FOCAL

HOW TO WRITE NEW SUBROUTINES AND USE INTERNAL ROUTINES

DOUG WREGE
Engineering Experiment Station
Georgia Institute of Technology
Atlanta, Georgia

TABLE OF CONTENTS

FOCAL: HOW TO WRITE NEW SUBROUTINES AND USE INTERNAL ROUTINES

FOCAL: HOW TO WRITE NEW SUBROUTINES
AND USE INTERNAL ROUTINES*

ABSTRACT

It is the aim of this paper to help the user to code specific routines in FOCAL so that his dialect of FOCAL can be applied to his application (without being forced to understand in detail all the workings of FOCAL). In this way, perhaps, each and every user can make his particular dialect of FOCAL 'perfect'.

I. INTRODUCTION

Many users have found FOCAL ** to be the answer to their real-time and computational problems. The language is extremely powerful and flexible with unique text editing and debugging features. Although FOCAL is slow in execution compared to machine language coding, for most real-time problems or one-time calculations, lack of speed is not a serious handicap. Most users will agree that a program can be written, debugged, and executed in "FOCAL" before the equivalent could even be coded (and/or punched) in any other language. Additions or changes are easily made.

It will be assumed that the reader has a basic knowledge of PDP-8 processor instructions, PAL mnemonics (see Digital's Small Computer Handbook or Introduction to Programming), as well as a familiarity with the Floating Point Package (DEC-08-YQYA-D). In addition, he should be familiar with the "FOCAL"** language.

As many users have discovered, the internal workings of FOCAL are an incredibly complex piece of programming. With the need to interface the computer to specialized equipment for individual applications, there is the corresponding need for appropriate software. If FOCAL could communicate with this equipment, one would have an extremely powerful and flexible computation and control package. This paper is an attempt to explain how user developed software can be interfaced to the basic FOCAL package, without requiring the user to spend valuable time trying to understand all of its detailed workings.

Section II will deal with a general discussion of how FOCAL works, in a descriptive fashion. Section III will be concerned with the philosophy of the language. The last few sections will be more technically oriented toward helping the user actually code his additions. Finally, several examples and ready coded routines, which may be used to simplify the user's problems, are included.

**Throughout this paper a "FOCAL" program written in the "FOCAL" language will be enclosed in quotes. The machine language coding of the FOCAL interpreter will be referenced by the word FOCAL without quotes.

II. ASSEMBLERS, COMPILERS, AND INTERPRETERS

In general, there are three routes that the programmer can follow for machine execution. Programs that perform translations are assemblers, compilers, or interpreters; each operate from conceptually different vantage points.

In a compiler level language, such as FORTRAN, ALGOL, and BASIC, coding is written in a syntax close to the way a human thinks. A compiler interprets this and generates an object code which is close to machine language. This, in turn, is translated into actual machine language instructions. Finally these machine language instructions must be read into core before execution. If any corrections are to be made to the program (debugging, additions, or corrections), one must recompile the source coding, read the new object coding in, and finally execute it.

An assembly level language is inherently closer to machine language than a compiler level language. The user's coding is indeed remote from the way he thinks about formulating a problem (he is even forced to think in binary or octal, the machine's way of formulationg problems). About all an assembler lets the programmer do is use mnemonics (words) and symbols instead of binary numbers. For example, in the PAL language, the instruction TAD I TEMP is assembled as follows from the definitions:

$$TAD = 1\emptyset\emptyset\emptyset_8 \qquad \text{/in the assembler's internal symbol table}$$

$$I = \emptyset4\emptyset\emptyset_8 \qquad \text{/internal symbol table}$$

$$TEMP = \emptyset1\emptyset\emptyset_8 \qquad \text{/user defined in coding}$$

The assembler masks out the first 5 bits from the last mnemonic if there are more than one (in this case TEMP); it then ORS the result with the other mnemonics:

```
     1ØØØ
&    Ø4ØØ
&    Ø1ØØ
     ────
     15ØØ      This is the machine equivalent.
```

The PAL assembler is a little more sophisticated than this, of course, and performs functions a little more complicated, but generally an assembler is incredibly stupid for what it can do. Note the similarity between PAL mnemonics and machine language. Throughout the following sections various mnemonics will be defined so that the PAL assembler can generate instructions compatible with FOCAL (e.g. GETC = 45Ø6 causes the assembler to add this to its symbol table).

In an interpretive level language, no machine language coding is generated for execution. An interpreter is essentially a subroutine caller. It contains a subroutine for every conceivable operation it thinks the user wishes to perform. If it cannot understand what the user wants, it prints an error message and waits for the user to make himself clear. Every character that the user inputs is stored in core. Upon execution the interpreter "interprets" the program character by character and calls the subroutine indicated.

FOCAL is an interpretive level language. In particular, it is a recursive interpreter (see Section III). That is, unlike FORTRAN, one may call a function from within itself. Nevertheless, it is basically a subroutine caller, even though these subroutines may be incredibly interlocked. It has a subroutine to evaluate arithmetic expressions (EVAL), subroutines to make it recursive (PUSHJ, PUSHA, etc.), branching routines (SORTJ), a subroutine to find a certain line (GETLN), one to get a character (GETC), etc. Once the user understands what all these routines do, he can add his own coding in a highly efficient and powerful manner. Descriptions of these subroutines will be given in later sections.

III. THE PHILISOPHY OF FOCAL

A. Text Editing

Since FOCAL is an interpretive language, it must have facilities for manipulation of user written text. In order to facilitate these manipulations, there are a number of text formatting and editing features, such as WRITE, MODIFY, TYPE, and the "trace" ("?") function. One of the main features of the FOCAL interpreter is the simplicity of concept and power of operation of the format controlling statements. The user finds a convenient, easily understood way of controlling the format of his output, regardless of his level of programming experience and sphistication.

Since much of FOCAL execution is involved in various text decoding routines, FOCAL is slow in execution of programs (compared to assembly or compiler language coding). The text handling routines may be called from the user written assembly language subroutines, and thus are listed with a short description of their function, in Table 1.

