DECUS NO.	FOCAL8-271
TITLE	Modification of FOCL/F for Data Acquisition and Control
AUTHOR	Douglas E. Wrege
COMPANY	Georgia Institute of Technology Nuclear Research Center Atlanta, Georgia
DATE	
SOURCE LANGUAGE	

PREFACE

FOCAL^{TM} has often been described as a "plastic" language since it may be molded into a form suited to an individual's needs and requirements. By "plastic" it is meant that the language structure, syntax, and commands, are modifiable into a form specified by the user. A truly plastic language must have, among its attributes, as few as possible syntactical rules, be easily modified, and above all be available to the user (i.e. an "open shop" language). FOCAL is not significantly more plastic than any other interpreter, however, since it is list driven and syntactical limitations are few compared to BASIC, FORTRAN, ALGOL, etc. it does a reasonable job of fulfilling the first two requirements. Availability of core is always a problem in a mini-computer, but if not too many changes are required it is not impossible to find space. Complete reworking of the interpreter itself is difficult since FOCAL was never intended to be a plastic language, but if the user is satisfied with most of the commands it is a reasonable starting point for scientific or arithmetically oriented problems. The large selling point is that FOCAL is the closest thing to an "open shop" language currently available, and this requirement must be met before talk can begin. The FOCAL interpreter has probably been modified by more non-systems programmers than all "accepted" languages currently in use. For this reason alone it perhaps warrents the tag "plastic".

This paper is intended to aid users in molding FOCAL to suit their own requirements as I have been doing for several years. As this paper will probably only be read by so called "FOCAL freaks" I will tend to be a little jargony and informal in its presentation. For those puriests at heart I apologize. It is, of course, impossible to describe in detail how all of FOCAL works in a finite period of time, but I hope that it will suffice to describe those routines most commonly used when adding user written code to the interpreter.

A few words are in order about that peculiar breed of animal called programmers. They are in general proud, competitive and usually extremely creative. They are quick to flaunt new creative ideas in the face of other programmers often implying a lack of creativity on their part. Conversely, a programmer will quickly maintain a defensive if not antagonistic position whenever their work is threatened by such comments. One must remember that good programmers have the tendency to take credit for an entire piece of work when they are only responsible for a segment or even a small addition to someone elses work. This heightens the antagonism between programmers. I know that I have been guilty of the above faults more than infrequently. The original creator of FOCAL, Rick Merrell, has very justifiably objected to user modification of FOCAL, especially since his interpreter was so free of flaws. Consequently, would be FOCAL "plasticizers" have been frowned on by Mr. Merrell and conversely the appreciation which Mr. Merrell deserves has often been omitted. The generation of a new language that has received so much attention and modifications by so many users should be looked upon with pride by its creator and sponsor. They have created something that is more than just a new language. I for one would have to plead guilty on all charges stated above. Mr. Merrell certainly deserves more credit than I for the work I have done in the past three years. However, being one of that arrogant breed called programmers, all I can bring myself to say, in print, is THANKS RICK!

MODIFICATION OF FOCL/F FOR DATA ACQUISITION AND CONTROL

D. E. Wrege
Small Computer Applications Lab
Nuclear and Biological Sciences Division-EES
and
The School of Nuclear Engineering
Georgia Institute of Technology
Atlanta, Georgia

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). Included are descriptive discussions of how FOCAL works, the philosophy of the language, and sections technically oriented toward helping the user actually code his additions. This paper is an extention of DECUS FOCAL 8-17 and includes most of the discussions contained therein. The particular versions of FOCAL described will be FOCAL/69 and FOCAL/F, the latter being a version of 8K FOCAL/69 with modifications by the author allowing assembler patches to be more easily added.

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.

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 some BASICs, 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 formulating 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 = 1000_8	/in the assembler's internal symbol table
I = 0400_8	/internal symbol table
TEMP = 0100	/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:

