaves the tangent of # in the floating accumulator)
X                 (clears the floating accumulator)
```

The XECUTE command allows as many expressions as will fit on a line to be evaluated without explicitly saving the results. Spaces or commas may be used to separate the expressions. This command is primarily intended for calling input/output control functions which do not return a useful numerical result. Another frequent use of the XECUTE command (as illustrated in the last two examples) is in conjunction with FOCAL Statement Functions (v.i.). Note that an unbalanced right parenthesis or an "=" sign in any expression will create an infinite loop. The TYPE command has the same problem due to a "quirk" of the arithmetic processor.

Y N C R E M E N T

Y X,P	(increments X and P, equivalent to SET X=X+1,P=P+1)
Y -X	(decrements X, equivalent to SET X=X-1)

This command is optional. It allows a list of variables separated by spaces, commas or minus signs to be incremented and/or decremented much faster and more compactly than they would be using the equivalent 'SET' commands. The 'Y' command also does not disturb the result of a FSF unless used with a subscripted variable; this is sometimes useful for 'post-indexing' operations. This command overlays the 'laboratory' functions unless there is additional core space available, for instance through the use of the EAE patches.

Z E R O

*ZERO [X,Y,Z]	(sets variables to zero or clears the symbol table)
ZERO	(clears the symbol table except for secret variables)
ZERO A,B,C	(defines three variables and/or sets them to zero)
Z,I,J,K	(places I,J,K at the beginning of the symbol table)
Z ! " # $ %	(sets all protected variables to zero)

Either a space or a comma may be used to separate the variable names. If no variables are specified, ZERO clears the symbol table, thereby effectively setting all variables to zero. ZERO replaces the old 'ERASE' command; it does not terminate the line. Z A is faster than SET A=0 and does not alter the floating-point accumulator, unless used with subscripted variables.

```
***** * * * * * * *   * * * * * * * *  *****
***** L I B R A R Y   C O M M A N D S  *****
***** * * * * * * *   * * * * * * * *  *****
```

L I B R A R Y   C A L L

```
*LIBRARY CALL [DEVICE:]<PROGRAM NAME>[SIZE]

LIBRARY CALL CHISQR      (loads program 'CHISQR.FC' for use)
L C DTA3:PRGRAM          (loads 'PRGRAM.FC' from DECtape #3)
L C TESTI                (loads 'TESTI.FC' from DSK:)
```

The 'L C' command returns FOCAL to 'command mode', hence commands following this one will never be executed.

L I B R A R Y   D E L E T E

```
*LIBRARY DELETE [DEVICE:]<PROGRAM NAME>[ ERROR]       (.FC is assumed)

LIBRARY DELETE TTEST     (removes 'TTEST.FC' from the directory)
L D DTA7:PROG            (removes 'PROG.FC' from DECtape #7)
L D HOLD.FD              (removes data file 'HOLD.FD')
L D TEST(I).FD 9.7       (branches to line 9.7 if TEST(I).FD is already
                          deleted)
```

This command closes open output files and sets the 'program not saved' flag. Hence programs containing 'L D' commands should not perform 'GOSUBS'. Note the use of a programmed error return.

L I B R A R Y   G O S U B

```
*LIBRARY GOSUB [DEVICE:]<PROGRAM NAME>[SIZE] [GROUP OR LINE NUMBER]

LIBRARY GOSUB TEXT 13.7 (line 13.7 of 'TEXT.FC' becomes a subroutine
                         which returns to the command following 'GOSUB')
                         (when a 'GOSUB' is executed by a new program,
                         the new program will be saved as 'FOCAL.TM')
L G SUMSQR               (treats entire 'SUMSQR.FC' program as subroutine)
L G CALC 7               (treats group 7 of 'CALC.FC' like a 'DO' subroutine)
L G <P(I)>[S(I)]         (calls the subroutine at block P(I) with size S(I))
```

This command closes open output files if given by an unsaved version of a program. See note above regarding use of 'L D' and 'L G' in the same program. Note that loading a new program or calling a subroutine, does not effect the variables.

L I B R A R Y   R U N

```
*LIBRARY RUN [DEVICE:]<PROGRAM NAME>[SIZE] [LINE NUMBER]

LIBRARY RUN JOHN          (loads 'JOHN.FC' then begins program execution)
L R DTA2:ZONK             (runs 'ZONK.FC' from dectape #2)
L R POP 22.81             (starts executing 'POP.FC' at line 22.81)
```

Transfers execution to named program. Programs containing 'RUNS' must be saved before execution.

L I B R A R Y   S A V E

```
*LIBRARY SAVE [DEVICE:]<PROGRAM NAME>

LIBRARY SAVE PROG         (saves the indirect program as 'PROG.FC')
L S DTA6:ZAAP             (saves 'ZAAP.FC' on dectape #6)
L S PRGNAM                (saves 'PRGNAM.FC' on DSK:)
```

Old 'PRGNAM.FC' is deleted when new 'PRGNAM.FC' is saved. Deleting the old program first will usually result in the new version being saved in the same location, unless it has gotten longer. This command closes open output files. There may be a programmed error return.

L I B R A R Y   N A M E

```
*LIBRARY NAME [PROGRAM NAME]  (inserts name and current date in header)

L N NEWONE       (puts the name "NEWONE" and the date in line 0)
```

Only the first six characters of the name are used; any device or extension given is ignored. The purpose of this command is to provide a way to fill in the comment line at the beginning of the program prior to making a listing. Thus 'LIBRARY NAME LSTSQR;WRITE' produces the line:

```
C U/W-FOCAL: LSTSQR 09/13/73
```

followed by the rest of the program. The LIBRARY SAVE command also enters this information before saving the program. Thus every time a program is loaded, the name and the date it was saved can be found in the header. Examining the header provides a quick way to find out which subroutine is in core when interrupting a program which uses the LIBRARY GOSUB command. An expedient way to do this is to type "M(CR)(LF)". This command also offers a convenient way to create program delays. For instance, 'L N (1E600)' will create approximately a 3-second delay with the interupt off. L N also sets the 'program not saved' flag, and thus should not be used before a 'GOSUB'.

L I S T   A L L

```
*LIST ALL [DEV:]        (lists a whole directory of the specified device)

L A                     (abbreviation)
L A LAST                (lists all files on DSK: following 'LAST.FC')
```

'LIST ALL' produces a listing showing only the name and length. The date and intervening 'empties' are not shown.

L I S T   O N L Y

```
*LIST ONLY [DEVICE:][FILENAME](prints the name & length of one program)

LIST ONLY DEMO  (looks up the program DEMO.FC and prints its length)
L O SYS:PROG01  (abbreviation - checks PROG01.FC on the system device)
L O PIP.SV      (prints the length of PIP.SV if it is found on DSK:)
L O             (lists all .FC files - same as LIBRARY LIST)
```

The L O command is quite useful for checking if a program with a given name already exists on a specified device since, if it does exist, only the information concerning that program will be typed out. If the program is not found, there is no printout.

L I B R A R Y   L I S T

```
*LIBRARY LIST [DEVICE:][FIRST FILE NAME TO BE LISTED]
                   (lists program and data file names)

LIBRARY LIST       (lists FOCAL '.FC' files saved on DSK:)
L L                (abbreviation)
L L DTA3:          (lists files saved on DECtape drive #3)
L L DTA6:TEST      (starts listing with 'TEST.FC')
L L .SV            (lists all files with the extension '.SV')
L L FOCAL.         (lists all files with the name FOCAL)
```

This command lists the length of any file with any extension specified. After listing the specified file (if any) only files with the same extension (or name if a null extension was specified) will be listed. The extension '.FC' is the default hence the command is usually used to obtain a program summary. However, the 'wild card' feature is equally useful for listing all 'temporary' files, or all forms of a program such as '.PA', '.BN', and '.LS'. See also 'LIST ALL', 'LIST ONLY' and 'ONLY LIST'.

L O G I C A L   B R A N C H

```
*LOGICAL BRANCH [L1] (continues program at L1 unless a TTY key is
                 struck, perhaps better remembered as 'LETS BRANCH')

LOGICAL BRANCH 1.1  (same as GOTO 1.1 unless any key on the TTY is hit)
L B 8+N/100;A :-1   (computed line numbers are acceptable, of course)
```

The logical branch command is a unique feature of U/W FOCAL. This command provides a way to use the teletype for program control without requiring continous responses from the operator. The most useful applications are in iterative calculations and real time control loops.

Normally the logical branch command simply branches to the designated line. However, if there is a character waiting in the teletype buffer, the branch is omitted and the program continues with the next command. This command can then input and test the character (using FIN), or else it can simply read and ignore it (using, say, ASK :-1). A third possibility is to do nothing, leaving the character in the buffer for later testing or input. Eventually whatever character was struck will be input, either by an ASK command (or FIN or FIND), or as the first letter of the next direct command when FOCAL returns to command mode.

Since there are 128 possible keyboard characters, the L B command effectively turns the teletype into a giant switch register. However CTRL/F and CTRL/C always cause an immediate interrupt and so are not useful for this purpose. LINEFEED and RUBOUT are especially convenient because they do not echo. Note that the L B command waits for all output from the teletype to be completed before testing the buffer so that decisions may be made on the basis of the results which were typed out immediately beforehand.

As a simple example of the utility of this command, consider a program which performs (say) a numerical integration and which requires 5-10 minutes of computation time. If this program is interrupted via CTRL/F at some point, it will, in general, be difficult to restart it again from the point where it was stopped. However, if a few L B commands are included at appropriate places, one can provide a way to allow the program to finish an entire cycle before responding to the teletype. Naturally these breakpoints should be chosen so that the program can be easily restarted.

As another example, consider an iterative program for finding the roots of a polynomial, or some other similar problem. Since the rate of convergence will probably not be known in advance, it is convenient to have the first few iterations typed out. Then, if convergence is slow, it is best to just let the program cycle without all the intermediate printout, except for an occasional check on the progress. The L B command is ideally suited for this sort of operation since it is very easy to check the keyboard just before the output routine and skip to the next cycle until a key has been struck. A similar check at the end of the printout can be used to decide whether or not to continue the iteration.

The only precaution necessary when using the L B command is that there be enough time between an input command and a L B command for a character to be sent from the teletype. The statement "ASK A;L B 1.1" will always branch (if the input is from the teletype) because there is no time after the ASK command to strike another key. However if the input were from a file this command would work properly. The L B command always tests the TTY, no matter which input device is selected.

Ex: (displays A/D input in the MQ) I.I X FMQ(FADC(FSR()));L B I.I;R

L O G I C A L   E X I T

*LOGICAL EXIT (leaves FOCAL; returns to OS/8 monitor)

L E (abbreviation)

This command is equivalent to CTRL/C, but may be built into the program. FOCAL may be restarted after a monitor exit by typing "START" (or just "ST") and hitting 'RETURN'.

O N L Y   L I S T

*ONLY LIST [DEVICE:][FILENAME] (prints the name and length of one file)

```
ONLY LIST DATA   (looks up and prints the length of the file DATA.FD)
O L LTA1:FILE01  (abbreviation - checks FILE01.FD on LINCtape 1)
O L              (lists all .FD files)
```

This command is the obvious counterpart of the LIST ONLY command; The difference is just that an extension of .FD rather than .FC is assumed. If an extension is given as part of the filename both commands are identical. ONLY LIST is convenient for checking the length of a data file since it is not necessary to specify an extension and only that one file will be listed, even if there are other FOCAL files present on the same device.

```
*****  * * * * * * * * * * * *   * * * * * * * *   *****
*****  I N P U T / O U T P U T   C O M M A N D S   *****
*****  * * * * * * * * * * * *   * * * * * * * *   *****
```

O P E N   I N P U T

```
*OPEN INPUT [DEVICE:][FILE NAME][,ECHO][ ERROR]
                              (prepares to read a file)

OPEN INPUT BLEEP            (switches input to 'BLEEP.FD' file on DSK:)
O I DTA4:RED 6.71           (opens 'RED.FD' file on DECtape drive #4,
                             branching to line 6.71 if the file is missing)
O I TABLE,ECHO              (will echo on output device while reading
                             'TABLE.FD' from the DSK:)
OPEN INPUT TTY:,ECHO        (restores terminal to normal function)
O I,E                       (short form)
O I                         (disables TTY echo for 'silent' typing)
```

CTRL/Z is the end-of-data-file character. Attempts to read past it will output a '_' (shift/O, back arrow) and switch I/O back to the terminal.

O P E N   O U T P U T

```
*OPEN OUTPUT [DEVICE:][FILE NAME][SIZE][,ECHO][ ERROR]
                                   (prepares to write a file)

OPEN OUTPUT OK             (opens 'OK.FD' to be written on the DSK:)
O O DTA7:ZZ[10] 11.7 (opens 'ZZ.FD' file to be written on DECtape)
O O DOPE,ECHO              (echoes on TTY: while writing 'DOPE.FD' on DSK:)
                           (be sure to give an 'OUTPUT CLOSE' when done)
O O TTY:                   (returns output to the terminal)
O O                        (short form)
```

Open output files will be closed if the commands 'LIBRARY SAVE' or 'LIBRARY DELETE' are given or if 'LIBRARY GOSUB' is given by a version of a program that has not been saved. The error return occurs if insufficient space is available or if a file is already open on the same device, or if only the device name is given with a file-sturctured device.

O P E N   R E S T O R E   I N P U T

```
*OPEN RESTORE INPUT[,ECHO][ ERROR]     (resumes input from non-TTY:)

OPEN RESTORE INPUT     (resumes 'asking' for data from a previously
                        opened input file after using TTY:
                        input with an 'OPEN INPUT TTY:,ECHO')
O R I                  (abbreviation)
O R I,E 15.1           (same plus echo and error option)
```

O P E N   R E S T O R E   O U T P U T

```
*OPEN RESTORE OUTPUT[,ECHO][ ERROR]    (resumes output to non-TTY:)

OPEN RESTORE OUTPUT    (resumes 'typing' on previously opened
                        output device after using TTY:)
O R O                  (abbreviation)
O R O,ECHO             (same plus echo)
```

There can be a programmed error return.

O P E N   R E S T A R T   R E A D

```
*OPEN RESTART READ [,ECHO] (restarts input from the first block of the
                            last file)
O R R                  (abbreviation)
O R R,E                (same plus echo)
```

This command saves a great deal of time when several passes on the same data are required, or when restarting an interrupted program. For non-file structured devices, like the PTR: or CDR: use of the O R R command causes input to commence immediately (it may be different, of course). Even if the file is deleted, O R R will return to the first block.

O U T P U T   D A T E

```
*OUTPUT DATE           (prints the system date in the form MM/DD/YY or
                        DD/MM/YY, installation dependent)
O D                    (abbreviation)
```

The current date should be entered with the monitor DATE command before loading FOCAL; it can be changed with the FDAY function. The date also appears in the header. If no date has been entered, the message 'NO/DA/TE' appears instead. This command is especially useful in data collection and analysis programs.

O U T P U T   C L O S E

```
*OUTPUT CLOSE [E1]  (ends file writing and saves output file if device
                     is file structured (disk or magnetic tape).)
O C                 (abbreviation - 'O C' restores the 'TTY:' as the
                     output device. Ignored if there is no file open)
O C Q               (closes file and enters length of Q blocks in
                     the directory)
```

The closing length is optional and if omitted (or equal to zero) the actual length will be used. This provision is offered to facilitate adding information to a file without moving it to a new location. An error will occur if the length requested exceeds that available, but no error is detected if the requested length is smaller than the actual length. Use 'FLEN' to keep track of the size.

O U T P U T   B U F F E R

```
*OUTPUT BUFFER      (dumps the contents of the output buffer)

O B                 (abbreviation - ignored if there is no file open)
```

This command may be used to create line-oriented output on non-file structured devices such as the LPT: and TV:. The 'O B' command simply dumps the buffer contents without removing the handler, thus additional output may be created immediately: O O TV:;T PI;O B;T FEXP(1);O O;O B

O U T P U T   A B O R T

```
*OUTPUT ABORT [E1]      (closes an open file immediately)

O A              (abbreviation; ignored if there is no file open)
O A FLEN()-10    (discard output file leaving a 10-block 'hole')
```

This command provides a way to end a file without making it permanent. It could be used, for instance, if a file were inadvertently opened with the wrong name. It can also be used in connection with FLEN to determine the amount of space available on a device. The most common use, however, is in data collection programs as a way to quickly terminate a partial file because of some interruption which invalidates the previous results. Without this command it is necessary to first close a useless file and then delete it which takes considerable time on a DECtape system. Another important use takes advantage of the optional length parameter. If a non-zero length is specified the file will be left in the directory with this length, although no output need have occured. This permits 'fixing' a directory after a program or file has been 'deleted' as well as establishing in advance file areas for use with 'FRA' or for other purposes such as forcing seldom used files to the end of a tape.

A file handling program from the OMSI manual:

```
12.10 C-SETUP OUTPUT FILE (.FD IS ASSUMED EXTENSION, DSK: THE DEVICE)
12.15 TYPE "LINE 12.20 WILL NOW OPEN 'NUMBRZ' FILE AND WRITE IN IT"!
12.20 OPEN OUTPUT NUMBRZ
12.45 FOR I=1,10;TYPE %3,I,!
12.50 COMMENT-NOW SAVE OUTPUT FILE AND RE-STORE OUTPUT TO TTY:
12.60 OUTPUT CLOSE
12.70 TYPE "LINE 12.60 JUST CLOSED THE 'NUMBRZ' FILE"!