FOCAL is concerned with interpreting what the user's text means by specific combinations of characters, so it must have a flexible means of decoding these characters according to type. The most efficient way this can be done is to use a subroutine (SORTC) that compares the present character with a list. It is necessary to have the address of the list as an argument for this subroutine. For example, suppose that it is desired to find a text terminator. To do this, a list is made of all legal terminators (;, carriage return, space comma, etc.), and the value of the present character (stored in location CHAR) is compared to the list: if a match is found, an index is set to the list element number, and a normal return is taken. If a match is not found, then another return is taken.

B. The Multiple Branch Routine

FOCAL is in many ways similar to JOSS[2]. All of the JOSS-like languages incorporate a "command" in addition to the arithmetic statements available in other languages (ALGOL, FORTRAN). One of the advantages of the command is that, using only the first symbol of a new statement, the interpreter (or compiler, in the case of BASIC) can decode the action required, and thus need not "understand" the whole line before proceeding. This is an advantage in a small machine such as the PDP-8, where the paucity of core demands highly efficient coding.

[2] Joss – An Introduction to a Helpful Assistant, Rand Memos 5058-PR July 1966.

A Unique feature of FOCAL is the ability to operate with single-letter abbreviations of the command. As an example, consider the subroutine that actually selects the command branches (and is used for other operations within FOCAL, as well). This routine (SORTJ) is called with an argument pointing to the list of characters to be compared and another argument containing a pointer to a list of associated addresses. FORTRAN programmers might recognize the result as a sort of character-driven computed GOTO. The calling sequence is:

```
SORTJ              /Sort and Branch Routine
TABLE1-1           /pointer to character list
TABLE2-TABLE       /difference in addresses of the tables
XXX                /return if not in table
```

Absolute addresses are specified in the arguments; hence, tables may be stored between pages.

Since FOCAL refers to lists for its decoding operations, it is often referred to as a table driven interpreter. A table driven interpreter is especially suited to addition of new coding, since only one or two addresses need to be added to a table (list) for a new branch.

C. Recursion

One of the features of FOCAL which makes it so powerful is that of recursion. Recursion is the ability of a subroutine to call itself, e.g. FSQT (1 - FSQT(X)). In most compiler level languages this operation is carried out by repeating the machine language (FSQT) coding so that one version of the subroutine can call the other. In these cases the subroutine never really calls itself, rather it calls a separate identical piece of coding. An interpretive level language cannot afford multiple identical subroutines for every possiblity, since it would take too much core.

Consider how a 'normal', nonrecursive subroutine works. Schematically we may divide the subroutine into a segment in which the logical operations are coded and a segment where temporary values in the calculation are stored. We can consider the subroutine return to be stored in this temporary storage area also. VIZ,
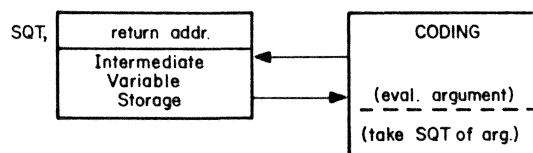
If this hypothetical subroutine were to call another subroutine (as is normally done in assembly language), there would be no difficulties provided that the intermediate storage of the two subroutines are separate.

If the subroutine was to call itself from within its own coding, the original intermediate values of the variables and the return pointer would be overwritten (as the program executes the coding the second time). If there was a way to use a different intermediate storage area, the original values would not be lost.

4

The Push-Down List (PDL) concept involves an intermediate storage area which is "pushed-down" (making a new intermediate storage area available) whenever a subroutine is called and "popped-up" whenever a return occurs.   VIZ,
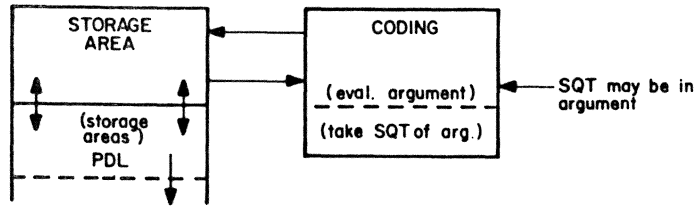
To continue the example, the steps in the evaluation of FSQT 1-FGST(X)) would proceed as follows:

1. The main program calls the FSQT subroutine. Storage area 1 is now pushed-down into the push-down list making area 2 available.

2. The argument "1-" is evaluated up to the next FSQT(X).
    In order to evaluate this, the FSQT subroutine is <u>called again</u> !

3. On second entry to the subroutine, storage area 2 (containing the main program return and the intermediate value of the argument) is pushed-down.

4. X is evaluated and then the square root is taken.

5. The subroutine returns (to the middle of itself) with the answer FSQT(X).
    When this return is effected, storage area 2 is popped-back-up (with the old intermediate values).

6. The answer FSQT(X) is subtracted from 1 to form the argument 1-FSQT(X).
    The square root of <u>this</u> is taken and the function returns to the main program.

Obviously, by using the PDL concept, subroutines may call themselves to any level (as long as there is PDL space available).

For most efficient core utilization, FOCAL uses the same PDL intermediate storage for all subroutines. To do this, one value (PDP-8 word) is pushed-down at a time. Values are 'popped' in the reverse order that they are 'pushed'.

An additional feature of a PDL is that it can be used for temporary storage of variables in non-recursive routines. One may consider the PDL as an extension of page zero since it can be accessed from any page. Section V will describe PDL handlers available in FOCAL.

D. Conclusion

 The concepts outlined above will introduce the experienced programmer to the internal working of FOCAL. In the sections that follow, a more technical exposition of these routines will be given.

TABLE 1

FOCAL TEXT HANDLERS

MNEMONIC	DESCRIPTION
GETC	Get the next character from the text
SORTC	Sort the present character against the table
TESTN	Sort the present character into one of three types
TESTC	Sort the present character into one of four other types
TESTLPR	Test CHAR from left parenthesis
READC	Read a character from the Teletype
PRINTC	Print CHAR on Teletype
PACKC	Pack a character into buffer (store it)
PRINTLN	Print the current line number
FINDLN	Find a given line
SPNOR	Ignore spaces

The Appendices contain examples elucidating the principles outlined in this report.