**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.

```
   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.

In a 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 clearer. 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. Thus an interpretor never generates machine language (object code) from the source code. It was once said that when evaluating arithmetic expressions the difference between a compiler and an interpreter is that a compiler comes up with code that can calculate the answer when loaded and executed while an interpreter merely comes up with the answer.[1] Thus an interpreter is at an advantage when doing one time calculations but at a disadvantage when calculating iteratively (since it must reinterpret the source code each time through).

A few words should be said about the relative merits of assembly, compiler, and interpretive language programs. Assuming good coding for all three, assembly language code will always be the most efficient code for a small computer. That is to say execution time will be minimized and the minimum core storage will be used. However, seldom is anyone willing to code a problem at this level since coding time will far outweigh the savings in machine time. Consequently assembly level coding is usually restricted to problems which are run as production calculations, or for the case of rate-limited problems, or because the programmer is hung up on elegance or just plain doesn't know better. Once again assuming equally efficent coding for compilers and interpreters (not always a valid assumption) the compiler will win in execution time of object code. This is primarily because the interpreter must operate on source code to decide what to do while the compiler has already figured that out when generating the object code. For single evaluations (non-iterative problems) the interpreter will win if compile and load time is added to the compiler's object run time. Where an interpreter looks bad is in nested [FOCAL]"FOR" or [FORTRAN]"DO" loops since the interpreter must reinterpret the source code each time through. At any rate, no matter how much one argues, compiler level languages generally look better with respect to execution speeds than interpreters, but interpreters will win when it comes to conserving core storage. For example, the FOCAL command F J+1,1E4;S A(J)-FSQT(J) takes 12 core locations in a 12 bit machine (packing two characters per word). The code generated by a compiler typically takes from a factor of 5 to 10 more core. One must remember that since most compiler level languages were originally designed to run on large machines, the name of <u>their</u> game is "save every microsecond" whereas interpreters like FOCAL were written with the basic premise "save every possible core location". The result is a time-core storage tradeoff. Small machine users, with limited core will find that interpreter level languages are probably the best route to take and FOCAL is an extremely efficient interpreter.

[1] R. P. Warnock III, Private Communication, Chemistry Department at Emory University, Atlanta, Georgia

Philosophy

<u>Rules and Syntactical Limitations</u> - In general, the fewer the rules and syntactical limitations of a language the simpler and more flexible the language will be. Conversely, few rules tend to increase the number of typographical errors which are not caught by syntax checks. FOCAL has very few rules, which tend to make it one of the simplest to learn and most powerful small machine interpreters around. The basic rules are as follows:

1. All indirect program lines must be numbered according to group number and line within the group, (GROUP NUMBER). (LINE IN GROUP) where
   $1 \leq$ GROUP NUMBER ≤ 31
   Ø1 $\leq$ LINE IN GROUP ≤ 99
   Hence, line numbers may run from Ø1.Ø1 to 31.99 (excluding XX.ØØ)
2. A line may contain any number of commands, (except WRITE, MODIFY, and ERASE) separated by semicolons and lines are ended by a carriage return. A command has the form
   (COMMAND) (SPACE) (STUFF) (TERMINATOR)
   where the command name must only contain the correct first character to specify the command (case of FOCAL/69) or the correct first one or two characters (FOCAL/F). (That is "SET", "S", or "SEBXSXYZ" are valid in FOCL/F, but not SABC). The command name must be followed by a space. The syntactical form of "STUFF" is determined by the particular command, and commands are terminated by a semicolon or carriage return.
3. Variables are specified by a one or two character name, <u>not</u> starting with "F" (see below) and a 12-<u>bit</u> subscript (a subscript of 4096 and Ø are the same). All variables are floating variables.
4. All functions start with F and contain parentheses. i.e., FNAM (ARGUMENT).

These are the basic rules for the FOCAL interpreter. As one can see they are relatively few and easy to remember.

Text Editing and Debugging

A valuable feature of the FOCAL language is the editing and debugging features. Since an interpreter stores the source code internally and must have routines for handling text, the price which must be paid for a built in editor is small. Consequently FOCAL has many unique commands which facilitate modification of text found only in editors. In addition, there is the trace feature, which when properly used, can cut debugging times by as much as an order of magnitude.

Interpretation

The interpretation of the source code is largely "table driven". This is to say there is a table in core which contains a list of all valid command characters, and a second table of dispatch addresses.

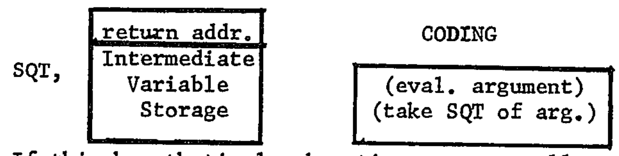
To interpret an individual command FOCAL picks up the first non-space character (or the first two, if present, in the case of FOCL/F) and searches a list of commands ("COMLST"). If a match is found, at say the k^th location in the list, then an address is fetched from the k^th location of a second list ("COMGO") and control is transferred to that address. In addition there are lists of characters which have the same or similar attributes. e.g., "TERMS" is a list of expression terminators, "ATLIST" is a list of "ASK" command special characters, "FNTABL" is a list of valid function names, etc. The virtue of a list driven interpreter is the ease in which additions or changes may be made to the language, i.e. often by merely inserting a new character into a list. This feature is in part what makes FOCL "plastic".

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 subtoutine 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 possibility, since it would take too much core. (Note the ubiquitous tradeoff.)

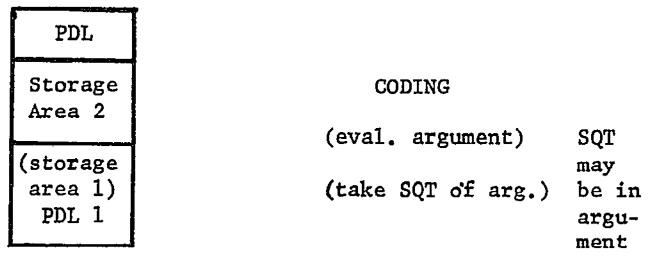
Consider how a 'normal', nonrecursive subroutine works. Schematically we may divide the subroutine into a segment in which the logical operations are coded and a segment where temporary values in the calculation are stored. We can consider the subroutine return to be stored in this temporary storage 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.

The Push-Down Stack or 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-FQST (X)) would proceed as follows:

1. The main program calls the FSQT subroutine. Storage area 1 is now "pushed" 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 called again!
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 this 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.

Data Acquisition Techniques

Data acquisition and experimental control situations are invariably real time problems. Since the original high level languages were not designed to run in real time, data acquisition tasks have historically been accomplished using pure assembly level coding. This approach was primarily dictated by the cost of computers, which required multiprocessing and/or time sharing of resources to maintain economic feasibility. It was ridiculous to tie up a multimillion dollar machine with a data acquisition task which only used a fraction of the resources of the computer. Consequently, when computer controlled data acquisition tasls were undertaken a special version of an operating system was usually written (in machine language) which ran the experiment in real time while using the rest of the computer resources for other tasks. With the advent of the mini-computer it became possible to dedicate an entire machine to data acquisition tasks. The machine language approach was maintained at first because of the absence of high-level languages around which to build a control system. As the flexibility and ease of use of high-level languages became apparent, not to mention the availability of these languages, data acquisition tasks were undertaken more and more frequently in these languages. Unfortunately, more often than not, the implementation of high-level languages on mini-computers was accomplished by system programmers attempting to simulate as closely as possible the high-level languages of the large scale computer. These programmers had been so far removed from real-time programming approaches that oftentimes the implementation of the language was not well suited to real-time data acquisition tasks. In addition, when the languages were adapted for data acquisition, the wrong approach was taken, leading to the widespread opinion that interpreters and the like could not handle these types of tasks. Before considering techniques of data acquisition in high level languages some general statements reguarding advantages of this approach should be given.