13.10 TYPE "LINE 13.20 WILL NOW OPEN THE 'NUMBRZ' INPUT FILE"!
13.20 OPEN INPUT NUMBRZ
13.30 TYPE "LINE 13.50 WILL NOW READ IN NUMBERS AND COMPUTE ROOTS"!
13.50 FOR I=1,10;ASK A;TYPE !"ROOT",%2,A," IS ",%5.04,FSQT(A)
13.60 TYPE !!"LINE 13.70 RESTORES INPUT TO TERMINAL"!
13.70 OPEN INPUT TTY:,ECHO
13.80 TYPE "LINE 13.90 WILL NOW DELETE 'NUMBRZ.FD' FROM THE DIRECTORY"!
13.90 LIBRARY DELETE NUMBRZ.FD
```

simulated execution of the program:

```
LINE 12.20 WILL NOW OPEN 'NUMBRZ' FILE AND WRITE IN IT
LINE 12.60 JUST CLOSED THE 'NUMBRZ' FILE
LINE 13.20 WILL NOW OPEN THE 'NUMBRZ' INPUT FILE
LINE 13.50 WILL NOW READ IN NUMBERS AND COMPUTE ROOTS

ROOT   1  IS   1.0000
ROOT   2  IS   1.4142
ROOT   3  IS   1.7321
ROOT   4  IS   2.0000
ROOT   5  IS   2.2361
ROOT   6  IS   2.4495
ROOT   7  IS   2.6458
ROOT   8  IS   2.8284
ROOT   9  IS   3.0000
ROOT  10  IS   3.1623

LINE 13.70 RESTORES INPUT TO TERMINAL
LINE 13.90 WILL NOW DELETE 'NUMBRZ.FD' FROM THE DIRECTORY
```

**** G R A P H I C S   R O U T I N E S ****

I. TEKTRONIX TERMINALS

The following commands and functions are available for Tektronix terminals types T4002, T4010, T4012, etc.:

1. Alphanumerics

These terminals can be used as normal teletypes and respond to all the commands in the same way that a normal teletype would. In addition the following characters are available:

```
X FOUT(8) or FOUT(136) backspace one character (CTRL/H)
X FOUT(11) or FOUT(139) 'line up' (CTRL/K)
```

2. Erase screen

The sequence X FOUT(27) FOUT(12) or X FOUT(155) FOUT(140) clears the screen. (ESCAPE, FORMFEED or Ctrl/Shift/K, Ctrl/L)

3. Hardcopy

X FOUT(27) FOUT(23) or X FOUT(155) FOUT(151) produces a hardcopy if the hardcopy unit is available.

4. Graphics

The VIEW command has been implemented for point-to-point plotting on these terminals. The command has the form 'VIEW X,Y,Z' where X is an expression for the horizontal coordinate and Y is the value of the vertical coordinate. The X range is 0-1023 and the Y range is 0-780. Z controls the beam intensity. If Z is negative the point at (X,Y) is intensified. If Z is zero, the beam is simply positioned at (X,Y) and if Z is positive, a bright vector is drawn from the current position to the point specified. If X, Y or Z are omitted a value of '0' is assumed. The following routine will draw a circle in the middle of the screen:

```
1.1 X FOUT(27) FOUT(12);SET R=200,X=512,Y=390;V X,Y+R
1.2 FOR A=PI/25,A,2*PI;VIEW R*FSIN(A)+X,R*FCOS(A)+Y,1
1.3 VIEW X-17*7+2,Y-9;TYPE "THIS IS A CIRCLE!";VIEW
```

As illustrated above, TYPE and VIEW may be freely intermixed to provide annotated displays. The letters are 10(x) by 18(y) with 4 additional spaces on the top and right-hand side (14x22 total).

5. Cursor or joystick.

Most models are equipped with a cross-hair cursor or (originally) a joy stick control. This allows user input of graphic as well as keyboard information. The FJOY function may be used to read either the current beam position or else the cursor location. The coordinates are returned in the variables 'XJ' and 'YJ'.

Note: the terminal response to the FJOY function is dependent upon one of the 'strappable options'. For best results the choice 'no terminator' for cursor enquiries should be selected. Operation with the CR terminator (the 'normal' position) is possible, but requires a patch to FJOY.)

To write some text on the screen and then underline it, the FJOY function is called with a NON-ZERO argument. The variables 'XJ' and 'YJ' return the beam position.

```
        2.1 S X=200,Y=700;V X,Y;T "A LINE OF TEXT"
        2.2 X FJOY(1);VIEW X,YJ-10;VIEW XJ,YJ-10,1
```

If the argument is ZERO, the cursor appears on the screen. Typing any character on the keyboard will put the coordinates of the cross in XJ and YJ. The typed character is returned by the function as its decimal equivalent, suitable for use by IF, JUMP and the like. The example program 'SKETCH' from the OMSI manual can now be coded:

```
3.1 X FOUT(27) FOUT(12);C 'SPACE' FOR BRIGHT LINES, 'RUBOUT' FOR DARK
3.2 SET Z=192-FJOY();VIEW XH,YH;VIEW XJ,YJ,Z;S XH=XJ,YH=YJ;G 3.2
```

6. Another example:

```
01.10 C WEB BY BARRY SMITH, OMSI SOFTWARE DEVELOPMENT GROUP
01.11 C LAST CHANGE: 4/15/76 BY J. VAN ZEE FOR U/W FOCAL
01.12 C ** TEKTRONIX T-4002 OR 4010 GRAPHIC TERMINAL REQUIRED **
01.13 C THIS PROGRAM DRAWS A SPIDER AND HER WEB.  IT IS NOTHING
01.14 C MORE THAN A HANDY DEMONSTRATION OF FOCAL'S DISPLAY POWER.
01.15 C PLEASE DO NOT COUNT THE NUMBER OF LEGS - I DO KNOW BETTER!
01.20 X FOUT(27) FOUT(12);TYPE :12 "A SPIDER AND HER WEB"

02.10 FOR A=,30,330;SET B=A/57.295,BS=FSIN(B),BC=FCOS(B);DO 2.2
02.20 VIEW 300,400;VIEW 300+BS*300,400+BC*300,1;NEXT

03.10 SET D=1.5;FOR A=,30,3500;SET B=A/57.295;DO 3.2
03.20 VIEW 300+FSIN[B]*D,400+FCOS[B]*D,1;SET D=D+2.5

04.10 VIEW 175,180;FOR A=,45,360;SET B=A/57.295;DO 4.2
04.20 VIEW 250+FSIN[B]*10,170+FCOS[B]*15,1
04.30 VIEW 220,210,1;VIEW 190,180,1
04.40 VIEW 250,185
04.50 VIEW 235,220,1;DO 4.4;VIEW 300,160,1;DO 4.4
04.60 VIEW 275,225,1;VIEW 300,210,1;DO 4.4;TYPE "*";VIEW
```

II. INCREMENTAL PLOTTERS

A comprehensive set of routines for controlling an incremental plotter (Calcomp, Houston Instruments, Bensen, Zeta, etc.) is available for either the new XY8/e interface or the older XY8I, XY12 or 350B types. The plotter is run under interrupt control so that program execution may proceed in parallel with plotter operations to some extent.

The two additions to U/W-FOCAL implemented by these routines are the 'PLOT X,Y,L,M' command and an internal handler called PLTR:. The PLOT command provides control of the pen position for drawing curves and marking data points while the PLTR: handler is used in conjunction with any of FOCAL's regular output commands for adding annotation. The routines require approximately 6 pages of core and hence work best with a 12k or larger machine although a somewhat compromised 8k version is possible.

1. PLOTTING:

The PLOT command moves the pen to the (X,Y) position indicated by the first two parameters. The coordinates are usually specified in -inches- or -centimeters- depending upon the type of plotter, however both values are multiplied by a scale factor variable SF so the programmer can use any set of units he finds convenient. The plotter, of course, must be driven a step at a time and this leads to a minor limitation on the magnitude of the coordinates. Since at most 4095 steps can be counted without an overflow the maximum coordinates for a plotter with a 10 mil (0.010") step size are approximately +/- 20 inches. In fact, the maximum -difference- between any two points is also restricted to this value so that on really large plots two calls might be required to reach a point. For normal plotting on 8-1/2 x 11 inch paper this restriction is almost never apparent.

Although these routines were originally developed for a 'low-resolution' plotter with a 10 mil increment, they are easily adapted (with a 2-word patch) to work with higher resolution plotters having either a 5 mil (0.1mm) or 2.5 mil (0.05mm) step size. The X,Y coordinates are still expressed in inches (or centimeters) and the maximum motion (except for the 2.5 mil unit) remains slightly over 20 inches, but the effective grid resolution is limited to 10 mils (0.25mm). While this reduces the absolute positional accuaracy by a factor of 2 or 4, the quality of the lines will be that associated with the smaller step size. The sample plots included in this manual were prepared on a such an instrument.

The 'L' parameter in the PLOT command determines the type of line used to connect two points. If L=0 the pen is raised during the motion, thus -no- line is drawn. If L=1 the pen is lowered onto the paper. Although not yet implemented, it was planned to use L=2,3,4... for creating various dotted and dashed lines. L=-1 moves with the pen up to the indicated position and then resets the location counters so that this point becomes the new origin. This feature is needed for moving from one plot to the next and is also used for initializing the margins on a new plot.

The 'M' parameter determines the kind of Mark which is drawn at the conclusion of the move. M=0 draws -no- mark while M=1 makes a small dot and M=2-16 draws marks of increasing complexity. These symbols are intended primarily for identifying data points but they are also convenient for adding 'tick' marks along an axis. Both open and filled symbols are available and each one can be drawn in 8 different orientations and in 2 different sizes. This generally provides an ample choice of identifiers - see the sample plot at the end of this section.

Not all four parameters are required in every PLOT command. Those which are omitted will be given the value '0'. Thus many PLOT commands specify only the X,Y values and sometimes L. Each parameter may, of course, be replaced by an arithmetic expression. This is most common for X and Y which are usually linear transforms of other variables.

2. ANNOTATION:

In addition to drawing curves and axes it is also necessary to provide labels for the axes and other forms of annotation. This is accomplished by switching output to the PLTR: with an OPEN OUTPUT PLTR: command and then using FOCAL's normal output commands such as TYPE, OUTPUT DATE, FOUT, etc. to generate the output. Note that the PLTR: handler is not a regular OS/8 system handler; it exists only within FOCAL. Because of this it is not necessary to CLOSE output sent to the plotter since that operation is specific for the system handlers. To switch to another output device simply use another 'O O' or 'O R O' command. Another minor difference is that the ECHO option is not supported by the PLTR: handler.

Writing on the plotter is thus essentially identical to writing on any other output device. However, there is obviously much more flexibility in the placement and size of the characters and this flexibility is controlled by the values of 4 special plotting variables. These all begin with a '$' and one of them, $F, has already been mentioned in connection with the PLOT command. $F is the Scale Factor for the X and Y values. Normally this variable is set to unity, but the user may change this at any time. In fact, $F may even be negative which creates an inverted axis system.

The other 3 variables are $S, the Symbol Size; $D, the Symbol Direction; And $R, the Scale Rotation. A '$' is used, rather than an 'S' to avoid any conflicts with user variables. These variables may be changed at any time by the users program; they may be used in loops and for computations, and may be examined in various ways. But since they all begin with a '$' they must treated like the 'secret' variables when TYPEing their values. Note however, that these variables are -not- protected and will disappear after a ZERO command. They will then be redefined with their default values by the next PLOT command. Except for $F, all default values are zero, although this is easily changed should other values prove more convenient.

The size of the characters drawn by the plotter output routines (but not the size of the marks!) is determined by the value of $S. This size is specified in integer multiples of the basic character grid which is 6 units wide(x) by 10 units high(y). The units are either 10 mils or 0.25 mm, depending upon the type of plotter. Of this 6x10 area, only 4x6 is actually filled by the character so that space is automatically left between the symbols. Setting $S=2 (about the smallest practical value) would thus make characters 0.08" by 0.12" in size with 0.04" spacing horizontally and 0.08" vertically (and corresponding values for metric plotters). All printable ASCII characters except the '@' symbol are available, plus CR and LF. Other control characters (such as Form Feed) will be converted to letters.

The orientation of -both- the marks and letters is controlled by the value of $D. There are 8 distinct directions determined by the major plotter motions. These directions are always -relative- to the absolute orientation of the axis system (discussed below) and follow the convention that Direction 0 is along the positive Y axis (assuming $F to be positive). Direction 1 is then along the diagonal in Quadrant I while Direction 2 is along the positive X axis, and so on, around the clock. Negative values for $D simply represent counter-clockwise rotations. Thus $D=-2 will produce upside-down lettering along the -X axis.

An interesting pecularity of the plotter is that annotation along the diagonals is larger by a factor of the SQRT(2) (about 40%) due to the greater effective step size. This has the useful result of creating both big and little marks simply by rotating them 45 degrees. On the few occasions when lettering along a diagonal is required it is a simple matter to reduce the size appropriately to compensate for this geometrical effect.

Finally we consider the role of the variable $R, the Scale Rotation. This variable determines the absolute orientation of the axis system with respect to the paper feed direction. From ancient times the rule has been that the X axis lies along the direction of paper advance while the Y axis lies along the width of the paper. If $R=0 this convention will be observed. However, many (if not most) users like to watch the plots as they are made and furthermore, like to take advantage of the 8-1/2 x 11 inch fanfold paper now in common use by placing the X axis along the width of the paper with the Y axis in the direction of the paper feed. This represents a 90 degree rotation and is accomplished by setting $R=2. No other changes need to be made to the program: X and Y will retain their usual meanings and the annotation and marks will all be turned appropriately. The only programming which does depend specifically upon the value of $R is the initialization code which runs the pen off-scale and the command which advances the paper between plots. These statements must therefore examine the value of $R and move accordingly.

It is possible to change the value of $R during a plot in order to simplify the drawing of a highly symmmetric figure, but one must ensure that the pen has been brought to the origin first as this is the only point not transformed by the rotation. A similar problem occurs during annotation since the value of $D is used as a temporary rotation. Thus it is necessary that the pen always be returned to the position at the start of the annotation before PLOT can move to a new location. This action is automatic and need not concern the user, but is explained here in case it appears puzzling. It is also for this reason that new values assigned to the plotter variables will -not- be used until -after- the next PLOT command. This peculiarity is seldom noticed except in direct commands when one forgets to include a PLOT call after changing, say, the size of the letters.

As with any new device or program, some experience is necessary to gain a complete understanding of the way it works. While study of the sample plots is useful, it is highly desirable that new users try out a few simple operations before coding a complex plotting program. One of the nicest features of 'on-line' plotting with FOCAL is the ease with which a program can be modified to adjust, say, the position of a label or to change the scale on an axis. Programmers who have never had the experience of typing a direct command and seeing the result appear immediately on the plotter are in for a real treat.

3. Examples.

a. `P;F $R=,7;P 1,,1;P,,1;C DRAWS ALL 8 BASIC VECTORS`

b. `S $R=2,$D=2,$S=3;P;O O PLTR:;O D;O O;C PRINTS THE DATE`

c. `P;S $R=2;F $D=,7;F M=,18;P M/4,-$D/2,,M;ROTATE ALL MARKS`

d. `C U/W-FOCAL:  XYAXIS  04/15/76`