IV. TECHNICAL DETAILS - GENERAL

A. Arithmetic Manipulations

Arithmetic is done using the three word floating point format. Input and output of numbers are handled via the Floating Point Package (FPP) I/O controller (with modifications to run with the interrupt enabled). For details, see FPP documentation (DEC-08-YQYA-D).

B. Storage - (Core Layout)

The FOCAL interpreter occupies locations 1 - 3220 (see Figure 1). The FPP occupies approximately 4600 - 7577, depending on how many functions are kept. The initial dialogue sets BOTTOM, the end of storage space, depending on the number of functions kept. The remaining storage is used for text, variable storage, and push-down lists.

3220 - 4577	with all functions
3220 - 5177	FEXP, FLOG, FATN deleted
3220 - 5232	FSIN, FCOS and above deleted

The text is built up from location 3220 occupying approximately two characters per location. Variables are built upward from the top of the text. They occupy 5 locations per variable and are created as they are found in execution. Whenever the indirect program is changed, (modified, appended, or collapsed), a new starting point for variables is indicated; hence, old variables are erased. The push-down list (explained more fully later) is built from the FPP down toward the variable storage area. Error messages occur with termination of the program whenever these lists overlap.

Instructions are stored in the command/input buffer when in the command mode; the buffer has sufficient locations for one line of characters.

C. Holes

The following locations are free for the user:

PAGE ZERO	16	(Auto Index Register)
	162 - 175	(Free in 4K FOCAL)
	171 - 175	(Free in 8K FOCAL)
FPP	5571 - 5577	
	5754 - 5777	
	6171 - 6177	
	7154 - 7177	
	7346 - 7377	
	7554 - 7577	
	6317 - 6377	is used by the high-speed reader control -- if you do not have one, this is available

```
        0000   ┌─────────────────────┐
               │     PAGE  ZERO      │
               ├─────────────────────┤
               │       FOCAL         │
               │    INTERPRETER      │
        3232   ├──────────┬──────────┤
               │   TEXT   │   AREA   │
               │        ⇩           │
               ├──────────┼──────────┤◄──── ENDT
               │   VARI   │  ABLES   │◄──── FIRSTV
               │        ⇩           │
               │                    │◄──── LASTV
               │        ⇧           │
               ├──────────┼──────────┤◄──── BOTTOM
        4600   │   PUSH   │  DOWN    │
        5200   ├──────────┴──────────┤
        6400   │   EXT. FUNCTIONS    │
               │      FLOATING       │
               │       POINT         │
               │      PACKAGE        │
        7600   ├─────────────────────┤
               │  LOADERS-MONITORS   │
        7777   └─────────────────────┘
```
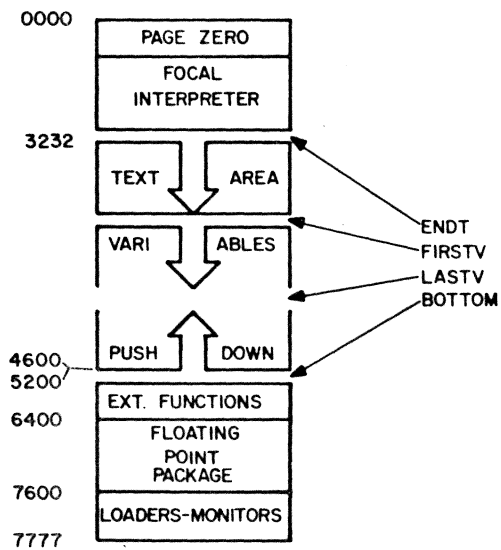
TEXT STORAGE FORMAT

LINE	
A	B
C	D
ASCII CHAR	
77	15
	C.R.

VARIABLES FORMAT

NA	ME
SUBSCRIPT	
±	EXP
±	MAN−
TISSA	
etc.	

Figure 1

8

D. Moving Bottom

For additional user coding room, BOTTOM may be changed at the sacrifice of text storage To move BOTTOM, set the contents of location 27 (C(27)) to the last location available for text (PDL) storage; e.g. in order to free locations 4420-4577 for user additions to the interpreter, change C(27) to 4417.

V. TECHNICAL DATA - FOCAL SUBROUTINES

With the use of subroutines available in the FOCAL interpreter and a listing, a must, it is relatively simple to write powerful user coded additions.

Unless otherwise stated, these subroutines must be entered with the AC = $\emptyset$; they return with the AC = $\emptyset$.

A. Page Zero Reference Locations

CHAR - The contents of this location (142) contains the current character (in ASCII code) from the text buffer.

SORTCN - This register contains references used by sorting routines (see below).

FLAC - This is the first word of the floating accumulator (contains the exponent). The floating accumulator occupies locations 44 - 46.
FLAC is defined as 44.

B. Text Handling Routines

GETC = 4506
Gets next character from the text; exits with next character is CHAR.

SORTC = 4511

Calling sequence:	SORTC	/call
	LIST-1	/address of LIST-1
	XXX	/return if in LIST
	XXX	/return if not in LIST

Description: If the accumulator is nonzero, its contents are used; otherwise the contents of CHAR are used to sort against the LIST. If it is in the LIST, return to call + 2; if not, return to call + 3. SORTCN is set to how far down in the list the match occurred.

Example: If we are testing for one of the following:

LIST	= .
254	/,
273	/;
215	/carriage return
7777	/list is terminated by a negative number

9

Assuming it is an error for CHAR not to be in the list, the following coding applies:

```
SORTC          /sort against LIST
LIST-1         /address of LIST
SKP
ERROR          /do an error exit as not in LIST
```

If a match were found, SORTCN would have the values:

Contents of CHAR	SORTCN Value
,	0
;	1
carriage return	2

NOTE: Lists are terminated by negative numbers.

PRINTC = 4512
Print the accumulator; if the AC = 0, print the contents of CHAR.