The primary advantage of using an interpreter is the flexibility which may be gained. Machine language coding, although more efficient in most ways, is such a monumental undertaking and so difficult to modify that programmers attempt to program the most general case anticipated. Consequently, the number of parameters which need to be supplied to the program becomes cumbersome and usually some segment of the program is not even used in a specific experiment. In addition, eventually there comes the time when the experimentalist wishes to do something outside the scope of the existing program but tends to use standard control algorithms purely because it is so difficult to modify the program. By this time, it is also common for the author of the data acquisition program to be in parts unknown, which makes the task of modification almost insurmountable. On the other hand, with an interpreter, programming the logical sequence of steps to run the experiment is sufficiently simple that it is not unreasonable to change the high-level language program for each and every experiment. As long as the interpreter can communicate with the special hardware without prescribing the acquisition technique there is almost complete flexibility in the software. In addition, since the data acquisition program is written in a high-level language, the experimentalist is continuously aware of exactly what functions he is performing in acquiring the data. Certainly the degree of "black boxiness" of data acquisition should be minimized in a research environment.

A second advantage of using a high-level language is the reduction in software development time and consequently cost. It is not unusual to spend 6 man-months to two man-years in developing a moderately sophisticated machine-language data acquisition system. What is seldom realized is that the majority of the coding is concerned with input of parameters to determine how to run the data acquisition task, calculations based on these parameters, reduction of the data, and outputing the results. The routines which actually concern the external hardware involve typically less than 10% of the total programming effort. By using an interpreter as the 90% base for building a system, the effort required to produce a sophisticated system is reduced, typically by an order of magnitude.

For almost all data acquisition systems, involving a high-level language, some machine language coding must be added. The idea of a general purpose data-acquisition high-level language is almost preposterous. FOCL/F, the subject of this paper, does not really intended to be more than a <u>basis</u> for a data acquisition system. The user must write some machine language functions or commands to do the actual data acquisition, otherwise the program is inefficient and clumsy at best. Probably one could not do what they wish by simply using FOCL/F without modification. However, FOCL/F is an attempt of a version of FOCAL which is easily modifiable (or plastic). To my knowledge there are very few (or no) examples of high-level languages which are expressly intended to be modified.

Now that we have established the need for some assembly level additions to the interpreter let us consider several classes of data acquisition and control tasks.

1. <u>Low Data Rates</u>: This is of course the easiest case for an interpreter to handle, and it is often assumed that this is the only case to which an interpreter may be applied. We shall put off trying to refute this misconception until following paragraphs. It should be noted that in the field of research almost 80% of the tasks may be put in this class. Physicists, for example, tend to gravitate toward the barely doable problem because those problems are the most interesting. In fact, if the experiment is not data-rate limited it should probably be left to those less fortunate in wealth of equipment. Biologists, medical people, psychologists and the like are dealing with relatively low data rates due to the nature of the Biological Life Scale. On occasion they may be taking data at high rates for smoothing purposes, but a more reasonable approach might be to smooth during the process of data collection, once again resulting in moderate data rates. Enough said about this class of applications.
2. <u>Moderate Data Rates</u>: By moderate data rates we are talking about 1-10 KC, probably in short bursts. The reason I say short bursts is that if the volume of data is large one is probably not expecting to do any analysis and hence little feedback control of the experiment. However one may be interested in, say, several hundred to several thousand pieces of information at 1-10 KC with a relatively long wait between bursts of data, in which analysis is undertaken. The <u>wrong</u> approach to taking this data is to link the data taking process (the "time") to the interpreter through the interrupt service routine. It is a result of this technique that interpreters are judged incapable of performing these tasks. A more efficient technique is to set up the data collection task in the interpreter by passing arguments to the interrupt service routine and then let the data collection proceed entirely under interrupt control. When the task is completed the interpreter can be informed to proceed with analysis. A second method is to set up the data collection arguments in the high-level language and turn the interrupt OFF for the duration of data collection. Measurements may then be passed to the high-level language for analysis after completion of I/O. In this manner data collection can proceed at full machine language speeds with no speed sacrifice from the interpreter.
3. <u>High Data Rates</u>: For cases with high data rates the latter technique discussed above is the only reasonable approach. This of course is true whether one is using an interpretive language or machine language. The one limitation is the case where large volumes of data are taken, continuously, at a relatively high data rate. For these cases one is not usually expecting to do any immediate analysis and probably will not do analysis on a mini-computer in any event.

Thus far little mention has been made about hardware design to optimize data acquisition or experimental control with an interpreter. The availability of complex logical functions that integrated circuit technology has made possible, coupled with decreasing core cost makes sophisticated data acquisition and control interfaces possible. Wherever it is possible the experimentor should design hardware which allows the computer software to do what it does best - make logical decisions. For example, when doing interval timing, it is relatively inexpensive to design a clock which alerts the computer after (computer) specified intervals. A stepping motor interface can either step the motor one time for each IOT, or have a register which is loaded with the total number of steps by a single IOT resulting in automatic hardware stepping of the motors. Scope display could utilize refreshed display via data break or non-refreshed display via a storage scope. High-rate data acquisition could be via data break rather than under AC transfer. The point that is being made is that a knowledge of both hardware and software should go into the design of any real time system. The phrase "Hardware-Software Tradeoff" is a very real one.

Inside FOCLF

By effective utilization of some of the powerful routines and subroutines in FOCL/F, additions of user code to the interpreter may be rapid and efficient. The user, of course, must understand something about the structure of FOCL/F and what the available subroutines do. The rest of this document will be concerned with descriptions of the function of these subroutines. It is intended that these descriptions be used only as a guide in the examination of the internal workings of FOCL/F. Those users who wish to make sophisticated additions or changes to the interpreter will need to examine the routines for detailed workings. Those users who wish to merely add simple functions should find these descriptions adequate for their purposes, however it is strongly recommended that no one attempt to make modifications without first consulting a listing.

As it is expected that most FOCL/F modifications will be in the form of user written functions, most users should perhaps skip to the section entitled "Addition of Functions". The other sections will more fully explain some of the routines referred to in that section.

It should also greatly improve the learning process to read this paper side by side with a listing of the interpreter, although that is not required. In the following paragraphs words in upper case will usually refer to a mneumonics defined in the listing.

Although this paper refers specifically to FOCL/F, most of the discussions may be applied to the standard FOCAL-8, or any other version of FOCAL. It will, of course, be mandatory that the user obtain a listing of that version they wish to modify in order to find which areas of this discussion apply. In developing FOCL/F, the internal philosophy of FOCAL, as layed out by Mr. Merrill, has been followed, hence there should be no philosophic variations.

Fields-Core Layout

FOCL/F is an 8K version of FOCAL/69, or FOCAL-8. The current version (12/1/72) is assembled to reside in fields Ø and 1. The major body of the interpreter resides in field 1, termed the program field or field "P", and almost completely fills that field. The interrupt service routine, error recovery, extended functions, and the dynamically allocated storage resides in the text field or field "T". The dynamically allocated area is occupied by the user text, push-down stack, and variables. A schematic core map is given in Figure 1.

Arithmetic Routines

FOCL/F does all of its arithmetic operations with a modified version of the Floating Point Package (FPP). The standard package has been modified so that it can be called from any field even though it resides in field P. There are two entry points; in both cases the FPP must be entered with the data field set to the field of call. These calls are:

```
            CDF X   / X=current field*10
            CIF P   / FPP in field P=1Ø
      JMS I (FPNTX  / enter FPP
              XXX   / pseudo FPP instructions
and
            CDF X
            CIF P
            JMI I (FPNT.
```

The entry to FPNTX will initialize the floating data field to be field P. The floating data filed may be changed at any time via the pseudo - FPP instruction:

```
      FCDF X  / X-desired field*10(8).
```

The pseudo - FPP instructions are:

```
      FGET=0000
      FADD-1000
      FSUB=2000
      FDIV=3000
      FMUL=4000
      FPOW=5000
      FPUT=6000
      FNORM=7000
      FEXT=Ø
      FCDF=1
```

The order of the arithmetic instructions reflects the priority of arithmetic operands.

Other subroutines of use in arithmetic operations are:

```
      JMS I INTEGER
```

which truncates the floating accumulator (FLAC) to a 23 bit signed integer and returns the low order 12 bits in the AC. The converse of INTEGER is:

```
      JMS I (XFIX
```

which sets the FLAC to the 12 bit integer contained in the AC. Note that XFIX is not a signed integer setup of the FLAC but rather allows a range of $0 \leq FLAC \leq 4096$. And finally a routine to negate the FLAC is

```
      JMS I MINSKI.
```

The above three routines may only be called from field P, however from FIELD T the pseudo-instruction FTINTG performs exactly like JMS I INTEGER and FTMINSKI is the equivalent of MINSKI. The FPP may be entered from field T via

```
                    FINT1=JMS I [FPNTX.
```

Stack Operations

As has been previously mentioned any recursive language must have a stack or push-do n-list (PDL). Since 8-family machines do not have the hardware facilities for stack operations these are accomplished with subroutines. It is important to remember that since stack operations are not accomplished with a single instruction they may not be used in the interrupt service routine. The physical location of the PDL in FOCL/F is shown schematically in figure 1. Anyone familiar with other versions of FOCAL (with the exception of FOCL/S) will note that the stack builds upwards in core from the Text area rather than downwards from top of core. The motivation for this change is so that the command line may be appended to text, resulting in long command line capabilities, and so that variables are not deleted when text is modified. Text, variables, and the PDL reside in the same field so that there may be trade-off between these areas for more efficient core utilization.

The PDL routines available are:

```
    PUSHA       /Pushes the accumulator onto the PDL

    POPA        /Pops the top element of the PDL into
                /the accumulator

    PUSHF       /PUSHES three successive words starting
       ADDRESS  /at ADDRESS (usually floating data)
                /onto the stack

    POPF        /The reverse of PUSHF
       ADDRESS

    PUSHJ       /Subroutine call with return address
       ADDRESS  /put on the stack. ADDRESS is the
                /subroutine address

    POPJ        /Subroutine return via top of stack
```

The autoinde register used as the stack pointer is PDLXR.

It should be noted that POPA is effectively a "TAD I (Top of Stack", hence the top of the stack is added to the current contents of the AC. Also PUSHA and PUSHF operations return with the AC=0. Another interesting fact to keep in mind (or be warned against) is that the first instruction in the POPJ subroutine is a POPA. Hence, if the AC is non-zero when doing a POPJ, the return will be to the contents of the top of the stack plus the accumulator. This may be effectively used for multiple returns from subroutines.

Argument Evaluation

The routine which evaluates arithmetic expressions is by necessity recursive in nature. Unlike compilers there is no line scanning to convert the expression into Polish Notation. Instead the push-down-stack is used to defer some operations until the proper priority order is established, while operations already in the correct order are immediately executed. Thus what is on the PDL at any time is in Polish Notation but does not necessarily contain the complete expression at any one time. Simplified flow charts of the EVALuation routine are contained in the appendix. The calling sequence is:

```
        PUSHJ    /recursive subroutine call
           EVAL  /Address of routine

        Return
```

Upon return from EVAL the floating accumulator (FLAC) contains the numerical value of the expression, the terminating character is in CHAR, and SORTCN (see SORTC in "List Processing") is set to correspond to the terminating character. For example after evaluating ARG1 in FX(3,ARG1,ARG2) a "," will be in CHAR and SORTCN=14 (from the table "TERMS"), while after ARG2 a ")" will be in CHAR and SORTCN=11.

When evaluating additional arguments in functions, as in FX(1,ARG1,ARG2), all temporary data must be stored in the PDL if the function is to be recursive. That is to say FX(1,ARG1,FX(1,ARG3)) will only operate successfully if the function is coded recursively.

The location preceeding EVAL happens to be the get a character routine, GETC. Hence, to move past an argument separator, e.g. a comma, one may

```
        PUSHJ
           EVAL-1.
```

Other routines which evaluate arguments by calling eval are:

```
        PUSHJ
         NXTARG
        Return 1     / No next arg
        Return 2     / Next arg evaluated.
```

This routine is for evaluating successive arguments separated by comma's. Upon entering NXTARG, CHAR is tested for a comma. If not present the first return is taken, otherwise the routine PUSHJs to EVAL-1 and takes the second return on completion. The routine

```
        PUSHJ
           ARG
        Return 1
        Return 2
```

operates similar to NXTARG except that the FLAC is truncated to an integer before returning. Both returns have the AC=).

For those wishing to code functions in FIELD T, a

```
        JMS I FTNXTARG
```

will perform similar to NXTARG with the following notable exceptions. Only one return is taken and that is when there is another argument, if not FOCL/F dumps to the error routine. And FTNXTARG is not recursive. The non-recursively arrises from the fact that there are no PDL subroutines in FIELD T. Hence, users can only write non-recursive functions in FIELD T unless they want to also code stack subroutines. This is not a serious limitation since most special application functions are seldom used recursively.

Sorting

An interpreter works directly on source code: interpreting it, and executing the appropriate routines as it goes. The sorting comparison of characters is the key to the interpreting process. FOCAL derives its efficiency from the direct way in which it interprets code. Being JOSS-like, FOCAL needs to do no line scanning, but rather examines characters in a "left to right" fashion. This is made possible by the presence of a unique word for each command. In addition, the requirement that only the first characters in the command word specify the operation, greatly reduces the number of character comparisons which need to be made during execution.

FOCAL is list oriented with all comparisons being between the current character, CHAR (or in some cases the accumulator), and a list. The character lists are all structured with sequential unpacked characters (to speed up the sorting process) and are ended with a negative number (bit Ø set). The use of such lists in a page oriented machines like the PDP-8 family affords high core efficiency since such lists may be situated between "pages". The subroutines for sorting are:

```
SORTC
LIST-1
Return 1
Return 2
```

which compares the contents of CHAR to the list of elements starting at LIST and ending with a negative number. The subroutine return is to Return 1 if CHAR is contained in the list, and to Return 2 if not. If CHAR is in the list the contents of SORTCN contains a number which when added to the address of the top of the list, LIST, give the address of the word that compared, i.e. SORTCN is the relative address of the positive comparison.