```
    01.10 Z;S $R=2,$S=2;P -12,,-1;P .75,,-1
    01.20 O O PLTR:;T %2;F X=1,10;P X,,1,3
    01.30 S $D=2;P,,1;F X=,10;P X-.4,-.2;T X
    01.40 S $S=3;P 4,-.5;T "X  A X I S"
    01.50 S $S=2;P;F Y=1,6;P,Y,1,3
    01.60 P,,1;F Y=,6;P -5,Y-0.6;T Y
    01.70 S $=3,$D=0;P -.2,2;T "Y   A X I S"
```

e. `C J/W-FOCAL:  PLOTER  04/15/76`

```
    01.10 C FOCAL PLOTTER DEMOSTRATION PROGRAM  -JVZ-
    01.20 Z;P,-12,-1;P,.2,-1;O O PLTR:;S $D=2,$S=4
    01.30 P,2.5;T "ASCII";P,2;T "SYMBOLS:";S $S=5
    01.40 P 2,2.5;T "ABCDEFGHIJKLM"!"NOPQRSTUVWXYZ"!
    01.50 T "0123456789-+="!".,:;!?()[]<>"!"" *%&$/\^ #"
    01.60 T #:2:FOUT(162):10:FOUT(223);S $S=4;P,.03
    01.70 T "MARKS:";S $D=-2;F M=1,18;P 1.8+M/4,.15,,M
    01.80 S $S=2,$D=2,X=6.25,Y=2;P 6.35,2.35;T "PLOTTER"
    01.90 P 6.53,2.1;T "UNIT";P X,Y;P X,2.6,1;P 7.25,2.6,1

    02.10 P 7.25,Y,1;P X,Y,1;P 6.35,Y;P 6.35,.25,1;P 7.15,.25,1
    02.20 P 7.15,Y,1;P 6.96,1.8;T "X";P 6.96,1.63;T "^"
    02.30 P 7,1.65;P 7,1.45,1;P 6.8,1.45,1;Z $D;P 6.82,1.41;T "^"
    02.40 P 6.62,1.41;T "Y";P 6.62,.7;T "PAPER";P 7,.7;T "FEED"
    02.50 S $S=3,$D=6;P 7,.5;F M=,2;P 7-M/5,.5;T "^"
    02.60 P 6.81,3.1;T "^;S $D=2,Y=3.15;P X,Y;T "SIZE 2"
    02.70 P 4.44,2.92;T "^";P 4,Y;T "SIZE 3";S $D=6
    02.80 P 2.55,3.1;T "^";S $D=2;P 2,Y;T "SIZE 5";S $D=6
    02.90 P .6,3.1;T "^";S $D=2;P,Y;T "SIZE 4"

    03.10 S $D=1;P 3.6,7.1,,16;S $S=5,$D=6;P 3.2,7.25
    03.20 T "DIRECTION 0"#:11"/;S $S=4,$D=7;P 3.44,7.5
    03.30 T "DIRECTION 1";S $S=5,$D=0;P 3.75,7.5;T "DIRECTION 2"
    03.40 S $S=4,$D=1;P 4,7.25;T "DIRECTION 3";S $S=5,$D=2
    03.50 P 4,6.95;T "DIRECTION 4";S $S=4,$D=3;P 3.75,6.7
    03.60 T "DIRECTION 5";S $S=5,$D=4;P 3.45,6.7;T "DIRECTION 6"
    03.70 S $S=4,$D=5;P 3.2,6.95;T "DIRECTION 7";S $D=2;P;O I
    03.80 I (141-FIN())3.9,1.3;P 9,,-1;C HIT 'RUBOUT' TO STOP
    03.90 O O;O I,E;C 'LINEFEED' TO ADVANCE, 'RETURN' TO REPEAT
```

```
*****  * * * * * * * * *  *****
*****  F U N C T I O N S  *****
*****  * * * * * * * * *  *****
```

One of U/W-FOCAL's major advantages is its expanded function table. Many programmers, particularly those in a laboratory environment, have often wanted to add dozens of new functions in order to be able to interact with all of their instruments. Consequently the limitation of earlier versions of FOCAL to a maximum of 15 functions was often a serious handicap. One could, of course, use a single function name for many different purposes, differentiating each use with a numerical parameter, but while this -does- work, it is very inconvenient since after a short lapse of time it is usually impossible to remember how 'FNEW(1)' differs from 'FNEW(2)'. The use of distinctive names such as FDVM and FDAC, on the other hand, makes the program much more self-documenting. With this sort of philosophy in mind, the function table has been expanded whenever possible, finally reaching its present level of 36 functions. Of these, only about 20 are in use, leaving lots of room for new additions.

Programmers who are adapting functions written for earlier versions of FOCAL should consult the section on adding user functions for some helpful hints. They will usually find that U/W-FOCAL has many internal routines which simplify the code for such functions.

The 'standard' functions are discussed below. Note especially that all 'arithmetic' functions have been upgraded to '10-digit' accuracy, consistent with FOCAL's extended precision arithmetic package. For functions employing a series approximation, this means, typically, that the tenth digit may be in error by, say, +/- 3, although the error is often much smaller.

FABS( ) absolute value

```
03.40 TYPE FABS(-3),FABS(2),!!!!
   3.0000   2.0000
```

FADC( ) analog to digital input function

```
04.22 SET X=FADC( )
```

This function works for the LAB8/e, PDP-12, and is easily patched for others. The argument selects the input channel.

FATN( ) arctangent. Principal range -PI/2 to +PI/2

```
05.25 TYPE 4*FATN(1),!!;C 4*THE ANGLE WHOSE TANGENT IS 1 SHOULD BE 'PI'
   3.141592653E+00
```

Note that FATN now returns full precision (10-digit) answers.

FBLK( )   Starting block number.

FBLK returns the starting block number of the current input file. Thus the location of any file (ASCII or otherwise) may be determined by using the OPEN INPUT command to read the directory. Remember to switch the input back to the terminal following the 'O I' command unless you actually wish to use that file for input. The following method can be used, for instance, to locate FOCAL programs. This is useful for later calls by 'direct access'.

```
   O I  PROG.FC;O I,E;SET BN(I)=FBLK(),SZ(I)=FLEN(I)
```

Note that an explicit extension must be given for programs since the 'O' commands assume an extension of '.FD'.

FBUF( ) Accesses the display buffer

This function operates like FCOM (see below) to store or retrieve 12-bit signed integer values in the display buffer. It is not a standard function.

FCOM( )   core storage function

This function provides access to a storage area apart from the regular variables - an area which is frequently undisturbed when FOCAL is loaded so that values placed there may be used at a later time. (This is not guaranteed since other system programs may use this area). Normally this area is shared with the program so that the user can trade off small programs with large arrays and vice versa. However those users with more than 16k can add a short patch to utilize Fields 4-7 providing space for up to 4096 values. The discussion below, however, assumes a smaller machine (12 or 16k).

FCOM may be called with either 1 or 2 arguments. The first is the storage index which serves as the subscript. The index may be either positive or negative and determines the mode of variable storage as well as the location. Positive numbers (0-ca.900) reference 4-word floating point numbers while negative indices (range -1 to ca.-1800) reference double precision integer values in a manner compatible with the double precision KE8E EAE instructions. Using integer mode storage obviously doubles the size of the array but does require data in the range +/-2^23. Since there is only one storage area the programmer must remember that FCOM(-1) and FCOM(-2) occupy the same core locations that FCOM(0) uses. (Note: the 'negative index' feature does not apply to the extended memory version mentioned above. Only floating point storage is available in that case.)

To store a value in the FCOM array, the function is called with that value (or expression) as the second argument. Thus 'X FCOM(I,PI)' will store PI at the 'Ith' location and 'T FCOM(I)' will print it out. FCOM may be called recursively - i.e. it may be used as a function of itself; In fact, recursive calls are frequently necessary when shifting parts of an array from one location to another:

```
F I=,199;S X(I+1)=FCOM(I+200,FCOM(I));X FCOM(I,X(I+1))
```

Since FCOM uses random access while the regular variables use a table search, FCOM not only provides more efficient storage, but faster retrieval as well. On the other hand FCOM is not quite as convenient as the regular variables since there is no provision for double subscripting and values can not be input with an ASK statment.

FCOS( ) cosine

```
06.27 TYPE %,FCOS(1),FCOS(3/2)!!;C OUTPUTS COSINES OF 1 & 1.5 RADIANS

     5.403023059E-01  7.073720172E-02
```

Note: The series approximation has been optimized for U/W-FOCAL by a least-squares fitting procedure.

FDAY( ) OS/8 date function

FDAY requires at least 12k; it allows the programmer to change the OS/8 system date as well as the text printed by the OUTPUT DATE command and the date saved in the header. The function always returns the value 0 and the argument must be the coded date as shown in the example below:

```
1.1 ASK "DATE? "MO,DA,YR :FDAY(MO*256+DA*8+YR-70)
```

Input for this example might be "7/15/76" using the "/" operator as a terminator. The ordering of month and day may be changed if desired. Note the use of a tab expression to call the function.

FDIS( ) installation dependent display function.

One version displays the contents of the buffer area filled by the FBUF function. The display time is determined by the argument: FDIS(20) displays for aproximately 2 seconds, FDIS() displays until a key is struck on the terminal. In this case the value returned by the function is the decimal value of the ASCII character code.

FEXP( ) exponential

```
08.24 TYPE FEXP(1),FEXP(2.17),!; C  NATURAL BASE TO THE POWER (X)

     2.718281829E+00   8.758284043E+00
```

FIN( )   character input function

Reads a single character for the input device. Ex: S X=FIN()

This sets X to the decimal ASCII code of the input character. See the example on p. 9 and also Appendix I.

FIND( ) character input function

example: 09.30 X FIND(Q)

If Q=193 then U/W-FOCAL will input characters until an "A" (code 193) is found. If the input is being echoed, all characters will be echoed up to, but not including, the target character. The value returned by FIND( ) is obviously that of the search character. If it is desired to have that character echo too, a command similar to the following should be employed: XECUTE FOUT(FIND()). Note the convenience of the XECUTE command in this case since no numerical result is required. The reason for not echoing the search character is to permit FIND to be used for editing and for merging files. This last operation is performed by searching for the end of file mark, CTRL/Z (code 154). Since this character will not echo it is possible to open another input file and add it to the previous one. The OUTPUT CLOSE command will then place a CTRL/Z at the end of the merged file.

The command X FIND(141) is a very simple way to input a comment line; it reads (and echos) all the characters up to the first carriage return. Finally, this function may be used to skip unwanted input in much the same way that the "negative tab" feature is used; If a simple search character exists (such as an "=" sign) this method is more desirable since no character count is required.

The argument must be in the range of legal decimal character codes as shown in Appendix I.

FITR( ) integer part

```
09.18 TYPE FITR(3/2),FITR(23.719),FITR(-3.999999999),!

     1.0000  23.0000  -3.0000
```

FJOY( ) joystick (cursor) position

Joystick or cursor on a Tektronix display terminal. The function returns the values of the X and Y coordinates in the variables XJ and YJ and the value of the key used to initiate the reading as the value of the function. See Appendix I for a list of decimal ASCII character codes and the graphics section for more discussion of FJOY.

FLEN( )   file length function

FLEN is used to determine either the number of free blocks remaining in an output file or the size of an input file. The choice is made by the argument: 0=output length, 1=input length. The later information is useful in connection with file size specifications. The following statement, for example, may be used to determine the size of the largest 'empty' on any device:

```
     OPEN OUTPUT DUMMY;OUTPUT ABORT;TYPE FLEN().
```

FLOG( ) natural logarithm

```
10.14 TYPE FLOG(1),FLOG(4.237),FLOG(10),!