READC = 4513
Read and echo a character from the keyboard. Put it into CHAR.

SPNOR = 4521
Ignore spaces in text; exit with the first character that is not a space in CHAR.

ERROR = 4526
Used to exit upon error detection; transfers control to the command mode and terminates execution; prints error message. (In the FOCAL listing there are ERROR2, ERROR3, and ERROR4. All of these are identical.)

TESTN
This subroutine is actually a series of SORTC's with various returns:

```
CALL:     TESTN            /call
          return1          /return if a period
          return2          /return if not a period or a number
          return3          /return if a number; SORTCN is set to the
                            binary equivalent.
```

This routine tests only CHAR. AC must be 0.

TESTC   (4525)
This subroutine is actually a series of SORTC's with various returns:

```
CALL:     TESTC                    /call
            return1                /terminator; SORTCN set according to TERMS
            return2                /number; SORTCN set as in TESTN
            return3                /function; (CHAR=F)
            return4                /alphabetic character
```

SORTJ   (4510)

This subroutine is used as a multiple sort and branch routine. CHAR (or the AC if nonzero) is compared to a list. If it is in the list, an address is looked up and an effective JMP ADDRESS is executed. If a match is not in the list, then return is to call+3.

```
CALL:     SORTJ
          LIST1-1              /ADDRESS of character list
          LIST2-LIST1          /difference in the addresses of lists
          RETURN               /return here if not in LIST1
```

An example of this is the FOCAL branch to a library command:

```
          POPA                 /get command CHAR
          SORTJ                /branch
          COMLIST-1
          COMGO-COMLIST
          ERROR2               /invalid command
```

where

```
          COMLIST = .              COMGO =  .

          323 /S    (ASCII)        SET         /ADDRESS OF SET CODING
          306 /F                   FOR         /ADDRESS OF FOR
          311 /I                   IF
          304 /D                   DO
          307 /G                   GO
          303 /C                   COMMENTS
          301 /A                   ASK
          324 /T                   TYPE
          314 /L                   LIBRARY
          -                        -
          -                        -
          -                        -
          7777 /list is terminated by a negative number
```

NOTE: Lists are terminated by a negative number.

C. Utility

RTL6 = 4520
Rotate the AC six places to the left.

D. Pushdown List Controllers

For those unfamiliar with more powerful processors than the PDP-8, the ideas of recursion and pushdown lists are explained in Section II. These subroutines appear to simulate hardware commands on more sophisticated machines like the PDP-10 and even use the same mnemonics !

PUSHA = 4503
Puts the contents of the AC on the PDL; clears the accumulator.

POPA = 1413
Get the top entry on the PDL and put it in the AC. (Note: auto-index register 13 is the pointer to the pushdown list; thus 'POPA' is actually TAD I 13.)

PUSHF = 4504
This is essentially three PUSHA's and is used for storage of floating point data.

      Call: PUSHF
                  ADRESS   /address of first location of three word floating point number.

POPF = 4505
The inverse of the PUSHF routine.

      Call: POPF
                  ADDRESS   /address of where to put data.

PUSHJ = 4501
This is the recursive subroutine call. The subroutine return is put on the PDL and a <u>JMP</u> to the subroutine address is executed.

      Call: PUSHJ
                  SUBROUTINE   /address of SUBROUTINE
                  XXX   /address of this location is
                                    /stored on the PDL

POPJ = 5502
Recursive subroutine return; the top element of the PDL is used as the effective address of the return.

E. Other Subroutines

INTEGER
Enter via a JMS I INTEGER. This routine makes an integer out of the FLAC. The low order part is in FLAC + 2, the high order part is in FLAC + 1. Also, returns with the low order part in the accumulator.

EFUN3I
This routine is the return from a function routine. It checks for a right bracket in CHAR (')') and normalizes the floating accumulator. Enter via a JMP I EFUN3I.

EVAL
This subroutine evaluates arithmetic expressions; because it is recursive, it must be called via:

```
        PUSHJ
        EVAL
        XXX          /return
```

The subroutine return is to call + 2 with the floating point value of the expression it evaluated in the FLAC. (How EVAL works is discussed in Appendix A.)

NOTE: All temporary storage must be in the PDL before calling EVAL. This data must be restored after the return. (see Appendix for examples.)

VI. LINKS TO FOCAL

A. Functions

The general form of a function in "FOCAL" is FUNC(ARG1,ARG2,---). The function coding is entered via a SORTJ where the address in designated in the table:

```
        FNTABF = .           /(376) in FOCAL-W  8/68
              XABS           /address of FABS coding
              XSGN           /FSGN
              XINT           /etc.
              XDIS
              XRAN
              XDXS
              XADC
              ATN
              EXP
              LOG
              SIN
              COS
              SQT
              NEW            /user defined function
```

To add a user coded function put the entry point of the function coding in the appropriate location in the above table. FOCAL will branch to that location after the function name is decoded, and ARG1 is evaluated in the floating accumulator (FLAC). To delete a function from the list, replace the current contents with 2725.

13

When the function evaluation is complete, the answer must be left in the FLAC, and a JMP I EFUN31 executed. The EFUN31 routine will check to see if there is a right parenthesis (")") in CHAR, and normalize the FLAC, before returning to the appropriate place in FOCAL. (See Hints and Kinks, Section XIII A, if the answer is an integer.)

B. Links to FOCAL - the LIBRARY Command

FOCAL has an unimplemented command, the LIBRARY command (SET, ASK, TYPE, etc. are commands). The general form of a command is:

X _ (any syntax allowable by coding).

For example the SET command's allowable syntax is:

SET _ (variable) = (arithmetic expression).