```
TESTN
RET1    /Period
RET2    /Other
RET3    /Number
```

This is a variation of SORTC for comparing CHAR against a series of lists. There are three returns. The first is taken if CHAR is a period ".", RET2 is taken if CHAR is not a period or a number, and RET3 if CHAR is a number. In addition if RET3 is taken then SORTCN is the binary value of that number. The AC must be Ø when calling the above functions as well as

```
TESTC
   RET1    /Terminator
   RET2    /Number
   RET3    /"F"
   RET4    /Other
```

This routine takes four returns, similar to TESTN, except different lists are used for comparison. For the first return SORTCN is set according to the list TERMS (see appendix _). The second return is as in TESTN and the others are self explainatory.

Finally, there is a sorting and branching subroutine which is used for most of the command interpretation. This routine, SORTJ, has two lists as arguments. The first list is the list of comparison words, and the second list is a list of addresses of where control is to be transferred (via a JMP) if there was a positive match. This routine

```
SORTJ
    LIST-1
    LISTGO-LIST
Return            /if not in list
```

may be entered with the search character or word in the accumulator. If the contents of the accumulator is zero the contents of CHAR is used for comparison to LIST.

By the use of these routines complex branching and flow may be accomplished.

Addition of Functions

The addition of functions to the interpreter by writing machine language routines is the most common way of modifying the interpreter (and the easiest). For details about routines and some of the discussion continued in this section, the reader is referred to other sections.

Functions are detected (in EVAL) via the presence of an "F" beginning the function name. The single exception is the variable F' which is the only legal variable beginning with "F" in FOCL/F. Upon detection of the letter "F", characters are hash coded into a number until the terminating left parenthesis "(" is detected. The hash code is constructed by first clearing EFOP multiplying by two and adding the next full 8-bit ASCII character of the function name. Note that all combinations of characters may not be unique. This hash coded name is compared to the list of valid functions contained in FNTABL for a match via:

```
SORTJ
  FNTABL-1
  FNTABF-FNTABL
```

The destination addresses (start of the function) are contained in the table FNTABF. There are four unused locations currently available in these tables, they currently correspond to the function names FADC, FNEW, FCOM, and FN. Additional functions may be implemented by replacing a non-desired function. There are currently available three types of functions. They will be denoted by the words "normal", Field T", and "FX" functions.

Normal functions must always start in field P. That is, the address contained in FNTABF is a pointer to a Field P location. In general, if a user wishes to write a recursive function, then all coding concerning argument evaluation should reside in Field P and a normal function should be used. When a function of any of the three types is entered the first argument has been evaluated and is in FLAC. If additional arguments are to be evaluated, and a recursive function is desired, all temporary variables used in the function should be pushed onto the PDL via PUSHAs or PUSHFs before evaluating the next argument. There are three routines to evaluate arguments, as previously mentioned. They are EVAL, ARG, and NXTRAG. Note that they must be called via the stack by a PUSHJ. When the function is completed, the functional value should be left in FLAC and a JMP I EFUN3I executed. This function return checks for the right paren and normalizes FLAC. Several examples of normal functions are given in the appendix.

The Field T Functions reside in Field T. Since the first argument is evaluated before the function is entered any single argument functions are always recursive in nature. As a result of the inaccessibility of PDI routines in field T, multiple argument field T functions are not recursive. To establish a link to a field T functi^n the function referenced in FNTABL should be at FADC or below and the address in FNTABF. should be the same as the other field T functions, e.g. FCOS. A second list of actual destination addresses resides in field T in the table starting at FABLE-2, the appropriate address should be placed in this table. Upon completion of a function in field T the pseudo-instruction LEAVE will preform the appropriate return to EFUN3.

Subroutines available from field T are:

```
FTINTG=JMS FIXINT   /effective JMS I INTEGER
FINT1=JMS FPNTX     /FPP entry
FTERR               /Error call
LEAVE               /Function return
FPJMS= AJMS         /Cross field subroutine call.
```

The cross field subroutine call is a routine which does an effective JMS I (Accumulator. The call is:

```
TAD (Subroutine address)
CIF P
FPJMS
```

Note that subroutines with multiple returns may not be called via the FPJMS instruction. One special routine has been coded in field P to evaluate arguments and return the integer part of the argument (similar to the ARG subroutine), called by

```
TAD   (FRSTARG
CIF   P
FPJMS.
```

The third type of function is the FX functions. The single function name FX has for its first argument a number specifying which subfunction to call. These functions must begin their coding in field P as "normal" functions do. The FX functions are decoded via

```
FX,    JMS I INTEGER     /FX NUMBER
       SORTJ
         FXLIST-1
         FXGO-FXLIST
FXNO,  ERROR             /*FX function not available
```

Hence, to implement an FX function the appropriate address is entered in the list FXGO.

A final caution: the function return EFUN3 checks for a right parentheses by testing SORTCN which is set by EVAL at the end of argument evaluation. Thus, if the SORTC routine is used by the function, SORTCN must be saved and restored before the effective JMP I EFUN3I.

Space Requirements

FOCL/F has been developed to the point where it appears to have few free core locations for adding user coding, especially in field P. The intent, in development of FOCL/F, was to make the interpreter as powerful as possible, using all free space. For users wishing to add their own functions, it is expected that some of the routines will no longer be desired. For example, the FX routines will be unnecessary once specific functions are coded to control special hardware, the high-speed reader/punch routines may be replaced, or functions may be added in field T at some sacrifice to the text buffer. Those who wish to delete some of the implemented routines should take care to delete the entrys to those routines from FNTABF and COMGO. Some of the routines users are most likely to delete are

IGETC	-	high speed reader
IPUTC	-	high speed punch
LIT,LEN,etc.	-	device switching

the deletion of these routines frees core from about 14660-15377. The LIBRARY commands may be removed by putting ERROR5s corresponding to the LIBRARY commands in COMGO, or a 7777(8) in IOLIST. Deleting the FX fucntions by removing appropriate entries in FXGO frees the areas occupied by DECNX, DECOCT, OCTDEC, FXCT, FCOR, and FAND. This will give approximately another 200 locations. Since coding can be written so compactly using FOCL/F routines this is a usually adequate amount of storage.

Routines may, of course, be added in field T at the expense of some of the text buffer. To free core in field T, include the following in your coding