     0.000000000E+00   1.443855472E+00   2.302585093E+00
```

- - - - 10-LOGARITHM, FL10 is not available - use a F.S.F.

Compute by multiplying FLOG()*0.4342944819, or FLOG()/FLOG(10).

FMQ( ) displays a number in the MQ register.

This works on all the later machines as well as the PDP-12 (with a 2 word patch), or on any machine with the EAE option. However, some of the EAE options do not save the MQ so the value loaded may be destroyed. The following line displays a selected channel of the ADC, allowing it to be adjusted to zero:

```
1.1 X FMQ(FADC(FSR()));L B 1.1;R
```

FOUT( ) character output function

example: 09.25 S X=FIN();X FOUT(X)

Outputs the character whose code equals X. If an 'L' were typed in response to a FIN() request, X would be set equal to 204. Then line 9.25 would cause an 'L' to be output. The value of FOUT() is always zero. This permits sending control characters as part of a command. For example: 'W FOUT(140)' is equal to 'W 0'. This sends a form-feed to the output device, then writes out the entire program. By placing FOUT in a tab expression the function can be evaluated without printing the results, since it is impossible to tabulate to column zero.

example: T "PRINT A ":FOUT(162)" MARK", produces: PRINT A " MARK

FRA( ) random access data storage

This function provides FCOM-like access to data stored in binary form on any mass-storage device. Several data modes are available: single word (signed and unsigned), double precision and floating point. The file used by FRA should first be looked up using the 'OPEN INPUT' command. Then 'FRA' must be initialized so that the necessary pointers can be transfered and the data format selected. The following types of calls are permitted: (I is non-negative, V is any expression). Note: the last two forms always return the value 0.

FRA(I)	read the I-th value
FRA(I,V)	stores and returns the Ith value (V)
FRA(-I)	writes out the last block
FRA(-I,M)	initializes FRA and selects the data mode

The various data modes described above are determined as follows:

M=0	unsigned integers (0-4095)
M=1	signed integers (-2048 to +2047)
M=2	double precision integers (-2^23 to +2^23)
M=3	- - not available - -
M=4	four word floating point numbers

While FRA may be used to read and write data in a file, it cannot create a new file - the file must already exist as far as the monitor system is concerned. This feature was intentional in order to avoid several difficulties associated with temporary 'open' files. Thus before an area can be used by FRA it must be entered in the directory using the OPEN OUTPUT / OUTPUT ABORT commands to create a suitable directory entry. For example, the command 'O O EMPTY:O A FLEN()' will reserve the largest empty area for use by FRA while O O ARRAY1.FB; O A 100 will set aside 100 blocks for ARRAY1. An area of this size can store 25,600 integers, 12800 double precision integers or 6400 floating point numbers. To initialize the function we would then write: O I ARRAY1.FB;O I,E; X FRA(-1,4). Note that while there is only one FRA function, it can be used with several different files simply by changing the file pointers and re-initializing.

For example:

```
        O I <A>;FOR I=FRA(-1,4),63;S A(I)=FRA(I)
        O I <B>;FOR I=FRA(-1,4),63;X FRA(I,A(I))
```

Since readings from a device such as an analog-digital converter are inherently integers (even though the value may correspond to, say, 5.346 volts), by storing the 'raw' data directly in integer format rather than the 'scaled' values, it is possible to store over 4 times as much data in a given file space. To distinguish these files it is suggested that the extension .FB (FOCAL Binary) be adopted or else perhaps .F0, .F1, .F2, or .F4 as appropriate. (Note the utility of FRA for reading DIAL-LINC tapes via the DL handler).

FRAC( ) fractional part

```
TYPE %11.1 FRAC(PI)

     0.1415926535
```

This is the complement of FITR(). Accuracy is poor for large numbers.

FRAN( ) random number

```
11.22 TYPE FRAN( ),FRAN( ),FRAN( ),FRAN( ),FRAN( ),!

     0.2725   0.2239   0.9841   0.1710   0.0131
```

The pseudo-random numbers produced are part of a very long and well distributed but deterministic series. You will usually observe an entirely different series each time FOCAL is initialized. However, to run a program several times with the same series of random numbers, hit CR twice when loading FOCAL. In this way the random initialization will be disabled immediately. Example: R UWF(CRCR) (two stars (*) will be shown).

FSGN( ) sign

```
12.34 T %1.0 FSGN(2.78)," ",FSGN(-299)," ",FSGN(0)," ",FSGN(-1),!!

     1   -1   0   -1
```

FSIN( ) sine of an angle given in radians

```
13.52 TYPE FSIN(1),FSIN(0),FSIN(37-2.22),!!

     8.414709847E-01  0.000000000E+00  -2.205499649E-01
```

Note that the series approximation used by this function has been optimized for U/W-FOCAL. The errors in the first quadrant are typically less than 5E-11.

FSQT( ) square root of a positive number

```
14.40 TYPE %10.09,FSQT(4),FSQT(391),FSQT(.0038953),!!

     2.000000000  19.77371993   0.062412339
```

FSS( ) or FSSW( ) return the status of the sense switches

FSS(N) tests sense switch 'N' returning -1 if off, +1 if on. Some programmers prefer the more descriptive name FSSW.

FSR( ), FRS( ), FLS( ) read the switch register.

FSR reads the console switch register. Since on the PDP12 there are 2 sets of switches, FRS (or FRSW) reads the right ones and FLS (or FLSW) reads the left ones. The functions FSR and FRS return a signed number while FLS returns an unsigned value (just for variety).

```
12.10 TYPE %5.04,FRS(),FLS()!;C BOTH SWITCHES SET TO 7777

     -1.0000    4095.0
```

FTAN( ) tangent   'FTAN( )' is not available

The tangent is computed as the quotient of (FSIN/FCOS). Note the FSF inplementation of this function - see the next section.

FTIM( ) elapsed time function

FTIM has been implemented using a real-time clock to count tenths of seconds. A negative argument resets the counter, zero (or no argument) reads the counter, and a positive, non-zero value presets it. The maximum time is 838,860.7 secs. (9 days, 17 hours).

FTRM( ) reads and compares the last terminator from an ASK command with its argument. ASK preserves the last terminator so the program can check for the end-of-input by looking for a special terminator. This method is decidedly superior to checking each item for a special value since no assumptions eed to be made about the range of data values. Any character except 0-9 and 'A-Z' terminates numeric input. While the space bar and the carriage return are commonly used, a comma, semicolon, question mark or any other special character will do just as well. The following example illustrates a typical input loop:

```
   1.1 T "END INPUT WITH A CR - OTHERWISE USE A SPACE";Z N
   1.2 S N=N+1;A ! X(N) Y(N)$I (FTRM(141))1.2,,1.2;T "N="N
```

The FTRM function compares the last terminator with the value 141 specified. Note that input from a null file will be terminated by a CTRL/Z (code 154). This allows FOCAL to read data from a file without knowing in advance how many data points to expect. N.B. files to be read in this fashion should not use CR/LF to separate items, since if they do, the last value will not be terminated by the EOF character (CTRL/Z) but rather by a CR.

FXL( ) or FXSL( )   tests an external level (PDP12)

FXL(N) tests the status of an external sense line returning -1 if high, +1 if ground. Some programmers prefer the longer function name FXSL.

***** F O C A L   S T A T E M E N T   F U N C T I O N S *****

FOCAL statement functions are one of the more exciting features of U/W-FOCAL. They allow any sequence of FOCAL statements (i.e. a subroutine) to be called as though it were a function. Thus it is possible to write a single command which computes the tangent of an angle and to then use this statement as F(TAN,A) to find the tangent of any angle! Similarly one can produce a function to find the maximum of a set of numbers and then check this maximum with a command such as IF (F(MAX)-1E4). FOCAL statement functions may also be used for computing line numbers (or formats) which adds a particularly novel capability to the FOCAL language.

The form of a statement function is: F(ref, args...) where "ref" is the line or group number of the subroutine and "args" are the explicit arguments required for the function. Although there are no restrictions on the number of arguments, usually only one - often none at all - is required, since all of the variables defined by the program are available to the function. The first three statement function arguments are saved in the last three protected variables, in the order # first, then $, and finally %. If more than three arguments are required, then the additional ones would be saved in the first variables defined by the program. The ZERO command could be used to initially set these variables in the desired order:

ZERO, A B C D E places A-E immediately after the last protected variable, %.

The value returned by the function is just the value of the last expression to be evaluated. Normally this value is computed by a SET or an XECUTE command, although other commands could also be used. It is also occasionally possible to use a line number calculation to set the result if the statement function ends with a branch command. Note that the commands RETURN, COMMENT, and ZERO do not alter the contents of the floating point accumulator and so may be used after the last evaluation without affecting the result. For example, X PI; Z PI will return the number 3.14159.. even though the variable PI has become zero! ('ZERO'ing a subscripted variable will leave the value of the subscript)

Clearly there are an unlimited number of such functions (in contrast to the fixed number of internal functions) and they may be used recursively in any combination with the regular functions and with each other. In particular a statement function may be an argument of itself. (See examples 6-8 in the next section.)

While a statement function is really not much more than a fancy DO call, it has been a missing element in the structure of FOCAL and its presence now permits a much more logical appproach to programming. Consider the tangent example: If line 13.7 contains the statement SET Z=FSIN(A)/FCOS(A) then DO 13.7 will only find the tangent of the angle A. To use this line to compute the tangent of 3 we would need to

SET A=8;DO 13.7; and then finally recover the result in Z. On the other hand, if we modify line 13.7 to be SET Z=FSIN(#)/FCOS(#) then we can compute the tangent of any angle with the logical call F(13.7,ANGLE). For extra elegance we can set TAN=13.7 and then use the call F(TAN,...) as described before. Note that we get the result in two ways: as the value of Z in line 13.7 and as the result of the statement function. If we did not care to set Z then we would just XECUTE FSIN(#)/FCOS(#).

FOCAL Statement Functions can also be used to implement DO calls in place of GOTO branches. This substitution is occasionally desirable when performing conditional transfers with the IF or JUMP commands, thereby allowing the program to call a subroutine in some cases while branching in others. Since branches to line 0 are ignored by these two commands (and also by the NEXT and BREAK commands), a convenient way to call a subroutine is to use 'F(G)*0' as a computed line number, where 'G' is the (line) or group number of the subroutine. Multiplying by zero simply ensures that the program will continue with the next command regardless of the value returned by the function call. To illustrate:

```
IF (N-2) 1.2,F(2)*0;C 'DO 2' IF N=2, OTHERWISE CONTINUE
```

An obvious extension of this idea is to add a non-zero line number so that after returning from the 'DO' call the program can be instructed to continue elsewhere, i.e. 'F(3)*0+12.1' would create a 'DO 3;GOTO 12.1' while 'F(7.4)*F(9)*0' would call line 7.4, then group 9 before continuing. Note that FSF's which end with an 'X' will return the value '0', thus eliminating the need to multiply by zero. Also note that it is often much clearer, and perhaps even more efficient to simply branch to a line with the appropriate 'DO' call rather that attempt to combine the operations as shown above. This method does, however, allow the program to return to the 'middle' of a line. Finally, note that no argument list may be included in such FSF calls because expressions containing commas may not be used for computing line numbers in IF, ON, or JUMP commands.

EXAMPLES OF FOCAL STATEMENT FUNCTIONS

0. F(PWR,X,Y) - raises X to the Y power (Y non-integer)

```
            9.1 X FEXP(FLOG(#)*FABS($))^FSGN($);R
```

This function must be used whenever the value of X^Y is required and Y is not an integer. If X is negative the sign of the result may not be correct, i.e. F(9,-27,1/3) will return '+3', not '-3' as might be expected.

1. F(!,N) - finds the factorial of (the integer part of) N

```
            1.1 SET $=1;FOR I=1,#;SET $=$*I
```

To use this function call, ! must be set to 1.1. F(!,5)=120, etc.

2. F(SUM) - computes the sum of the numbers X(I)

```
            2.1 ZERO $;FOR I=1,N;SET $=$+X(I)
```

using this function we can write SET AVE=F(2.1)/N

3. F(PN,X) - evaluates the polynomial
Y=C(0)+C(1)*X+C(2)*X^2+..+C(N)*X^N

```
            3.1 ZERO $;FOR I=N,-1,0;SET $=$*#+C(I)
```

Of course the main program must set the values of the coefficients C(0)-C(N) before calling this function. Note the decrementing FOR loop; also note that we could evaluate a polynomial containing half-integer powers by calling F(PN,FSQT(X)).

4. F(%,X,) - determines the smallest integer format

```
            4.1 SET $=$+1;IF (10^$-FABS(#))%,%;XECUTE $
```

This FSF finds the smallest integer format which will output all the significant figures to the left of the decimal point. The variable % must have the value 4.1 for the function to work as shown. Note that there are two arguments, the second one is always zero and is used to initialize the loop. TYPE %F(%,X,),X will store X in the minimum file space.

The use of the secret variable $ in examples 1-3 has no significance other than offering somewhat faster lookup and being an appopriately named variable for accumulating Sums. However, in examples 0,4 this variable is actually set by the second argument of the function.

5. F(MAX) - returns the (algebraic) maximum value of X(I)

```
          5.1 SET I=1;DO 5.3,5.2;X $;R
          5.2 FOR I=2,N;IF ($-X(I))5.3
          5.3 SET %=I;SET $=X(I)
```

An entirely similar function can be written to find the minimum. By replacing 'X(I)' with 'FABS(X(I)))' one can search for the largest magnitude. Note that the command TYPE %F(%,F(5),) will set the format to the minimum size required for the largest element in an array of positive numbers.

6. F(TAN,A) - returns the tangent of A

```
          27.01 C TAN = 27.1
          27.1 X FSIN(#)/FCOS(#)
```

or if FSIN and FCOS have been removed:

```
          27.10 I (#*#-.01)27.2;S #=F(TAN,#/2);S #=2*#/(1-#*#+1E-99)
          27.20 S #=#*#^3/3+#^5/7.5+#^7/315
```

7. F(ASIN,A) - returns the arc sine of A, F(ACOS,A) the arc cosine

```
          30.01 C  ASIN=30.1  ACOS=30.3
          30.1 X FATN(#/FSQT(1-#^2)
          30.3 X FATN(FSQT(1-#^2)/#)
```

or

```
          30.10 I (#*#-.01)30.2;S #=2*F(ASIN,#/(FSQT(1+#)+FSQT(1-#)))
          30.20 S #=#+#^3/6+.075*#^5+#^7/22.4
          30.30 X PI/2-F(ASIN)
```

8. F(HSIN,A) and F(HCOS,A) for the hyperbolic functions

```
          31.01 C  HSIN=31.1  HCOS=31.3
          31.1 X (FEXP(#)-FEXP(-#))/2
          31.3 X FSQT(1+F(HSIN)*#)
```

or

```
          31.10 I (#*#-.01)31.2;S #=F(HSIN,#/3);S #=3*#+4*#^3
          31.20 S #=#+#^3/6+#^5/120
          31.30 X FSQT(1+F(HSIN)*#)
```

Examples 6, 7 and 8 are recursive formulations as reported in DECUS: FOCAL8-89 BY A.K. Head of Melbourne Australia. Of course the more conventional computations can also be used if the regular trancendental functions are available.

9. HYBERBOLIC TAN, ARC HYPERBOLIC SINE, ARC HYPERBOLIC COSINE and ARC HYPERBOLIC TAN:

```
31.4 X F(HSIN,#)/F(HCOS,#);C HTAN=31.4
31.5 X FLOG(#+FSQT(#^2+1));C USE 31.5 FOR ARC HYPER SINE
31.6 X FLOG(#+FSQT(#^2-1));C USE F(31.6,A) FOR ARC HYP.COS
31.7 X (FLOG(1+#)-FLOG(1-#))/2;C F(31.7,A) FOR ARC H.TAN
```

10. Radix conversion: Decimal to octal, etc.

Group 8 returns a Pseudo-octal number while group 10 re-converts such numbers back to decimal. A Pseudo-octal number is one which -prints- in an octal representation. The method may easily be extended to other number bases by changing the numbers 8, 10.

```
08.10 Z $;F %=4,-1,0;S "=FITR(#/8^%),#=#-8^%*",$=10*$+"

10.10 Z $;F %=4,-1,0;S "=FITR(#/10^%),#=#-10^%*",$=8*$+"
```

The loop limits have been chosen to convert up to a 5-digit number which permits printing all addresses in a 32k machine. It is quite an easy matter to extend this range by changing the value "4".