To generate the link to the user's LIBRARY command, put the entry address in 1201. FOCAL will enter via a JMP with CHAR containing 240_8 (a space). The following coding may be used at the end of a LIBRARY command to space over extraneous characters to a semicolon or carriage return, which must be in CHAR before doing an effective JMP PROC to return to FOCAL:

```
SKP         /entry
GETC        /fetch the next character
SORTC       /sort for a ; or c.r.
GLIST-1
JMP PROC    /FOUND IT !
JMP .-4     /not yet
```

C. Debugging

It has always been a problem to debug FOCAL programs, as FOCAL runs with the interrupt on. Recently, a DECUS program XOD (DECUS #8-89) became available. This program may be used in field 1 to debug FOCAL in field Ø with the following patches made by J. C. Alderman.

FIX UP XOD

Patch FOCAL	0001	5575
(field Ø)	0175	2603
	6761	5002
Patch XOD	6762	0002
(field 1)	6763	5404
	6764	0003
	6765	6613
	6766	0004

VII. ACKNOWLEDGEMENTS

VIII.   APPENDIX A

A. A Prescription

To add a function:

1. Put the function address in FNTABF.

2. Do coding.

    a. Use PDL for temporary storage

    b. If more than one argument is needed:

        PUSHJ
        ARG

        where ARG is a supplied subroutine (See Appendix B). ARG is
        a subroutine which moves past commas and evaluates arithmetic
        statements, leaving the result in the FLAC.

3. Put the functional result in the FLAC.

4. Return to FOCAL via JMP I EFUN3I.

To add the LIBRARY command:

1. Put the initial address in the contents of 1201
    (for expansion of commands see Appendix B).

2. Exit from coding via an effective JMP PROC. Note: the contents of CHAR
    must be either ; or a carriage return.

X. APPENDIX C

A. Example of a Recursive Subroutine – EVAL

The subroutine EVAL is an example of a recursive subroutine. The PDL is used to defer evaluation so that the arithmetic operations are performed according to operand priority.

In order to take care of bracketed quantities EVAL does the following:

> if a left bracket occurs – PUSHJ
> EVAL
> if a right bracket occurs – POPJ .

Given that EVAL evalueates arithmetic expressions, the above operations have the effect of changing all bracketed quantities to evaluated numbers. Hence, all bracketed quantities have now "gone away" and we are left with expressions like:

$$A + B*C - D/E \uparrow F .$$

Operand priority is assigned as follows:

opperation	priority level
+	1
-	2
*	3
/	4
↑	5

A flow diagram approximating this subroutine is given in Figure 2.

IX. APPENDIX B

A. A Few Useful Routines

1. Argument evaluator

A common requirement, expecially in function additions, is a routine which test for and evaluates additional arguments. The subroutine ARG (coded below), checks if the contents of CHAR is a comma (,), moves past the comma, evaluates the argument, and returns to call + 3. If the contents of CHAR is anything other than a comma, return is to call + 2.

```
        Call:           PUSHJ
                        ARG
                        XXX         /CHAR was not a comma
                        XXX         /return with ARG(next) in FLAC

        ARG,            TAD CHAR    /get CHAR
                        TAD MCOMMA
                        SZA CLA     /A comma?
                        JMP .+4     /yes: exit via POPJ
                        PUSHJ       /move past comma and evaluate next arg.
                        EVAL-1
                        IAC         /increment return
                        POPJ
```

2. LIBRARY expansion

As FOCAL has only one 'extra' command character, LIBRARY, a routine to expand the number of commands is useful. In this way the normal format:

$$L \_ \text{(statement)}$$

which allows only one command branch, may be extended into the syntax:

$$L \_ X \_ \text{(statement)}$$

where X represents another command. A listing of this routine follows.

3. Function-command extention

The user may desire to perform a branch within a function, e.g. ARG2 in the function call FNEW (ARG1,ARG2,ARG3, ---) may be used as a command letter to specify a branch to perform different operations. An example of a subroutine to do this follows. (see next page)

NOTE: The return to FOCAL from each branch must be via a JMP I EFUN3I.

With the use of the last two routines, the number of commands and/or functions may be extended to any level.

```
/
*COMCO+10
        LIPPAP
/
*7346
/COMMAND PROCESSOR
/
LIPPAP, SPNOP    /IGNOPP SPACES
        TAD CHAP          /GET COMMAND CHAP
        PUSHA             /STASH IT
        GPTC              /GET NEXT
        SORTC             /MOVE TO TERMINATOP
          CLIST-1
        SKP
        JMP .-4
        SPNOP             /IGNOPP SPACES
        POPA              /GET COMMAND CHAP
        SORTJ             /GO THEPP
          CLIST-1
          GOLIST-CLIST
        FPROP             /NOT IN LIST
/
/
*5571
CLIST=.
323     /SWAP
322     /PFSTOP
320     /PUT
7777    /COMMAND LIST TFRMINATOP
/
/
*6171
GOLIST=.
SWAP
FFSTOP
PUT
/
/

*
```

```
/
/FOCAL COMMAND DECODER
/
FDEC,    JMS I INTFGER      /MAKE ARGUMENT AN INTEGER
         PUSHA              /SAVE IT
         TAD CHAR           /COMMA SHOULD BE NEXT
         TAD MCOMMA
         SZA CLA
         ERROR4
         GETC               /MOVE PAST COMMA
         SPNOP              /IGNORE SPACES
         TAD CHAR           /GET COMMAND CHAR.
         PUSHA              /STASH IT
         SORTC
          TERMS-1           /IGNORE REST OF NAME
         JMP .+3            /IN LIST
         GETC       /GET NEXT AND IGNORE
         JMP .-4
         SPNOP              /IGNORE SPACES
         POPA               /GET COMMAND CHAR
         SORTJ
          COMMANDS-1
          ADDS-COMMANDS     /GO TO APPROPRIATE ROUTINE
         ERROR4             /NOT IN LIST
MCOMMA,  -254
/

*
```

SIMPLIFIED FLOWCHART OF SUBROUTINE EVAL

EVAL

set LASTOP=0

Evaluate a variable
(possible link to
functions)
(check for brackets)
(check for terminators)
answer in FLAC

get THISOP
SORTC for priority

THISOP >
LASTOP
?

yes

no

DONE
?