```
    FIELD FP    /FP=1
    *BUFR;ULINE1
    *ENDT;ULINE1
*LINE1
    /INSERT USER CODING HERE
     ....
    /END OF USER CODING
ULINE1,Ø;Ø
```

For those users who wish to add coding to FOCL/F with the PS/8 overlay, there is both good news and bad news. The bad news first: Much less space is available in field P. Those areas which are likely candidates are OCTDEC, FCOR, FXCT, FAND, and others which you must ferret out. Implementation of the PS/8 commands takes most of the space formerly used by the LIBRARY commands. The good news is that the field T functions, including the table FABLE are saved with programs in the "PROGRAM" commands. Hence, programs which need data acquisition routines may be called with those functions necessary for data acquisition, and programs needing extended functions for data reduction may be called with their functions. In this way only those functions necessary for execution need be in core. For further information on formatting of such coding refer to a listing of the overlay FOCFUN.

Page zero locations 172-175 in field P are available and numerous page zero locations in field T. See program listings for specifics.

Summary of Rules for Addition of Functions

I. Normal Functions

1. Select an area of core, if necessary by deleting undesired functions.

2. Enter function name in FNTABL and starting address of function in FNTABF.

3. Write function: function is entered with first argument evaluated. Additional arguments may be evaluated using calls:

```
PUSHJ;EVAL
PUSHJ;ARG
PUSHJ;NXTARG.
```

Care should be taken to PUSH temporary variables before evaluating arguments if a recursive function is desired. Care should also be taken to restore SORTCN and CHAR if changed during execution.

4. Return from function via a JMP I EFUN3I with function value in FLAC

II. FX Functions

1. Select an area of core, if necessary by deleting undesired functions.

2. Enter function address in FXGO corresponding to the desired FX number in FXLIST.

3. Write the function: function is entered with the comma following the FX function number in CHAR and the function number in FLAC. Proceed as in normal functions.

III. Field T. Functions

1. Enter FCOS in appropriate location in FNTABF and function name in FNTABL. The location in the table must be between FADC and FN. Put starting address in appropriate location in FABLE.

2. Free core if necessary in field T (see via discussion in text) or delete undesired function (e.g. FRAN,FX4, FX5, FX6).

3. Write function: function is entered with first argument evaluated in FLAC. If additional arguments must be evaluated the function will <u>not</u> be recursive and the arguments may be evaluated (as integers) by

```
TAD  (FRSTARG
CIF  P
FPJMS.
```

If non-integer arguments are required put a NOP(7000) in FRSTARG+4. The FPP may be entered via

```
CDF  T
CIF  P
FINT1
```

4. Exit is via the pseudo instruction LEAVE with function value in FLAC.

<u>The TEXT</u>

The text is stored in core as shown in figure 2. Characters are stored in 6-bit stripped ASCII, two characters per word. For characters which are not in the basic 64 character set, an octal 77 followed by the 6-bit code of that character is stored.

Viz. "A LINE OF TEXT would be stored as

```
01 40          /A(SPACE)
14 11          /L I
16 05          /N E
40 17          /(SP) 0
06 40          /F(SP)
24 05          /T E
30 24          /X T
77 15          /(carriage return)
```

Note that carriage return would conflict with the character "M" if not preceeded by a 77.

Lines of indirect program (numbered lines) are physically located in the text area in the order in which they were created. The order of the lines is maintained through use of "Threading" the lines. That is, sotred with each line is a pointer to the next sequential line, according to the line number, see figure 2.

In order to find a line, therefore, it is merely necessary to pick-up the address of LINEØ, which is always fixed in core (the comment line), and thread the list until the appropriate line is found.

Subroutines used for handling the text buffer are:

```
GETC      /puts next character in CHAR
PACKC     /packs CHAR into text buffer
GETLN     /forms a line number from characters
          /contained in the text into LINENO
FINCLN    /search for a LINENO
  RET1    /not found; set LASTLN,THISLN
  RET2    /found; and TEXTP

ENDLN     /inserts a line in text by fixing up
          /pointers
DELETE    /deletes duplicate line and siphors
TSTGRP    /SKIP IF AC=LINENO
```

In the above routines, where applicable, the location LINENO contains the sought for or assembled line number, LASTLN points to the lesser and/or last line compared to LINENO, THISLN points to the found or next larger line, and TEXP are pointers used by GETC to unpack characters. The three locations AXOUT, XCT, and GTEM are the TEXTPointers; AXOUT pointing to the next word to be fetched from the text buffer, XCT containing a zero or 7777(8) depending on whether the next character to be fetched is the left or right half respectively and GTEM contains the last <u>word</u> fetched from the text buffer. The equivalent locations used by PACKC are AXIN, XCTIN, and ADD.

Other page zero locations are intimately concerned with text. BUFR contains the address of the next free core location in which text may be added, i.e. BUFR points to the upper boundary of the text buffer used for storage of the indirect program. AXIN will of course point to the last location filled by the command line and PDLXR the top of the stack. CFRS contains the address of line zero and is thus the starting place for the search for lines. ENDT contains the start of the text buffer, not including LINEØ, and is used to reset BUFR when an ERASE TEXT or ERASE ALL is executed. BOTTOM contains the upper limit of the entire buffer including PDL and variables, and FIRSTV points to the start of the variable table. For a better feel for how these pointers define the text area see figure 1.

I/O and Interrupt Processing

With the exception of PS/8 devices I/O is controlled by the interrupt service routine. The purpose of handling I/O with the ION is so that input and output may proceed in parallel with calculations. Of course, there is a limit to the amount of buffering so that programs may get I/O bound. There is a single character TTY input buffer and a 16 character output buffer. The interpreter communicates with the interrupt processor through the subroutines