```
example: TYPE %5 F(8,512)  F(10,F(8,512))

      1000    512
```

- - - - - - -

Other examples of FSF'S are found in the next section. The FSF in line 20.1 of Example 2 is used to call the triangularization routine. It was noticed that the last element computed by that routine (after hundreds of steps!) was the first value required in the back substitution. In this case the result returned by the function is almost trivial in comparison with the major purpose of the FSF call.

FSF's have also been used extensively in the routine to transform a general matrix to Upper HESSENBERG form; this approach considerably simplified the coding of this routine.

- - - - - - -

***** EXAMPLES OF MATRIX PROGRAMS *****

1. Matrix inversion by GAUSSIAN elimination

This routine computes the inverse of a matrix on top of the original. By appending one or more columns to the matrix, the same routine can be used to solve a system of simultaneous linear equations. The solution is obtained in the appended column(s). No interchanges are used, hence this method is not unconditionally stable and small diagonal elements at any stage will lead to poor results. The use of double subscripting permits this useful routine to be coded in only 3 lines! 'DO 10' is the call. When using this routine to solve a system of equations, change the FOR loop limits in lines 10.2 and 10.3 to !+N where N is the number of additional columns.

```
10.10 FOR I=1,!;DO 10.2;FOR J=1,!;IF (I-J)10.3,,10.3;N;N;R
10.20 SET Z=A(I,I),A(I,I)=1;FOR J=1,!;SET A(I,J)=A(I,J)/Z
10.30 SET Z=A(J,I);ZERO A(J,I);FOR K=1,!;SET A(J,K)=A(J,K)-A(I,K)*Z
```

Group 3 of Example 5 adds the row-interchange technique to this basic algorithm in order to improve the numerical stability.

2. Householder's method for simultaneous linear equations.

This routine uses elementary Hermitians to triangularize the matrix of coefficients followed by back-substitution to compute the unknowns. The method is completely stable but does not compute the inverse at the same time. 'DO 20' is the call. The FSF in line 20.1 calls the triangularization routine and the rest of the line does the back substitution. The "inhomogeneous" terms of the equation must be appended as an extra column to the matrix of coefficients. The vector X returns the solutions.

```
20.10 SET X(!)=F(20.2)/A(!,!);FOR R=!-1,-1,1;SET X(R)=F(20.7)/X(R);N;R
20.20 FOR R=1,!-R;D 20.3,20.4,20.5,20.6;C A(I,J) ARE THE COEFFICIENTS
20.30 ZERO $;FOR I=R,!;SET $=$+A(I,R)^2;C THE "KNOWN TERMS" ARE APPENDED
20.40 SET X(R)=FSQT($)*FSGN(A(R,R)),A(R,R)=A(R,R+X(R),$=A(R,R)*X(R)
20.50 FOR J=R+1,!+1;ZERO X(J);FOR I=R,!;SET X(J)=X(J)+A(I,R)*A(I,J)
20.60 FOR J=R+1,!+1;FOR I=R,!;SET A(I,J)=A(I,J)-A(I,R)*X(J)/$
20.70 SET $=-A(R,!+1);FOR I=R+1,!;SET $=$+A(R,I)*X(I)
```

3. Reduction of a general matrix to upper Hessenberg form

This routine uses the stabilized transformations described by J.H. WILKENSON (The Algebraic Eigenvalue Problem - Clarendon Press, Oxford 1965). Reducing a full matrix to this form greatly decreases the amount of computation required to complete the diagonalization. DO 3 is the call.

```
03.10 FOR R=1,!-2;DO 3.2;ON (F(3.4,R)),4;IF ("-!)3.6
03.20 SET "=R+1;IF (R-2)3.3;FOR I=1,R;SET #=I-1;DO 3.7
03.30 XECUTE;REDUCE A(I,J) TO UPPER HESSENBERG FORM
03.40 ZERO Z;FOR I=!,-1,";IF (FABS(F(3.7))-%),,3.5
03.50 SET %=FABS(A(I,R)),J=I,R(R)=J
03.60 FOR I=R+",!;SET A(I,R)=A(I,R)/A(",R)
03.70 IF (R-1)4.3;SET A(I,R)=A(I,R)+F(3.8)-F(3.9)
03.80 IF (!-")3.3;ZERO $;FOR K=",!;SET $=$+A(I,K)*A(K,R-1)
03.90 IF (#-2)3.3;ZERO $;FOR K=2,#;SET $=$+A(I,K-1)*A(K,R)

04.10 FOR I=1,!;SET #=A(",I),A(",I)=A(J,I),A(J,I)=#
04.20 FOR I=1,!;SET #=A(I,"),A(I,")=A(I,J),A(I,J)=#
04.30 XECUTE A(I,R);ROW AND COLUMN INTERCHANGES
```

4. Finding the eigenvalues of a general matrix

```
01.10 C THIS PROGRAM FINDS THE EIGENVALUES OF THE MATRIX A(I,J)
01.20 C USING THE QR ALGORITHM. THE MATRIX IS NOT REQUIRED TO
01.30 C BE SYMMETRIC - COMPLEX EIGENVALUES WILL BE COMPUTED IF
01.40 C NECESSARY. THE ORIGINAL MATRIX IS SAVED IN C(I,J) FOR
01.50 C USE LATER BY EIGVEC.
01.60
01.70 F I=1,!;F J=1,!;S C(I,J)=A(I,J)

02.10 S K=A(!,!),M=1,N=!,%=1E-10,CO=2.9
02.20 D 2.3;F L=M,N-1;D 3;N;F L=M,N-1;D 4
02.30 S K=-K;F L=M,N;S A(L,L)=A(L,L)+K
02.40 S L=L-1;I (L-2)2.5;I (%-FABS(A(L,L-1)))2.4
02.50 S K=A(N,N),M=L;I (L+1-M)2.2,2.7;S X(N)=K;Z Y(N);G 2.9
02.60 S J=FSQT(-J),X(L)=I,X(N)=I,Y(L)=J,Y(N)=-J;G ?COMPLEX ?
02.70 S I=(A(L,L)+K)/2,J=I*I+A(L,N)*A(N,L)-A(L,L)*K;I (J)2.6
02.80 S J=FSQT(J)*FSGN(I+%),X(L)=I+J,X(N)=I-J;Z Y(L) Y(N)
02.90 S L=L-1,N=L;I (1-L)2.4,2.5;RETURN EIGENVALUES IN X(I),Y(I)

03.10 Z $;F I=L,N;S $=$+A(I,L)^2
03.20 S $=FSQT($)*FSGN(A(L,L)),B(L,L)=A(L,L)+$
03.30 S Q(L)=B(L,L)*$;F I=L+1,N;S B(I,L)=A(I,L)
03.40 F J=L,N;Z P(J);F I=L,N;S P(J)=P(J)+B(I,L)*A(I,J)
03.50 F J=L,N;F I=L,N;S A(I,J)=A(I,J)-B(I,L)*P(J)/Q(L)

04.10 F I=M,N;Z P(I);F J=L,N;S P(I)=P(I)+A(I,J)*B(J,L)
04.20 F I=M,N;F J=L,N;S A(I,J)=A(I,J)-P(I)*B(J,L)/Q(L)
```

5. Finding the eigenvectors of a general matrix.

```
01.10 C THIS ROUTINE COMPUTES THE EIGENVECTORS OF A(I,J) USING
01.20 C THE METHOD OF INVERSE ITERATION. GOOD ESTIMATES OF THE
01.30 C EIGENVALUES ARE REQUIRED FOR RAPID CONVERGENCE AND IM-
01.40 C PROVED VALUES ARE RETURNED.  THE FINAL XFORM IS: A*C*B
01.50
01.60 F N=1,!;D 2,3,4,5;CALL THIS SEQUENCE FOR EACH EIGENVALUE
01.70 F I=1,!;F J=1,!;S A(I,J)=B(I,J);N;N;D 3;RETURN BOTH SETS

02.10 COPY THE MATRIX AND SUBTRACT THE EIGENVALUE
02.20 F I=1,!;F J=1,!;S A(I,J)=C(I,J)
02.30 F I=1,!;S P(I)=I,A(I,I)=A(I,I)-X(N)

03.10 F I=1,!;D 3.5,3.6;S P(I)=#;F J=1,!;I (I-J)3.4,,3.4
03.20 F J=!-1,-1,1;S #=P(J);F I=1,!;S "=A(I,#),A(I,#)=A(I,J),A(I,J)="
03.30 C GAUSSIAN INVERSION OF A(I,J) USING THE ROW INTERCHANGE METHOD
03.40 S $=A(J,I);Z A(J,I);F K=I,!;S A(J,<)=A(J,K)-A(I,K)*$
03.50 Z $;F J=I,!;S K=FABS(A(J,I));I (K-$)3.3;S #=J,$=K;I ($),3.7
03.60 S $=A(#,I),A(#,I)=1;F K=I,!;S "=A(#,K)/$,A(#,K)=A(I,K),A(I,K)="
03.70 S A(#,I)=%;C THE PIVOT IS ZERO! SET TO 'TOLERANCE' AND CONTINUE

04.10 C THIS ROUTINE PERFORMS THE INVERSE ITERATION
04.20 F I=1,!;Z Q(I);F J=1,!;S Q(I)=Q(I)+A(I,J)*P(J)
04.30 Z $;F I=1,!;I (FABS(Q(I))-FABS($))4.1;S $=Q(I)
04.40 Z #;F I=1,!;S #=#+FABS(P(I)-Q(I)/$),P(I)=Q(I)/$
04.50 I (!*%-#)4.2;RETURN AS SOON AS THINGS CONVERGE

05.10 CORRECT THE EIGENVALUE & NORMALIZE THE VECTOR
05.20 S X(N)=X(N)+1/$;Z $;F I=1,!;S $=$+P(I)^2
05.30 S $=FSQT($);F I=1,!;S B(I,N)=P(I)/$;C SAVE IT
```

```
*****  * * * * * *  * * *  * * * * * * * * *  *****
*****  A D D I N G  N E W  F U N C T I O N S  *****
*****  * * * * * *  * * *  * * * * * * * * *  *****
```

ADDING USER FUNCTIONS AND COMMANDS TO U/W-FOCAL

This section outlines the procedure for adding new functions and commands to U/W-FOCAL and reviews the various internal routines which are frequently called by such functions. Those acquainted with other versions of this language will find most of the principles familiar, but there are some new features in this version which facilitate argument checking and data conversion.

Basically there are two reasons for implementing new functions or commands: one is to gain access to specific IOT instructions for controlling on-line equipment and the other is to obtain a speed advantage by eliminating the overhead of the interpreter. Writing time critical funtions in assembly language and linking these together with a simple FOCAL program creates a very effective combination.

User routines may be added to this version either as new commands or as new functions. The choice in a given situation is generally governed by the nature of the information transfer: those operations which perform only output are usually most conveniently programmed as commands while those which return information to the program need to be implemented as functions. Examples of the first class are PLOT (which moves pen and paper), and KONTROL (which operates relays). Obvious examples in the second catagory are things like FADC (reads an A/D converter) and FXSL (tests the status of an external sense line). There are also functions which perform control (output) operations, but which use the value returned by the function as a status report.

CALLING THE FUNCTION

The first question which arises is how internal routines are called by a users program. When a FOCAL program is running (including the execution of direct commands), the characters stored in the text buffer are scanned one at a time in order to determine what action to take. The letter "F", for instance, might cause a branch to the FOR command processor, or, in another context, it might represent the beginning of a function call. In order to make these decisions, FOCAL employs character-driven branches in which the value of the letter code is used to determine which address in a table to branch to. Thus in order to implement new commands or functions, all a user needs to do is to patch the appropriate table with an address indicating the entry point of his routine and the interpreter will do the rest. At the conclusion of this routine an appropriate JMP instruction ('CONTINUE' or 'RETURN') will re-enter the interpreter and the text scan will be resumed.

COMMAND BRANCHES are performed through a table containing 26 addresses starting at the location COMGO. (Only symbolic names will be used in this discussion; consulting a current listing of the symbol table will provide the octal values). The letter "B", for example, causes a branch to the second address, the letter "C" to the third address, etc. Since FOCAL uses only the first letter of a command word, this table cannot be expanded; furthermore most of the letters are already in use. However, a given letter can be 'multiplexed' by using it for a series of double-word commands as has been done for the letters "L" and "O". With this idea in mind the letter "U" has been reserved for new USER commands, i.e. "U A", "U B", etc. A method for implementing such commands is shown on page 78.

FUNCTION BRANCHES are a bit more complicated since functions names may consist of several letters (and/or numbers) and it is obviously impossible to store entry points for all conceivable 1,2 & 3 character names. In this case, then, two tables are employed: FNTABL and FNTABF, each containing 35 entries. When FOCAL detects a function call it compares the name given in the program with the list of possible function names in FNTABL, and upon finding a match, jumps to the corresponding address in FNTABF. When adding a new function it is thus necessary to patch both tables at the same relative location. The patch to FNTABF is quite simple: it is just the address of the new function. The entry in FNTABL needs some additional comment.

In order to pack an entire function name into a single core location FOCAL resorts to a method of 'hash-coding' in which all the letters (except for the "F") are combined. The algorithm for doing this is very simple, but with simplicity lurks the possibility that the result may not be unique. Basically the 7-bit character codes are summed after multiplying the previous sub-total by four. This generates a polynomial function using the ASCII codes as coefficients. The resulting value must be positive since negative numbers are used to define the end of the table. This evaluation may be performed by the assembler as follows: (the code for 'FABC' is shown.)

"A-200^4+"B-200^4+"C-200

WHERE DOES IT GO?

The next question which usually arises is where to locate the code for new functions. Since only 12-bit addresses are used in the branch tables, all entry points

MUST be in the same field as the interpreter; in U/W-FOCAL this is Field 1. However, if a large amount of code is required to implement the operation (and it seldom is - most user functions can be coded in 10-40 instructions) it may be prudent to locate such code in another field. All the "L" and "O" commands, for example, are coded in Field 0 except for about 50 instructions which provide the necessary linkage between the fields. On the other hand, it is generally much more convenient to be able to write the function 'all-in-one place' and in addition, functions which reside in Field 1 have access to all the text-scanning and I/O routines as well as use of the floating-point package. For this reason the area from 3200-4177 in Field 1 has been reserved for user additions; the only catch is that this area is also used by the 8K version for storing variables. Thus those with only 8K must give up some variables in order to add new functions and commands (see p. 88). Note also that there are 10 unused page-zero locations to facilitate cross-page references or patches to other routines which might be required by a sophisticated programmer. These and any other 'holes' can be easily determined by consulting the appropriate 'bitmap'.

COMMUNICATING WITH THE FUNCTION:

It is, of course, essential that a FOCAL program be able to pass information to the function or command which it invokes. This link is usually in the form of an arithmetic expression whose value is available to the user's routine (1). Such parameters appear as a normalized floating-point number in a set of page-zero registers known as FLAC. Most user routines, however, prefer a 12-bit integer result which can be manipulated in the hardware accumulator (the AC). One of the first steps in most user functions is thus to convert from one form to the other. This is done by a call to FIXIT which converts FLAC into a 24-bit integer, clears the link and loads the least significant 12-bits (LORD) into the AC. In this way the result of what may have been a very complicated expression in the program is ultimately reduced to a few managable bits in the accumulator!

We have been assuming above that there -was- a parameter and that it had already been evaluated and saved in FLAC before arriving at the user's routine. This is true of all function calls since FOCAL assumes that there will always be at least one argument and goes ahead and evaluates it. If, in fact, there is not an explicit argument (for example in a function call like FIN()), the evaluation process returns the

---

(1) string arguments are also possible - this subject will not be discussed here.

value zero. The situation for commands is slightly different since not all commands require a numerical parameter so FOCAL leaves it up to the user to ask for an evaluation if one is required. This step is also necessary for functions which have more than one argument, so the following comments are pertinent to both types of routines.

PERFORMING AN ARITHMETIC EVALUATION:

Evaluating an arithmetic expression can obviously be a very complicated process involving nested parentheses, subscripted variables, mixed operators, etc. The evaluator routine, for example, must be able to stop in the middle of one evaluation and start on a completely new one whose result is required by the first expression. Obtaining the value of a subscripted variable is a simple example of this sort of complication: the subscript expression must be evaluated before the value of the variable can be obtained. FOCAL is, of course, prepared to handle all these situations, but it should not be surprising that a special call is required to initiate an arithmetic evaluation. This call is a PUSHJ operation which is sort of a combination JMP and JMS instruction.

PUSHJ performs a JMP to the entry point of the routine, but it saves the return address (like a JMS) on the 'stack'. PUSHJ instructions are coded as two-word instructions with the second word being the address of the routine. In order to evaluate an arithmetic expression, for example, the call:

```
        PUSHJ
           EVAL
```

is required. (The second word of a two-word instruction is usually indented a few spaces in the listing). Upon reaching the end of the evaluation the program will resume at the location following the address EVAL.

PROCESSING MULTIPLE ARGUMENTS:

Many user routines find a need for multiple arguments. For example, a routine to control a series of stepping motors might need the drive number and the number of steps. Another common situation is the use of a variable number of parameters. This technique is used, for example, by functions like FCOM which 'read' if called with one argument and 'write' if called with two. In other situations it may just be convenient to default to certain values if explicit values are not provided by the function or command call. In all these cases the user routine needs to be able to anticipate whether there are additional parameters, and if so, ask FOCAL to evaluate them as described above.

This check is simplified by requiring that multiple expressions be separated by commas (expressions not beginning with a minus sign may also be separated by spaces; however in the general case commas are the only permissable separators). Thus after each evaluation the programmer needs only check the current character from the text buffer to see if it is a comma, and if it is, skip it and evaluate the next parameter. This requirement occurs so frequently that there is a special routine for performing this service: TSTCMA. TSTCMA also provides an example of a technique used by several other internal subroutines: multiple return points. If TSTCMA does not find a comma in the text buffer, it returns to the next core location (i.e. 'call+1') whereas if it does find a comma, it skips that location and returns to 'call+2' or the 'second return'. In addition, TSTCMA also skips over the comma if there was one so that EVAL won't get stuck(1). Combining the routines mentioned so far leads to the basic input structure of all user functions and commands. An example which processes multiple arguments and jumps to the main routine when no more arguments are available might be coded as follows:

```
 CENTRY, PUSHJ              /THE COMMAND ENTRY
              EVAL
 FENTRY, FIXIT              /THE FUNCTION ENTRY
         DCA STORAGE        /SAVE THE INTEGER ARGUMENT
         ...                /UPDATE STORAGE POINTERS, ETC.
         TSTCMA             /ARE THERE ANY MORE PARAMETERS?
         JMP MAIN           /NO, GO DO SOMETHING
         JMP CENTRY         /YES, GET THE NEXT ONE.
```

An example which forces all calls to have two arguments might be written:

```
    FTWO, FIXIT          /ASSUME THAT THIS IS A FUNCTION
          DCA ARG1
          TSTCMA         /THERE SHOULD BE TWO ARGUMENTS
          ERROR2         /BUT WE ONLY HAVE ONE!
          PUSHJ
              EVAL
          FIXIT          /ETC.
```

The ERROR2 instruction above calls FOCAL's error routine which stops the program and prints out an error code equal to the page number and relative location on the page where the error occured. For example, if ERROR2 is located at location 3246(oct) the error code would be 13.38.

---

(1) It is also possible to ask EVAL to skip the comma by performing a call to EVAL-1; however this method does not work as nicely for commands as it does for functions since commands need the initial call to EVAL anyway.

Functions which have more than one argument require some care when coding them if the second argument is to be permitted to contain references to the same function (i.e. recursive calls). This point has been omitted in the preceeding examples. Basically code which might be reused by a second function call cannot store arguments in fixed locations since these values would then be overwritten before they could be used. The solution is just to place all such arguments on the 'stack', recalling them later as necessary. There are 4 instructions which are available for such operations: PUSHA, POPA, PUSHF and POPF. The first two save and restore a single 12-bit number in the AC, the second two save and restore 4 consecutive locations such as might be used to contain a floating-point number. As an example of the use of these routines, a function to read or write into any core location in any field might be coded as follows:

```
FCOR,   FIXIT           /1ST ARGUMENT IS THE FIELD CODE
        PUSHA           /SAVE IT ON THE STACK
        PUSHJ           /ASSUME A MINIMUM OF 2 ARGUMENTS
             EVAL-1
        FIXIT           /THIS IS THE CORE LOCATION
        PUSHA
        TSTCMA          /A 3RD ARGUMENT MEANS 'DEPOSIT'
        JMP EXAM        /OTHERWISE JUST 'EXAMINE'
        PUSHJ           /GET THE VALUE - THIS CALL MAY
             EVAL       /REENTER FCOR WHEN MOVING DATA
        JMS LOCN        /SET THE ADDRESS AND DATA FIELD
        FIXIT           /BRING THE DATA INTO THE AC
        DCA I XRT       /DEPOSIT IT
        RETURN          /AND RETURN THE SAME VALUE

EXAM,   JMS LOCN        /CALLED WITH ONLY 2 ARGUMENTS
        TAD I XRT       /READ THE VALUE
        FLOATR          /AND RETURN WITH IT

LOCN,   0               /SUBROUTINE TO SET THE DATA FIELD
        CMA             /COULD BE AVOIDED, SHOWN FOR GENERALITY
        POPA            /BACK UP THE ADDRESS BY ONE
        DCA XRT         /AND SAVE IN AN INDEX REGISTER
        POPA
        AND P7          /MASK FIELD BITS
        CLL RAL
        RTL
        TAD FCDF        /ADD IN A 'CDF'
        DCA .+1
        0               /THIS IS THE 'CDF' INSTRUCTION
        JMP I LOCN
FCDF,   CDF
```

Note that if the second argument had been saved directly in an index register and the routine had been reentered by a call such as X FCOR(3,1000,FCOR(2,512)) the initial value would have been destroyed. Many user routines do not require this feature and may thus reduce their execution time and simplify their coding by storing their arguments directly into the locations where they are used. A useful bit of knowledge in this respect is that the auto-index register AXIN is always available while the program is running. Thus non-conflicting functions or commands may initialize this register and use it to store a variable list of parameters. To avoid problems however, it is generally recommended that an on-page register be used for such addressing with an ISZ instruction to advance it.

CONVERTING BCD DATA:

Many user functions, especially those which read laboratory instruments, need to convert from BCD to binary. J/W-FOCAL has a routine specifically designed for doing this called MULT10. MULT10 first multiplies the value in FLAC (treated as an integer) by ten, then adds in the value of the calling AC. This is just the algorithm needed for processing BCD digits starting with the most significant. It only remains to set EXP to 43(oct.)(=35 bits) before returning from the function (the constant P43 is on page zero). Of course FLAC must be cleared initially. This is accomplished with another handy routine called FLOAT which turns the AC into an unnormalized floating-point number; if the AC is zero, FLOAT obviously just clears FLAC. A call to NORMALIZE is advised before using the floating-point package to operate on numbers prepared by FLOAT or MULT10.

ACCESSING USER VARIABLES:

Some functions and commands require explicit access to specific variables. Examples of this requirement occur in FJOY which uses XJ and YJ for the coordinates of the 'joystick' cursor and in the PLOT command which makes use of the special variables $S, $D, $R and $F. The following discussion applies only to unsubscripted variables and an examination of the routine GETARG is suggested before adapting this procedure to subscripted ones. Since all variables may be subscripted, the variable-lookup routine employs reentrant code and hence must be entered with a PUSHJ call. The name being searched for is coded in packed ASCII and stored in THISOP; LORD must be set to zero. The call

```
                    PUSHJ
                          GS1
```

then searches the entire symbol table and either finds or creates a variable with the name in THISOP. It is sometimes useful to know which course of action occured. This information is returned in the link which is set if the variable was found and cleared otherwise. In any case the variable pointer PT1 is set to the location of the first data word (i.e. the exponent).

Since the variables may now be in almost any field, user routines which are expected to work in several different configurations must 'follow the rules' for accessing them. The only 'legal' way to do this is to use the floating-point package which has been modified especially for this purpose. The following routine, for example, will bring the value of the variable 'AB' into FLAC:

```
                TAD NAME        /SET UP THISOP AND LORD
                DCA THISOP
                DCA LORD
                PUSHJ           /SEARCH THE SYMBOL TABLE
                    GS1
                FENT
                FGETIPT1        /BRING IT INTO FLAC
                FEXT
                ....

        NAME,   0102            /PACKED ASCII FOR "AB"
```

The special operation FGETIPT1 is interpreted by the F.P.P. as an out-of-field reference which is otherwise equivalent to FGET I PT1. This will be discussed more fully in a later section.

CHECKING FOR LIMITS:

Arguments which are used to build instructions should always be masked or checked for limits to avoid creating erroneous operations. For example, if the instruction 6540 sets D/A channel 0, then the following code might be used to implement a multichannel FDAC function:

```
  FDAC,     FIXIT         /PLACE THE CHANNEL NO. IN THE AC
            AND P7        /MASK OFF THE LOWER 3 BITS
            TAD (6540     /ADD INSTR. FOR CHANNEL 0
            DCA .+ 5      /AND PLACE IN-LINE FOR EXECUTION
            etc.
```

Note that P7, P17, P77 and other useful constants are on page-zero. Another technique of general utility for checking that an argument falls in the range 0-N is to add -(N+1) and check for overflow into the link. This is conveniently done following a call to FIXIT since this routine always returns with the link cleared:

```
        FIXIT           /GET THE 1ST PARAMETER INTO THE AC
        TAD (-6         /CHECK FOR A VALID RANGE
        SZL
        ERROR2          /NOT IN THE RANGE 0-5
        TAD (INST+6     /COMPENSATE FOR THE TEST CONSTANT
        ...
```

This is a 'friendly' approach to limit checking in which the programmer is permitted to make improbable errors which will not lead to disaster. For example, the limit check just shown does not consider the possibility that the HORD bits of the argument are non-zero. Thus an argument of 12,292 will be gracefully accepted by this function (supposedly limited to 0-5) producing the same result as an argument of 4; the additional code to check for such an unlikely argument seems completely unnecessary.

RETURNING TO THE INTERPRETER:

Finally we come to the end of the function or command and need to return to the interpreter. Returns from a command are coded as CONTINUEs. This return clears the AC and fetches the next command. Function returns may be coded in 1 of 3 ways, some of which have already been illustrated: RETURN, FLOATR and FLOATR. The first clears the AC and returns whatever value is stored in FLAC; generally this is just the value of the last argument although other results may be set up, for instance through the use of calls to MULTIO or the floating-point package. The contents of FLAC will be normalized by RETURN so this need not to be done by the function. FLOATR (with "oh") on the other hand uses the value in the AC as the value of the function. This is convenient for returning error values as well as for converting 12-bit results. The function FSR, for example, is coded as LAS;FLOATR. The third possibility, FLOATR (with "zero"), is almost the same as the second except that the AC is treated as an unsigned number in the range 0-4095 rather than as a signed number in the range -2048 to +2047. On the PDP12, for example, the value of the left switch register is returned as an unsigned number.

TRICKS OF THE TRADE:

FIXIT actually turns FLAC into a 24-bit integer so that both -1 and 4095 load the AC with 7777. All calls to EVAL leave the result in FLARG as well as in FLAC, thus the value in FLAC may be tested or destroyed without losing the original value. FLARGP is the address of FLARG. The temporary registers T2 and T3 may be used by any function which calls EVAL or the floating-point package. T1, SIGN and the index registers are certain to be altered, however. Many of the common subroutines (such as FIXIT, SHIFTL) do not care about the data field setting. This is sometimes convenient but subject to change as additional EAE routines are developed.

Functions which use LINC mode instructions (PDP-12) must disable the interrupt system even though only a few microseconds may be spent in this mode. Rather than using an IOF instruction followed later by ION it is usually more convenient to add a CIF 10 instruction at the start of the LINC mode section. This will temporarily disable the interrupt system until the next JMP or JMS which is typically just the RETURN or CONTINUE command.

USING THE FLOATING POINT PACKAGE

U/W-FOCAL has a 4-word floating point interpreter which simulates floating point memory reference instructions for data in Field 1 plus limited references to data in the field containing the variables. There are 8 possible operations which are arranged in the order of FOCAL's arithmetic priorities and hence do not correspond exactly to the definitions used in some other packages. These operations are:

```
FGET = 0000  loads FLAC, leaves OPER unchanged
FADD = 1000  loads OPER, adds it to FLAC
FSUB = 2000  loads OPER, subtracts it from FLAC
FDIV = 3000  loads OPER, divides OPER into FLAC
FMUL = 4000  loads OPER, multiplies it with FLAC
FPWR = 5000  loads OPER, raises FLAC to integer power of OPER
FPUT = 6000  saves FLAC, leaves FLAC, OPER unchanged
FNOR = 7000  loads OPER, normalizes FLAC
```

It should be noted that FPWR now works for both positive and negative values of OPER. In particular this permits a simple way to form the reciprocal of a number by raising it to the -1 power. Since only the integer part of the operand is used one does not need a constant exactly equal to -1, and the following 'trick' may be used to produce a value equal to approximately -1.74:

```
FM1, 1               /THIS IS THE EXPONENT
     FENT            /THIS IS 'WORD'
     FPWR FM1        /TAKE THE RECIPROCAL
     FEXT            /OR ANYTHING ELSE
```

To enter the floating point package one uses the FENT call, to exit, code FEXT as the operation. In order to provide access to data in the 'variables' field all indirect references through location 0 (i.e. things like FPUT I 0) are mapped to be out-of-field indirect references through PT1 (the 'variable pointer'). It is convenient to define special symbols for such operations by condensing the expression 'FGET I PT1' into 'FGETIPT1' = 'FGET I 0', etc. There is one exception to this interpretation: data located at 7600 will always be fetched from Field 1 since this location is trapped for references to FLARG. No variables are located at this position anyway (by design).

All floating-point operations assumed normalized numbers and all conclude with a call to the normalize routine. FADD and FSUB may be used with unnormalized numbers with a consequent loss of precision, but FMUL, FDIV and FPWR should always have normalized arguments. Floating point calculations frequently need temporary storage areas. There are two which are conveniently reached via indirect pointers on page-zero. FLARG is used by EVAL to copy the result of each operation and BUFFER is used by some of the trancendental functions. Their pointers are known as FLARGP and BUFFPT.

EXPANDING THE COMMAND LIST:

The following code may be used to expand any one of the command letters; it is shown for the USER command:

```
        *COMGO+"U-"A
        USER                /ENTRY IN THE MAIN COMMAND TABLE
        .
        *SOMEWHERE
USER,   SPNOR               /IGNORE SPACES BETWEEN WORDS
        TAD CHAR
        DCA LASTC           /SAVE THE SECOND COMMAND LETTER
        SORTX               /TEST FOR ANOTHER TERMINATOR
        JMP .+3             /FOUND ONE
        GETC                /IGNORE ALL BUT THE FIRST LETTER
        JMP .-3
        TAD LASTC           /PERFORM  A SECOND BRANCH
        SORTJ               /USING THE SECOND COMMAND WORD
            USRLST-1
            USERGO-USRLST
        JMP I (CERR         /UN-AVAILABLE COMMAND

USRLST,"A                   /LIST OF SECOND COMMAND LETTERS
        "W                  /THESE LISTS MAY BE PLACED ANYWHERE
        "G                  /IN FIELD 1
        ETC.                /MUST BE ENDED BY A NEGATIVE NUMBER

USERGO, USERA               /ENTRY-POINT OF THE UA COMMAND
        USERW               /                  U W
        USERG               /                  U G
        ETC.