yes → **POPJ**

no

do LASTOP
between FLAC &
var. on top of PDL

POP- up
new LASTOP

put THISOP
IN LASTOP

push-down
variable(FLAC)

push-down
LASTOP
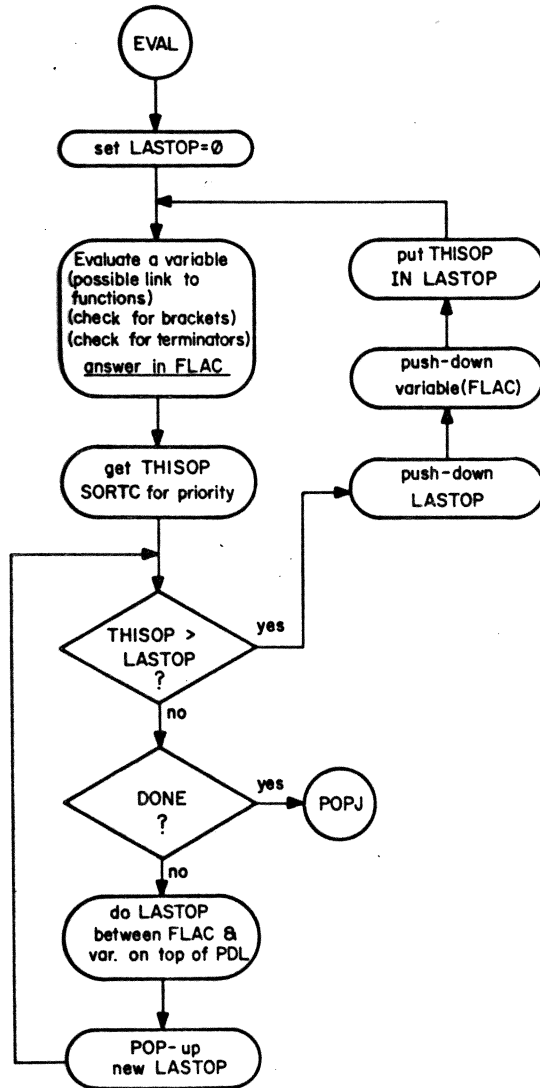
Figure 2

Two locations, LASTOP and THISOP, contain the priority assignment of the present and last operands respectively. The steps in the evaluation of

$$A + B*C - D/E\uparrow F$$

would be:

THISOP	LASTOP	FLAC	PDL	EXPLANATION
N.A.	Ø	A		evaluate A into FLAC; lastop starts out Ø.
1	Ø	A		plus has priority 1
	1		A Ø	THISOP higher than LASTOP; put LASTOP and FLAC in PDL
	1	B	A Ø	evaluate B into FLAC; put THISOP into LASTOP
3	1	B	A Ø	THISOP has priority 3 – *
	3		B 1 A Ø	THISOP higher than LASTOP; put LASTOP and FLAC in PDL put THISOP into LASTOP
	3	C	B 1 A Ø	evaluate C into FLAC
2	3	C	B 1 A Ø	– has priority 2
2		C*B	1 A Ø	do the last operation between FLAC and top of PDL.
2	1	C*B	A Ø	get new LASTOP from PDL

THISOP	LASTOP	FLAC	PDL	EXPLANATION
	2		C*B 1 A Ø	THISOP higher than LASTOP put LASTOP and FLAC in PDL put THISOP in LASTOP
	2	D	C*B 1 A Ø	evaluate D
4	2	D	C*B 1 A Ø	/ has priority 4
	4		D 2 C*B 1 A Ø	THISOP higher than LASTOP put LASTOP and FLAC in PDL put THISOP into LASTOP
5	4	E	D 2 C*B 1 A Ø	↑ has priority 5 evaluate E
	5		E 4 D 2 C*B 1 A Ø	THISOP higher than LASTOP put LASTOP and FLAC in PDL put THISOP into LASTOP
Ø	5	F	(same as above)	evaluate F no more operations so this operation has priority Ø

THISOP	LASTOP	FLAC	PDL	EXPLANATION
Ø	4	E↑F	D 2 C*B 1 A Ø	THISOP lower than LASTOP do LASTOP with top of PDL get new LASTOP from PDL
Ø	2	D/E↑F	C*B 1 A Ø	THISOP lower than LASTOP do LASTOP with top of PDL get new LASTOP from PDL
Ø	1	C*B-D/E↑F	A Ø	(same as above)
Ø	Ø	A+B*C-D/E↑F		(same as above)

THISOP LASTOP Ø hence we are done: do POPJ exit

24

XI.   APPENDIX D

A. Field One Variable Array

1. Abstract

A new form of 8K FOCAL W. (DEC-08AJAD-PB), is available which uses field one to store data arrays in three word floating-point form. This facility is added to 4K FOCAL W via the function call FNEW. The function may be called recursively to any level, and all of the features of FOCAL are retained. In addition, an ERASE or ERASE ALL command will not wipe out the array. Hence, variables may be stored for use in successive programs.

2. Requirements

Fits into unused locations in the Floating-Point Package (DEC-08-YQYA-PB)

$$7154-7177$$
$$6572-6576$$
$$5755-5764$$
$$7554-7577$$

3. Usage

Loading

Load after FOCAL W. has been loaded into the machine (before or after initial dialogue). Restart FOCAL W. at 200_8.

Calling sequence

To store a variable Z as array element J:

$$\underline{*}\ S\ \ X = FNEW(J, Z)$$

or
$$\underline{*}\ 4.3\ S\ \ X = FNEW(J, Z)$$

In addition X will be set equal to Z.

To get the data from array element K and set Z equal to this element:

$$\underline{*}\ S\ \ Z = FNEW(K)$$

i.e. If there is only one argument the instruction is interpreted as a 'GET'. If there are two arguments it is interpreted as a 'PUT'. In the above examples the arguments may be any arithmetical expression that can be evaluated.