```
PRINTC=JMS I OUT
READC = JMS I INDEV.
```

In FOCL/F there are two output devices, one being the console terminal (TTY) and the "other" device. For the basic version the "other" device is the high speed punch, and for the PS/8 version the "other" device is specified by a device handler. The switching between these devices is accomplished by the LIBRARY commands which change OUTDEV (which is called by PRINTC). OUTDEV is the routine XOUTL for the console and OPUTC for the "other" device. The reader will note, from consulting the listing, that before any I/O is attempted to a PS/8 device, that a routine XWAIT is called. This routine waits for TTY interrupt routine to empty the TTY buffer and then turns the IOF. The Motivation for turning the interrupt off is that some PS/8-OS/8 devices do not have an interrupt enable. If the user has a system in which I/O devices are normally disconnected from the interrupt facility, they may overlap TTY I/O by replacing the WAIT subroutine call with a NOP. TTY output is buffered by a 16 character buffer starting at IOBUF. When XOUTL is called it checks to see if the teletype is in progress by examining the contents of TELSW in field T. If TELSW=Ø then the teletype is not in progress and the character is immediately typed, and TELSW is set to a non-zero number. If TELSW is non-zero then XOUTL examines the buffer to see if there is room to stash the character. If not it waits until there is room, otherwise the character is placed in the buffer. This buffer is a circular buffer with locations cleared as characters are typed. Hence, there is room in the buffer if the next location is zero. The two pointers OPTRO and OPTRI are output and input pointers to the buffer. They are circulated by updating pointers as follows:

```
TAD OPTRO     /PICK UP POINTER
IAC
AND 17        /16 locations in buffer
TAD OPTRØ     /Top of buffer:  ADD HIGH
              /ORDER
DCA OPTRO     /UPDATE POINTER
```

The reader will note that this causes the pointer to circulate between OPTRØ and OPTRØ+17(8) [the low order 4 bits of OPTRØ are zero]. When the interrupt service routine detects that the teletype is ready to print another character and the I/O buffer is empty, then it clears the flag and sets TELSW=Ø to indicate "not in progress".

There are two input devices in FOCL/F. One is always the consol TTY and the other is the high speed reader or a PS/8-OS/8 device handler. As READC calls the input routine via a JMS I INDEV, the device switching is accomplished by filling INDEV with XI33 for TTY input or IGETC for the "other" device. For the case of the high speed reader, IGETC buffers the input by a one page buffer. If the PS/8 overlay is being used IGEC calls XWAIT and then buffers the output into PS/8 blocks in field T.

As previously mentioned there is only a one character input buffer for the TTY (called INBUF). This location is cleared by XI33 each time a character is fetched by FOCL/F. If a character is input from the TTY and the location is non-zero then an exit is taken to the error routine. The calling location of this error is at KINT+4 in field T. This problem is usually encountered when inputting a program or data from a paper tape in the TTY reader. Fortunately, PDP8/e owners can clear the TTY reader flag without setting the reader run, hence an overlay is available which will correct this problem. For non-PDP8/e owners, this problem may not be overcome as it is in the hardware. Fortunately, by control of the echo device in the LIBRARY commands, this error exit may be circumvented for program input. Note that when inputting data via the ASK command the last data point should be followed by blank leader-trailer (or even turning the echo off will not correct the overflow condition).

For users who wish to add to the interrupt processor they may establish a link to their skip chain by using one of the user skip areas set up by FX(4,ARGS). These skip routines follow USKIP, or EXIT in the interrupt processor, (see the listing). There is a useful routine available which will set the user defined interrupt flag and exit from the interrupt service routine, if this facility is desired. This routine is called by a JMP I (UEXIT. The user defined interrupt is a powerful way of communicating between the interrupt processor and the user program.

The user defined interrupt is activated by the presence of a non-zero number in core location USRINT-2 in field P. This location is checked whenever a carriage return is encountered in the normal text execution. If non-zero the current program is interrupted and the line or group specified by the LIBRARY BREAK command is executed. Note that this routine is not currently nestable.

Program Control

When in the command input mode (START through IGNOR) characters are input from the input device until the receipt of a carriage return. If first input character is a neumeric, the line is added to the indirect program text. If not a neumeric then the command line is executed immediately (INPUTX). All information concerning program flow is contained in the PDL, the TEXTPointers, PC, and NAGSW.

The TEXTPointers (AXOUT, XCT, and GTEM) contain the information necessary to "remember" where the program is executing. NAGSW is a switch which determines whether a single line, a group, or all is being executed:

NAGSW=4000(8)	for single line
0001	for group
0000	for all.

The PC is FOCAL's "program counter" and contains a pointer to the beginning of the line which is being currently executed. Note that when executing sequential lines, the next line can be found by replacing PC with the contents of the core location PC points to, viz.

C(PC) replaced by TAD I PC

(see indirect text format - FIGURE 2). When executing the command line, in FOCL/F, PC=77.

At the end of each complete command control is transferred to PROC (usually via an effective JMP, the exception is the FOR command which transfers control via a PUSHJ). PROC, the primary control and transfer coding, tests and of line by the presence of a carriage return in CHAR. If a carriage return is present then a POPJ is executed, if not then the next command word is decoded and the appropriate routine called. PROC, when deciding the command word, ignores characters out to a terminator before branching.

If text execution is at the end of a text line then pushing NAGSW and PC will "remember" where to restart text exectuion. If at the end of a command the TEXTP, and CHAR must also be pushed. If one wished to execute a line or group from within a function or when making up special commands, the DO routine may be used as it pushes the appropriate pointers. See the coding of user defined functions (FNUM) or user defined interrupts (USRINT) in the listing for examples.

Variables

FOCL/F has two types of variables, "normal" variables and arryed variables. It is important to note that subscripted variables are not necessarily arrayed variables.

A "normal" variable is stored in core according to figure 1, with 5 words allocated per variable. The first word contains the first two letters of the variable name, left justified. The second word contains the subscript, if no subscript is specified then the subscript that is stored is zero. Therefore, in FOCAL and FOCL/F, all variables are subscripted variables. One will note, in verification of this fact, that if A(50) is defined there is not necessarily an A(1), A(2), etc. In FOCAL the subscript is merely an extention of the name. The "normal" variable format is shown in figure 1.

The routine which looks up variables is GETARG or GETVAR. The difference between these two entries is that GETARG tests the leading character to make sure it is not a neumatic. Both entries to the routine must be called via a PUSHJ. This routine packs the variable name (into ADD) and evaluates the subscript if any. Then a special routine, GS1P, is called to see if this variable name is reserved as a special variable, (discussed in later paragraphs). If not, then the low order twelve bits of the subscript is stored, in XSUBS, and the variable search is begun. The variable search is begun, starting at FIRSTV and ending at either PRIMEV (if the variable is a primed variable) or BOTTOM (if a non-primed variable). If the variable is found the pointer PT1R is set to the beginning of the data (past the subscript) and the value of the variable is copied into DATA in field P. The pointer PT1 is left pointing to DATA. If the variable is not in the list, then the variable is created and initialized to zero by adding to the beginning of the variable list (at FIRSTV-5), and FIRSTV is reset to the beginning of the list. Finally PT1 and PT1R are set as indicated above. Exit is via a POPJ.

In the case that the variable name was reserved for a special variable, an alternate scheme for finding the variable is taken. Currently, these reserved variables are true arrayed variables which are set up by the VARIABLE commands. These arrayed variables are looked up using subscript computation and are stored as three words per variable rather than five. The routine used to do this variable lookup is ARRVAR and associated routines.

Miscelaneous Routines

There are several utility routines which sould be mentioned as they may be found useful.

SPNOR:	ignores spaces and ALTMODES;
RTL6:	rotate accumulator left 6 places
ERROR:	exit to error routine, print error message, and return to command mode.
ENDCOM:	ends a command by ignoring characters to next ; or carriage return.
IGNORE:	ignores characters to the end of an argument. i.e. searches for ",".
PRNTLN:	prints LINENO as a line number

Debugging

FOCL/F has been coded so that a modified version of XOD (DECUS 8-89) may be used for debugging. This version of XOD must be assembled in field T (T=Ø) with break locations 2, 3, 4. This version is available from the author. The following changes must be made to the interpreter to avoid overwriting XOD with FOCL/F variables