USERA,  .....               /START OF THE U A COMMAND
        CONTINUE            /RETURN TO THE INTERPRETER

USERW,  DITTO
```

ASSEMBLING AND OVERLAYING A USER FUNCTION:

After working out the general plan of the code for a new user function (or command) the programmer should create a source file using EDIT, TECO, etc. This file should also contain definitions for all the subroutine calls and other FOCAL symbols which it references as well as plenty of commentary so that a listing is self-documenting as to the purpose and form of the function call. Typically this source file is given the name of the function. This file is then assembled and the resulting binary used to patch (overlay) the original U/W-FOCAL binary. A typical source file example: FNOP.PA

```
/  FNOP:  EXAMPLE FUNCTION FOR U/W-FOCAL      -JVZ-

/ *FNOP* DOES NOTHING IN PARTICULAR, BUT IT DOES IT VERY WELL!
/CALLS MAY HAVE EITHER 1 OR 2 ARGUMENTS: FNOP(X) OR FNOP(X,Y).


                FIELD 1

                FIXIT=4560          /THESE DEFINITIONS MAY NOT BE
                RETURN=5555         /CORRECT FOR EARLIER VERSIONS
                TSTCMA=4543
                PUSHJ=4522
                EVAL=1606
                FNTABF=2157
                FNTABL=2356

                *FNTABF+26          /PATCH THE ENTRY POINT
                FNOP

                *FNTABL+26          /PATCH THE NEW NAME
                "N-200^4+"O-200^4+"P-200

                *3200               /USER AREA
        FNOP,   FIXIT               /CONVERT THE FIRST ARGUMENT
                TSTCMA              /CLEARS THE AC FIRST
                RETURN              /THE INTEGER VALUE
                PUSHJ               /EVALUATE THE NEXT PARAMETER
                    EVAL
                FIXIT               /CONVERT THIS ONE TOO
                RETURN              /THIS ALSO CLEARS THE AC
                $
```

To add this function, do the following:

```
                .R PAL8             (or .COM FNOP)
                *FNOP<FNOP          (creates FNOP.BN)
                .R ABSLDR
                *UWF,FNOP=100$      (merge main binary and patch)
                .SAVE SYS:UWF       (save the new version)
```

ADDING INTERNAL DEVICE HANDLERS:

While the use of the OS/8 system handlers is generally quite convenient, occasionally there is a need to add other internal handlers to U/W-FOCAL. One example might be the inclusion of a line-printer handler so that output could be sent to a file and to the LPT:, if desired. Other examples might be a CRT display which needs character-oriented output or the annotation routines for a digital plotter. Even though routines such as the last one could never be written as OS/8 handlers, it is convenient to let the FOCAL program treat all I/O calls in the same way, so that, for instance, the call 'O O PLTR:' could be used to switch output to the annotation routines.

This approach is already in use for trapping calls to the TTY: handler, i.e. 'O I TTY:' does not load the OS/8 handler but merely switches to the internal input routine. Hence it is a relatively simple matter to add checks for other internal handlers, all of which must have their entry points in the upper half of Field 1. Traps for such handlers are inserted into the OPEN routine in Field 0. The most convenient place to do this is at OPEN+5 which contains a GTNAME instruction. This may be moved (by condensing the IAC;RAL sequence preceding it) or replaced by 'JMS I (PATCH' (note that this requires modifying the next instruction as well) and PATCH is coded approximately as follows:

```
PATCH,    0               /THIS MUST BE SOMEWHERE IN FIELD 0
          GTNAME          /REPLACE THE ONE WE ELIMINATED
          CMA CLL RAL     /=-2
          COMPARE         /CALL THE BLOCK COMPARISON
             NEWDEV-1     /ROUTINE TO SEE IF THE NAME
             YORDEV-1     /MATCHES YOUR DEVICE NAME
          JMP I PATCH     /NO, CONTINUE WITH OPEN
          TAD (ENTRY      /YES, LOAD YOUR ENTRY POINT
          CDF 10
          DCA I (OUTDEV/OR INDEV
          JMP EXIT        /CONTINUE WITH THE NEXT COMMAND
YORDEV,   DEVICE NAME     /THIS IS YOUR DEVICE NAME
```

note: Output to an internal handler should not be 'closed' as this operation is reserved for the file output routines. To switch to another device, simply perform another 'O O' command.

SERVICING INTERRUPTS IN U/W-FOCAL:

One of FOCAL's virtues is that it runs with the interrupt system enabled. Although the standard version only uses this facility for running the terminal, there is room in the 'skip chain' for adding other user flag checks. In addition there is space for inserting instructions to remove unwanted flags which the user does not wish to acknowledge. If U/W-FOCAL ever fails to respond when loaded or after calling a non-standard device handler the problem is most likely an unknown flag which is requesting an interrupt. This may be cleared by halting the computer, loading address 100 and restarting. Press the CLEAR or I/O PRESET key if your computer has one, otherwise START will perform this function, resetting all the device flags. If the problem persists, consult with someone familiar with the I/O devices attached to your machine.

Some experience is necessary before attempting to add an interrupt driven task to U/W-FOCAL. Most programmers tend to feel a little uneasy (at last at first) in setting a device in operation and then returning to the interpreter with the assumption that it will take care of itself. Some of the concerns in programming such routines involve (1) the question of buffering; (2) queuing of service requests and checking for a 'done' state; (3) preventing premature returns to the monitor system. Devices which are commonly added to the interrupt system include clocks, digital plotters, light pens, power-fail/auto restart, and event flags.

Notes on the interrupt routines: The MQ is not ordinarily saved although there is a register (SAVMQ) available for it. It is also important to note that the teleprinter routine (TINT) expects the DF to be set to 0. Thus in the 8/e version which uses the SRQ instruction to save some overhead, it is important that any user routines reset the DF before returning to the skip chain. In the non-8/e versions the DF will be reset via the interrupt process but service is slower. PDP12 programmers may be pleased to note that it is possible to provide for LINC-mode interrupts in this version. The code at location 40 is easily relocated to provide room for a suitable patch.

PROGRAM-LEVEL INTERRUPTS

One of the novel features of FOCAL is the ability to service interrupts within the FOCAL program. These interrupts are limited to a rate not much greater than 20-30 per second and are only serviced when the program is actually running, but these conditions are met in many physiology experiments as well as other situations. Such servicing would not be appropriate for a real-time clock, on the other hand, since the clock would 'stop' whenever the program quit or was interrupted, even though the clock flag was continuing to be set.

The basic plan is to add 4 instructions to the interrupt routines which check for some event flag and mark its occurance by setting a software flag. Whenever the program reaches the end of a line of text it then checks this flag and if an event has occured an automatic 'DO 31' command is executed. The programmer then uses group 31 to respond to this interrupt, probably by sampling data, incrementing a counter, etc. and at the end of this group the program automatically continues with whatever it was supposed to do next. The implementation of this feature to be shown below, ignores the 'DO 31' command if there is no such group so that the code can be added without being used by every program.

Clearly the response time to such interrupts will not be 'instantaneous'. Furthermore the program must be written so that the text scan comes to the end of a line reasonably often. The following command, for example, would completely inhibit the interrupt feature:

```
3.20 IF (D)3.2,3.2;C ATTEMPT TO WAIT FOR AN INTERRUPT

31.1 SET D=1
```

The reason is that line 3.2 which is supposedly waiting for 31.1 to set 'D' never executes through an end-of-line so the software flag is never tested! A possible fix:

```
3.10
3.20 IF (D)3.1,3.1;C THIS EXECUTES THROUGH AN EOL
```

CHANGES TO ADD THE PROGRAM INTERRUPT FEATURE:

```
        FIELD 1

        *TABCNT+1       /PAGE-ZERO USER SPACE
FLAG,   -1              /SOFT WARE EVENT FLAG
        CRTRAP          /END-OF-LINE TRAP
        ERTRAP          /MISSING GROUP 31 TRAP

        *PC1
        JMP I FLAG+1    /CHECK THE EVENT FLAG

        *ZMA+7
        JMS I FLAG+2    /CHECK IF WE NEED GROUP 31

        *UINT           /ADD A NEW INTERRUPT TEST
        SKPFLG          /SKIP ON THE EVENT
        JMP .+3
        CLRFLG          /CLEAR THE HARDWARE FLAG
        DCA FLAG        /SET THE SOFTWARE FLAG

        *SOMEWHERE
CRTRAP, TAD FLAG        /ANY INTERRUPTS YET?
        SZA CLA
        POPJ            /NO
        ISZ FLAG        /YES, SET TO 'STANDBY'
        TAD P7600
        DCA LINENO      /SET POINTERS FOR GROUP 31
        DCA NAGSW
        PUSHJ           /EXECUTE A 'DO 31' COMMAND
            DO+3
        CMA             /RESET THE SOFTWARE FLAG
        DCA FLAG
        POPJ            /AND CONTINUE THE PROGRAM