25

C. Recursive calling

The function FNEW may be called recursively at any level. VIZ,

$$\underline{*}\ S \quad Z = FNEW(J, FNEW(J +10))$$

sets Z FNEW(J+10) <u>and</u> stores FNEW(+10) in array element J.

$$\underline{*}\ 3.2\ S \quad Z = FDXS(J*1000)\ +\ FDIS(FNEW(J)*NORM)$$

The arguments may be any arithmetical expression. The following are valid:

$$\underline{*}\ S \quad Z = FNEW(J*M-3,\ FEXP(X*2)*Y)$$

$$\underline{*}\ S \quad Z = FNEW(J, FNEW(J)*FEXP(FNEW(L)))$$

4. <u>Description</u>

The function FNEW protects the binary loader in upper core. The user, of course, may subdivide his array into any number of smaller arrays, keeping track of his own indecies.

```
ARG     5755
BOTTOM  0027
CHAR    0142
FFUN3I  0100
FNT     7573
FNT     0077
ERROR   4526
EVAL    1603
FFNT    4407
FFNT    0000
FLAC    0044
FLIST1  0603
FMUL    3000
FNEW    7154
FNTARR  0376
GET     7554
GLIST   1406
IGNOR   0217
ILIST   0761
INTEGR  0052
IRETN   0231
MCOMMA  0163
MCR     0065
POPA    1413
POPJ    5502
PUSHA   4503
PUSHJ   4501
PUT     7564
P7600   0024
RRADC   4513
SETUP   6572
SOFTC   4511
SOFTJ   4510
SPNOP   4521
STARTV  0134
THREE   7173
TLIST   1407
T2      0157
```

```
                    /
                    /
                    /
                    FIELD 0
                    /
                    /PAGE ZERO CONSTANTS
                    /
                    *163
0163    7524        MCOMMA, -254
                    /
                    /
                    *FNTABF+15
0413    7154                    FNEV    /PUT ADDRESS IN FNTABF
                    /
                    /
                    *7154
                    /FIELD ONE FNEV VARIABLES
                    /CALL:  FNEV(ARG1)       /GET ARRAY ELEMENT ARG1
                    /       FNEV(ARG1,ARG2) /PUT VALUE OF ARG1 IN ARRAY ELEMENT
ICE                                         A
                    /
                    /
7154    4407        FNEV,   FENT    /ENTER FPP
7155    3373                FMUL THREE      /MULT. ADDRESS BY THREE FOR THREE .
7156    0000                FEXT            /FP STORAGE
7157    4452                JMS I INTEGER   /MAKE IT AN INTEGER ADDRESS
7160    7500                SMA             /BEGIN CHECK FOR OVERWRITING LOADER

7161    5366                JMP .+5         /O.K.
7162    1056                TAD 56          /+2XX
7163    7700                SMA CLA
7164    4526                ERROR           /MUST PROTECT LOADER
7165    1046                TAD FLAG+2      /GET ADDRESS OF ARRAY
7166    4503                PUSHA           /STORE IN PDL
7167    4501                PUSHJ           /EVALUATE ARG2
7170    5755                 ARG
7171    5777                JMP GET         /ARG2 EXISTS; GET DATA
7172    5776                JMP PUT         /PUT DATA AWAY
7173    0002        THREE,  2               /CHANGE THIS FOR TWO WORD
7174    3000                3000            /OR INTEGER STORAGE
7175    0000                0000
                    /
                    /
                    /
7176    7564
7177    7554
```

```
                *5755
                /EVALUATE AN ARGUMENT; IF NOT
                /THERE RETURN TO CALL+2 VIA POPJ
                /IF THERE TO CALL+3
                /
5755  1142  ARG,    TAD CHAR
5756  1163          TAD MCOMMA
5757  7640          SZA CLA /IS IT A COMMA?
5760  5364          JMP .+4              /NO:ARG2 MISSING
5761  4501          PUSHJ
5762  1602           EVAL-1
5763  7001          IAC                  /INCREMENT RETURN
5764  5502          POPJ     /DO SUBROUTINE RETURN
                /
                /
                *7554
7554  4777  GET,    JMS SETUP            /SET UP POINTER TO DATA
7555  1416          TAD I 16             /GET EXPONENT
7556  3044          DCA FLAC
7557  1416          TAD I 16             /GET HIGH ORDER MANTISSA
7560  3045          DCA FLAC+1
7561  1416          TAD I 16             /GET LOW ORDER
7562  3046          DCA FLAC+2
7563  5373          JMP END
7564  4777  PUT,    JMS SETUP
7565  1044          TAD FLAC
7566  3416          DCA I 16             /PUT AWAY EXPONENT
7567  1045          TAD FLAC+1
7570  3416          DCA I 16
7571  1046          TAD FLAC+2
7572  3416          DCA I 16
7573  6201  END,    CDF                  /RESTORE DATA FIELD
7574  5500          JMP I EFUN3I         /DO FUNCTION RETURN
                /
                /
7577  6572

                *6572
                /SET UP POINTER TO ARRAY IN XR-16
                /CHANGE TO DATA FIELD 1
                /
6572  0000  SETUP,  0
6573  1413          POPA     /GET ADDRESS
6574  3016          DCA 16
6575  6211          CDF 10
6576  5772          JMP I SETUP
                /
                /
```

XII. APPENDIX E

A. Disk Variable Storage

1. Abstract

This FOCAL overlay is equivalent to the FIELD ONE variable addition to FOCAL described in Appendix D. In this case, however, variables are stored on the Disk.