ERTRAP, 0               /COME HERE IF GROUP 31 IS MISSING
        TAD FLAG
        SMA SZA CLA     /DO WE NEED IT?
        JMP I .+4       /NO
        TAD ERTRAP      /YES
        DCA I TABCNT    /CONTINUE THE
        JMP I TABCNT    /ERROR RETURN
        DOEXIT-2
```

SUMMARY OF U/W-FOCALs INTERNAL ROUTINES

The following list of internal subroutines has been prepared to help the beginning programmer 'feel his way around' in FOCAL.

Notation: X = don't care or unknown
U = unchanged
addr = address of data or entry point of a subroutine

FIELD 0 ROUTINES

Name	Function	Call DF	Rtrn DF	Call AC	Rtrn AC	Rtrn LINK
CLOSER	Closes an output file	0	0	0,-1	0	X
COMPARE addr-1 addr-1 different identical	Compares 2 blocks of data to see if they are the same (AC=-# of words/block) 1st return if different 2nd return if the same	0	0	0	7	X
DISMISS	Remove the USR	0	0	X	0	X
ERROR1	Lower field error call	0	1	X	0	X
ERROR0	Special error call which allows trappable errors	0	1	X	0	X
GETHND	Load a device handler	0	0	0	0	X
GETMON	Loads the USR	0	0	0	0	X
GTNAME	Read a dev:filename.ex	X	0	0	0	X
ICHAR0	Reads 1 char from input buff	0	1	0	char	X
IOWAIT	Waits for the output buffer to empty, turns off interupts	X	0	X	0	U
LJUMP list-1 difference	The equivalent of SORTJ	X	0	X	0	predic- table
LPOPF addr	The equivalent of POPF	0	0	0	0	X
LPUSHF addr	The equivalent of PUSHF	0	0	0	0	0
NUHEAD	Places name and date in header	0	0	0	0	X

Name	Function	Call DF	Rtrn DF	Call AC	Rtrn AC	Rtrn LINK
OCHARO	Writes 1 char to output buff	0	1	char		X
	1st return for echo				char	
	2nd return for non-echo				0	
OPEN	Performs "lookup" and "enter" functions via the USR	X	0	GOSW	0	0=lookup 1=enter
SAVE	Saves the program buffer	0	0	0	0	X
UNPACK	Prints the AC as 2 ASCII chars	X	0	valu	0	X

FIELD 1 ROUTINES

Name	Function	Call DF	Rtrn DF	Call AC	Rtrn AC	Rtrn LINK
ALIGN can't can	Align the binary points of FLAC, OPER	X	U	0	0	X
CONTINUE	Clear AC, get next command	1	1	X	0	X
DCAIAXIN	Store the AC in the text buff	X	1	valu	0	U
DELETE	Remove a line of text	X	1	0	0	X
DIV1	Shift OPER right, adjust EX1	X	U	X	0	X
DIV2	Shift FLAC right, adjust EXP	X	U	X	0	X
DUBLAD	Triple precision addition with link plus overflow	X	U	X	0	X
ECHOC	Print AC if the echo is on	1	1	char	0	X
EOF	Restores TTY I/O, returns a '_'	X	U	0	337	U
ERROR2-4	Print error message	X	1	X	0	X
EVAL ECALL	Performs an arithmetic evaluation - see prior discussion	1	1	0	0	X
FETCH	Performs a GETC or READC, depending if CHIN is 0 or 1	1	1	0	0	X
FINDLN	locate a line no. in buffer	X	1	0	0	X
can't	THISLN = next line					
can	THISLN = LINENO					

Name	Function	Call DF	Rtrn DF	Call AC	Rtrn AC	Rtrn LINK
FIXIT	Converts FLAC to 24-bit integer	X	U	X	LORD	0
FLOAT	Put AC into HORD, P13 into EXP and clear LORD, OVER	X	U	valu	0	U
FLOATR	Return signed AC as fn.value	X	1	valu	0	X
FLOATR	Return unsigned AC as fn.value	X	1	valu	0	X
GETC	Read CHAR from the text buffer	X	X	0	0	X
GETLN	Reads a lineno, sets LINENO, NAGSW	X	1	0	0	0=not all 1='all'
GRPTST	Compare group # in AC w/LINENO	X	U	0	0	X
different	1st return if different					
identical	2nd return if the same					
LPRTST	Decides if CHAR is a left-parenthesis	X	U	0	0	
no						1
yes						0
MULTIO	Multiply FLAC by ten, then add in calling AC	X	U	digit	0	X
NEGATE	Performs a 'CIA' on FLAC	1	1	0	0	0-nonzero 1-zero
NORMALIZE	Normalizes FLAC	1	1	X	0	X
ONTEST	Evaluates a lineno and performs a DO, GOTO, or 'NOP'	X	1	0	0	X
PACKC	Pack CHAR into the text buffer	1	1	0	0	X
POPA	Restores the AC with a 'TAD'	X	1	X	X+valu	U
POPF	Restores 4 words	1	1	0	0	-L
POPJ	Return from a PUSHJ	X	1	0	0	U
PRINTC	Print AC or CHAR	X	1	char 0	0	X
PRINTD	Print BCD digit in the AC	X	1	digit	0	X
PRINTN	Convert binary to BCD & print Convert to BCD & do not print	1	1	0 -1	bufr addr	X

Name	Function	Call DF	Rtrn DF	Call AC	Rtrn AC	Rtrn LINK
PRNTLN	Print a line number	X	I	0	0	X
PUSHA	Saves the AC on the stack	X	I	valu	0	0
PUSHF	Save 4 words on the stack	X	I	0	0	0
addr	=address of 1st word (Field 1)					
PUSHJ	Jump and save return address	I	I	0	0	0
addr	=entry point of routine					
READC	Get a character from INDEV	I	I	0	0	X
READN	Convert ASCII to binary	I	I	0	0	X
	AC=0 reads a char to start					
RETURN	Clear AC, return FLAC as value	I	I	X	0	X
REVERS	Performs a "CIA" on OPER	I	I	0	0	0-nonzero
						1-zero
RTL5	Implements RTL;RTL;RTL	X	U	valu	ansr	X
SGNCHK	Takes the absolute value of FLAC	I	I	0	0	
zero	Checks for 0 and sets T3					1-positive
non-0	Link=0 if arg. is negative					0-negative
SHIFTL	Shifts FLAC I position left	X	U	0	0	X
	Does not change EXP					
SHIFTO	Moves OPER -> FLAC	X	U	0	0	U
SORTJ	Table branch routine	I	I	code	0	X
list-1	=list of possibilities			or		
diff	=offset between lists			0=		
fail	=return point if not in list			CHAR		
SORTX	Sorts for comma, space semi-	X	U	0	0	X
yes	colon, carriage return					
no						
SPNOR	Use GETC to remove spaces	X	?	0	0	X

Name	Function	Call DF	Rtrn DF	Call AC	Rtrn AC	Rtrn LINK
TESTC	Decide if CHAR is a:	I	1	0	0	X
	terminator					
	function					
	number					
	variable					
TESTN	Decide if CHAR is a:	X	U	0	0	
	period					0
	neither					0
	number					1
TESTX	Check if CHAR is a terminator	X	U	0	0	X
	yes SORTCN is also set in this case					
	no SORTCN is not changed					
TSTCMA	Check if CHAR is a comma	X	?	X	0	X
	no					
	yes and removes comma with GETC					
XI33	Input routine for the terminal	I	1	0	char	U
XOUTL	TTY output routine	X	1	char	0	U
ABSOLV	Takes absolute value of FLAC	X	U	0	0	predictable
RESOLV	Restores proper sign to FLAC	X	U	0	0	predictable

Note for 8K programmers: Since both user routines and the symbol table must compete for the same core space, the symbol table is now created at run-time starting at the location found in FIRSTV. This allows new routines to be coded at the beginning of the user area and ended with a declaration 'STVAR=.' which defines the origin of first symbol. This value should then be patched into location 32 (Field 1).

A P P E N D I X   I

decimal ASCII codes for FIN( ), FOUT( ) FIND( ) and FTRM( ):

CODE	CHARACTER	CD.	CHAR.	CD.	CHAR.	CD.	CHAR
128	CTRL/SHFT/P	160	SPACE	192	@	224	`
129	CTRL/A SOH	161	!	193	A	225	a
130	CTRL/B STX	162	"	194	B	226	b
131	CTRL/C ETX	163	#	195	C	227	c
132	CTRL/D EOT	164	$	196	D	228	d
133	CTRL/E ENQ	165	%	197	E	229	e
134	CTRL/F ACK	166	&	198	F	230	f
135	CTRL/G "BELL"	167	'	199	G	231	g
136	CTRL/H "BACKSP"	168	(	200	H	232	h
137	CTRL/I H.TAB	169	)	201	I	233	i
138	LINE FEED	170	*	202	J	234	j
139	CTRL/K V.TAB	171	+	203	K	235	k
140	CTRL/L FF	172	,	204	L	236	l
141	CARRIAGE RETURN	173	—	205	M	237	m
142	CTRL/N SO	174	.	206	N	238	n
143	CTRL/O SI	175	/	207	O	239	o
144	CTRL/P DLE	176	0	208	P	240	p
145	CTRL/Q DC1	177	1	209	Q	241	q
146	CTRL/R DC2	178	2	210	R	242	r
147	CTRL/S DC3	179	3	211	S	243	s
148	CTRL/T DC4	180	4	212	T	244	t
149	CTRL/U NAK	181	5	213	U	245	u
150	CTRL/V SYNC	182	6	214	V	246	v
151	CTRL/W ETB	183	7	215	W	247	w
152	CTRL/X CANCEL	184	8	216	X	248	x
153	CTRL/Y EM	185	9	217	Y	249	y
154	CTRL/Z SUB	186	:	218	Z	250	z
155	CTRK/SHFT/K ESC	187	;	219	[	251	{
156	CTRL/SHFT/L FS	188	<	220	\	252	|
157	CTRL/SHFT/M GS	189	=	221	]	253	} or ALT MODE
158	CTRL/SHFT/N RS	190	>	222	^	254	~ or PREFIX
159	CTRL/SHFT/O US	191	?	223	_	255	RUBOUT

FOUT(141) will output a RETURN/LINE FEED; FOUT(13) will output a CARRIAGE RETURN only. Code 134 (CTRL/F) is U/W-FOCAL's break character. 225 through 250 are lower-case letters on some terminals. Many terminals use shift/K, /L, and /M for '[', '\', and ']'. Codes 0 through 127 are similar to 128-255 except that the parity bit is equal to zero.

A P P E N D I X   I I

ERROR CODES FOR U/W-FOCAL (V3N) 15 JULY 1976

00.00	RESTART FROM 10200
01.00	^F - KEYBOARD INTERRUPT
01.47	GROUP NUMBER LESS THAN 1
01.53	DOUBLE PERIODS IN LINE NUMBER
01.97	GROUP NUMBER GREATER THAN 31
02.06	NON-EXISTENT LINE REFERENCED IN A MODIFY OR MOVE COMMAND
03.00	NON-EXISTENT LINE CALLED BY GOTO, IF, JUMP, L B, OR L RUN
03.23	ILLEGAL COMMAND
03.47	NON-EXISTENT LINE REFERENCED BY DO, ON, L GOSUB, OR A FSF
03.66	NON-EXISTENT GROUP CALLED BY DO, ON, L GOSUB, OR A FSF
04.06	ERROR TO THE LEFT OF THE = SIGN IN A FOR OR SET COMMAND
04.25	EXCESS RIGHT TERMINATORS IN A FOR OR SET COMMAND
04.38	NO STATEMENTS FOLLOWING A FOR COMMAND
06.03	ILLEGAL USE OF A FUNCTION OR NUMBER: ASK,SET,FOR OR ZERO
06.73	TOO MANY VARIABLES
07.33	OPERATOR MISSING BEFORE A LEFT PARENTHESIS
07.85	INCORRECT FUNCTION SYNTAX (PARENTHESES MISSING?)
07.94	DOUBLE OPERATORS OR AN UNKNOWN FUNCTION
08.19	PARENTHESES DONT MATCH
10.52	PROGRAM TOO LARGE
11.34	INPUT BUFFER OVERFLOW (NOT WITH 8/E VERSIONS)
17.22	FRA INITIALIZATION ERROR: USE O I, THEN X FRA(-1,M)
17.33	FRA INDEX TOO LARGE, VALUE LIES OUTSIDE THE FILE AREA
17.62	FRA MODE ERROR. ONLY MODES 0,1,2,4 ALLOWED
18.42	FCOM INDEX TOO LARGE - (REDUCE PROGRAM SIZE?)
18.64	BAD SENSE SWITCH NUMBER (PDP-12 RANGE IS 0-5 ONLY)
18.77	NON-EXISTENT EXTERNAL SENSE LINE (RANGE IS 0-11)
19.:4	LOGARITHM OF ZERO REQUESTED
21.94	NEGATIVE ARGUMENT IN A FSQT CALL
22.99	NUMERIC OVERFLOW: TOO MANY DIGITS IN A STRING
23.20	OUTPUT ABORT OR CLOSE REQUESTED TOO MUCH SPACE
23.37	OUTPUT FILE EXCEEDED SPACE AVAILABLE (FILE WAS DELETED)
23.;5*	CANNOT OPEN OUTPUT FILE (TOO BIG, FILE ALREADY OPEN OR NO NAME)
24.05*	NO OUTPUT FILE TO RESTORE
24.20	ILLEGAL OPEN COMMAND
24.32	ILLEGAL RESTORE COMMAND
24.36*	INPUT FILE NOT FOUND (WRONG NAME?, WRONG DEVICE?)
24.48*	NO INPUT FILE TO RESTORE
24.61	NO INPUT FILE TO RE-READ
25.97	DEVICE DOES NOT EXIST ON THIS SYSTEM OR ILLEGAL 2-PAGE HANDLER
26.07	ILLEGAL LIBRARY COMMAND
26.15	ATTEMPTED LIBRARY OPERATION ON A NON-FILE STRUCTURED DEVICE
26.37*	FILE SPECIFIED IS ALREADY DELETED
26.75*	SAVE ERROR: NO NAME, DEVICE FULL, OR NOT A FILE DEVICE
27.06	PROGRAM REQUESTED NOT FOUND (WRONG NAME?, WRONG DEVICE?)
27.57	ATTEMPT TO LOAD THE INITIAL DIALOG
27.<2	ZERO DIVISOR
28.00	STACK OVERFLOW: REDUCE NESTED SUBROUTINES AND EXPRESSIONS

```
29.27   CANNOT USE THE <> CONSTRUCTION WITH OPEN OUTPUT
29.70   DEVICE ERROR (WRITE-LOCK OR PARITY ERROR)
30.30   ARGUMENT MISSING IN THE VIEW COMMAND
30.86   DOUBLE PERIODS IN A FILE NAME
31.<7   FUNCTION REFERENCED IS NOT YET IMPLEMENTED IN THIS VERSION

     *  MAY BE PROGRAMMED TO BRANCH TO A SPECIFIED LINE NUMBER
```

Recovery from a MONITOR 2 error due to a write attempt on a write-locked device:

```
1) .ST 100        /just plain .ST won't work
2) *L D DEV=      /'DEV:' is (was!!) the write-locked one
3) *CONTINUE      /but be sure to write-enable it this time!!
```

A P P E N D I X   I I I

Since there is no Initial dialog the following simple ODT patches may prove useful:

```
03155/ 7610 6213  add CR/LF before an error message (12k version)
03766/ 7610 6213  ditto for the 8k version
05517/ 5337 4347  add four more secret variables (&,',",\)

10413/ 7000 4547  add line number printout to MODIFY/MOVE
11216/ 2533 4533  add the ":" printout to ASK
15574/ 4534 7200  eliminate the initial space printed by TYPE
15772/ 0375 0376  change ALTMODE to PREFIX
15716/ 7001 0376; 15725/7144 1316 change leading zeros to spaces

14446/ 1104 7000  allow FCOM to use locations above 7600
14452/ 4562 7000; 14456/ 6221 6241  place FCOM in field 4

10033/ 4565 4465  protect PDP-12 patches in the 8k version
17413/ 7421 6212; 6141;314;2;5555 modify FMQ for a PDP-12 without EAE

10033/ 4565 5204; 12170/5020 2733;2733;2733  remove FLOG, FEXP and FATN
                  to gain more variable space in the 8k version
10033/ 5204 5312; 12166/5210 2733;2733  remove FSIN and FCOS to gain
                  still more variable space in the 8k version.
```

*** I N D E X ***

```
---INPUT---,ECHO .........echo input onto output device
---OUTPUT---,ECHO ............echo output onto terminal
```

.FC & .FD are the program & data file name extensions,
.FB & .FH are the binary and help file name extensions

<> enclose required terms. [] enclose optional terms.
one letter abbreviations may be used as command words.
X represents a variable. E1, E2 and E3 are arithmetic
expressions. L1, L2 and L3 are line numbers. G1 is a
line or group number. L1-L3 and G1 can be replaced by
arithmetic expressions.

FUNCTIONS		PAGE
*FABS	ABSOLUTE VALUE	52
*FADC	ANALOG-DIGITAL CONVERSION INPUT	52
*FATN	ARCTANGENT	52
*FAVR	SIGNAL AVERAGER FUNCTION	
*FBLK	STARTING BLOCK NUMBER OF INPUT FILE	53
*FBUF	DISPLAY BUFFER STORAGE	53
*FCLK	CLICK (TIME OF DAY)	
*FCOM	STORAGE	53
*FCOS	COSINE	54
*FCTR	FREQUENCY COUNTER	
*FDAC	ANALOG OUTPUT	
*FDAY	CHANGE THE SYSTEM DATE	54
*FDIS	DISPLAY FUNCTION (SEE ALSO VIEW COMMAND)	54
*FDVM	DIGITAL VOLTMETER INPUT	
*FEXP	EXPONENTIAL	55
*FFT	FOURIER TRANSFORM	
*FIN	CHARACTER INPUT	55
*FIND	CHARACTER SEARCH ON INPUT	55
*FITR	INTEGER PART OF EXPRESSION	55
*FJOY	CURSOR OR JOYSTICK	41
*FLEN	FILE LENGTH OR FREE BLOCKS DETERMINATION	56
*FLOG	NATURAL LOGARITHM	56
*FLS,FLSW	READ LEFT SWITCHES (PDP-12)	59
*FMQ	DISPLAY IN MULTIPLIER QUOTIENT REGISTER	56
*FNEW	USER FUNCTION	
*FOUT	CHARACTER OUTPUT	56
*FRA	RANDOM ACCESS FILE FUNCTION	57
*FRAC	FRACTIONAL PART OF ARGUMENT	58
*FRAN	RANDOM NUMBER GENERATOR	58
*FRS,FRSW	READ RIGHT SWITCH REGISTER (PDP-12)	59
*FSGN	SIGN PART OF EXPRESSION	58
*FSIN	SINE	58
*FSQT	SQUARE ROOT	58
*FSS,FSSW	TEST SENSE SWITCHES (PDP-12)	58
*FSR	READ SWITCH REGISTER	59
*FTIM	MONITOR ELAPSED TIME	59
*FTRG	SCHMITT TRIGGER INPUT	
*FTRM	TERMINATOR FROM ASK COMMAND	59
*FXL,FXSL	TEST SENSE LINES ON PDP-12	59
*FXTR	UNDEFINED USER FUNCTION	
*F	FOCAL STATEMENT FUNCTION	60

- - - - -

c
this manual for U/W-FOCAL was prepared by
Paul Diegenbach,
Zoological Laboratory
University of Amsterdam
Pl.Doklaan 44
Amsterdam, The Netherlands.

with material from the
manual for DMSI PS/8 FOCAL and
notes and comments by Jim van Zee.
send questions concerning the manual
and indicate errors to
Paul Diegenbach.

send questions, notes concerning
U/W-FOCAL to its originator:

Jim van Zee
Dept. of Chemistry/BG-10
University of Washington
Seattle, Washington 98195

150776N

DATE FIX FOR U/W-FOCAL, DECUS FOCAL8-301

The date routine in U/W-FOCAL (Version 1) was not designed to work beyond the end of 1977 since no provision had been made at the time it was written (1974) for extending the system date beyond that time. Consequently dates in 1978 are printed as '1970', and in general, dates will cycle through 70-77 every eight years. The following patch may be used to add the necessary offset so that the proper year is shown. Note that the offset will have to be changed every eight years, and in any case will not work after 1999.

```
.GE SYS UFOCAL               (see listing pp 16, 16-1)
.ODT
14510/ 7440 7000                       NOP
14460/ 4317 0767                       AND I (7666
14461/ 1767 1264                       TAD BASEYR
14462/ 0366 4274                       JMS PACK2
14463/ 4317 7410                       SKP
14464/ 1061 0116     BASEYR,           116
↑C
.SA SYS UFOCAL             (Do it twice to put the program
.SA SYS UFOCAL              back in its original location.)
```

Note that the last location, 14464, will have to be changed every eight years. The table below gives the proper value:

1970-1977:	0106	
1978-1985:	0116	(Shown above)
1986-1993:	0126	
1994-1999:	0136	

The version distributed from the library will always have the current base year patched. Other versions (currently V4D) are available form the author.

15 August 1978 JvZ