2. Comments

The contents of location 167 (BASE) must be set for the user's machine configuration. Disk variables are written on the disk from BASE upward. BASE is the disk extended address of the lowest used location.

e.g.

last 4K of one disk system $C(167) = 700_8$

last 8K of two disk system $C(167) = 1600$

last 16K of two disk system $C(167) = 1400$

The present listing is for the last 4K on a two disk system, i.e. $C(167) = 1700_8$.

```
AFC      5755
BASE     0167
BOTTOM   0027
CA       0164
CHAR     0142
FRUN3I   0100
FNT      0077
FFIOF    4526
FVAL     1603
FENT     4407
FENT     0000
FLAC     0044
FLIST1   0603
FMUL     3000
FNEU     7154
FNTALF   0376
GLIST    1406
IGNOF    0217
ILIST    0761
INSTF    7565
INTECF   0052
IIFTN    0231
MCOMMA   0162
MCF      0065
MOFF     7554
POPA     1413
POPJ     5502
PUSHA    4503
PUSHJ    4501
P43      7574
P700     7172
P7600    0024
FFAD     0165
FFADC    4513
FTL6     4520
SOFTC    4511
SOFTJ    4510
SPNOF    4521
STAFTV   0134
THFEE    7173
TLIST    1407
T2       0157
VC       0163
WFITE    0166
```

```
/DEFINITIONS* FOCAL
/
CHAR=142
PTL6=4520
PUSHA=4503
POPA=1413
PUSHJ=4501
POPJ=5502
EVAL=1603
INTPCPT=52
FPLOP=4526
FLAC=44
SOFTJ=4510
SOFTC=4511
EFUN3I=100
STARTV=134
IFFTN=231
VCI=65
TLIST=1407
FLIST1=603
BOTTOM=27
FNT=77
T2=157
P7600=24
IFALC=4513
CLIST=1406
SPNOP=4521
IGNOP=217
FENT=4407
FMUL=3000
FEXT=0
ILIST=761
FNTALF=376
/
/
/
FIELD 0
```

```
                    /
                    /
                    FIELD 0
                    /
                    /PAGE ZERO CONSTANTS
                    /
                    *162
0162    7524    MCOMMA,   -254
0163    7750    WC,       7750
0164    7751    CA,       7751
0165    6603    READ,     DMAR
0166    6605    WRITE,    ....
0167    1700    BASE,     1700
                    /
                    /
                    /
                    /LINK TO FOCAL
                    *FNTABF+15
0413    7154              FNEW
                    /
                    /
                    /
                    /
                    *5755
                    /EVALUATE AN ARGUMENT; IF NOT
                    /THERE RETURN TO CALL+2
                    /IF THERE TO CALL+3
                    /
5755    1142    ARG,      TAD CHAR
5756    1162              TAD MCOMMA
5757    7640              SZA CLA
5760    5364              JMP .+4
5761    4501              PUSHJ
5762    1602                EVAL-1
5763    7001              IAC
5764    5502              POPJ
                    /
                    /
                    *7154
                    /DISK FNEW
                    /
7154    4407    FNEW,     FENT              /ENABLE 3-WORD FP NUMBER
7155    3373              FMUL THREE
7156    0000              FEXT
7157    4452              JMS I INTEGER     /MAKE AN INTEGER
7160    4503              PUSHA             /PUSH DISK MEM. ADL.
7161    1045              TAD FLAC+1        /GET HIGH ORDER PART
7162    4520              RTL6              /SHIFT FOR EXTENDED ADDRESS
7163    0372              AND P700          /MASK FOR EXTENDED BITS
7164    1167              TAD BASE          /ADD DISK BASE ADDRESS
7165    4503              PUSHA             /SAVE DEA
7166    4501              PUSHJ             /EVALUATE ARG2
7167    5755                ARG
7170    7344              STA CLL RAL       /-2 FOR READ
7171    5777              JMP MORE          /SAVE DATA
7172    0700    P700,     700
7173    0002    THREE,    2
717
```

33

```
7173   0002   THREE,    2
7174   3000             3000
7175   0000             0
                        /
                        /
7177   7554

               *7554
7554   1166   MORE,     TAD WRITE        /MAKE DISK INSTRUCTION
7555   3365             DCA INSTR
7556   1413             POPA             /GET IRC
7557   6615             IFAL
7560   7346             STA CLL RTL      /TRANSFER 8 WORDS
7561   3563             DCA I WC
7562   1374             TAD P43          /INTO FLAG
7563   3564             DCA I CA
7564   1413             POPA             /GET DMA
7565   0000   INSTI,    0
7566   6002             IOF              /DISABLE INTERR.
7567   6622             DFSC             /DONE?
7570   5367             JMP .-1          /NO WAIT
7571   6601             DCMA             /MASH FLAGS
7572   6001             ION
7573   5500             JMP I FFUNGI     /DO A FUNCTION RETURN
7574   0043   P43,      43
                        /
                        /
```

XIII. APPENDIX F

A. Hints and Kinks Department

For the experienced programmer the following may be helpful.

1. Location EVAL-1 contains the subroutine call GETC. Hence, to move past a character and evaluate an argument one may:

```
            PUSHJ
            EVAL-1
```

2. The first instruction in the POPJ subroutine is TAD I 13. Hence, for multiple returns from a subroutine one may POPJ with the AC nonzero, e.g. if the AC is 1, return is to call +3 instead of call +2 (as in a normal POPJ return).   VIZ,

```
            PUSHJ           /call
             SUB
            XX              /normal return
            XX              /POPJ return if AC= 1 when POPJ
                              called
            XX              /return if AC= 2
                            /etc.
```

In all cases the subroutine will return with the AC = Ø.

3. When using signed and unsigned integers core must be taken that minus zero is not in the FLAC since EFUN3I normalizes the FLAC. (FOCAL will 'hang' in that event.) The following coding will apply for unsigned integers.

```
            CLL RAR         /make sure sign bit is Ø
            DCA FLAC +1
            RAR
            DCA FLAC + 2    /put carry bit away
            TAD P14
            DCA FLAC        /put exponent in
            JMP I EFUN3I
```

for signed integers:

```
            CLL RAL
            SNA
            CLL             /make sure positive Ø
            RAR
            DCA   FLAC + 1
            DCA   FLAC + 2
            TAD DCA FLAC
            JMP I EFUN3I
```

35

4. There is a BUG in FOCAL. The RMF in the interrupt routine must be moved to just prior to the ION. This will not give trouble until field one coding is added.

5. For hardware initialization when FOCAL recovers (Control–C) one may use location 2775.

6. For machines without a high-speed reader, additional coding room of 6320–6377 may be gained by overwriting the HRS routine. To remove the * command deposit 2725 in location 1207.