```
10031/7574  6400
10034/7574  6400
10070/7574  6400
```

This moves BOTTOM so that variables begin below XOD.

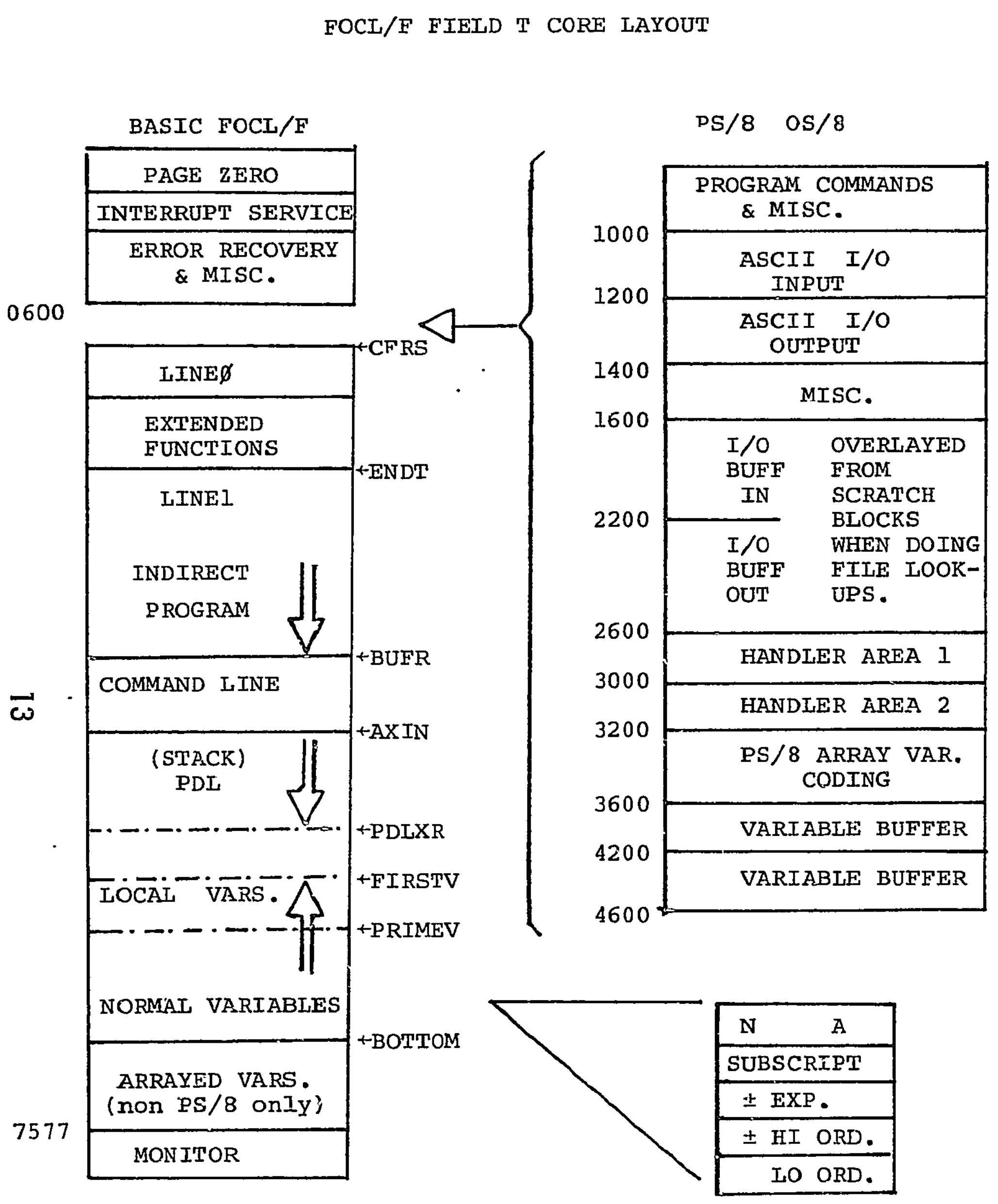


FIGURE 1. FOCL/F DYNAMIC STORAGE.

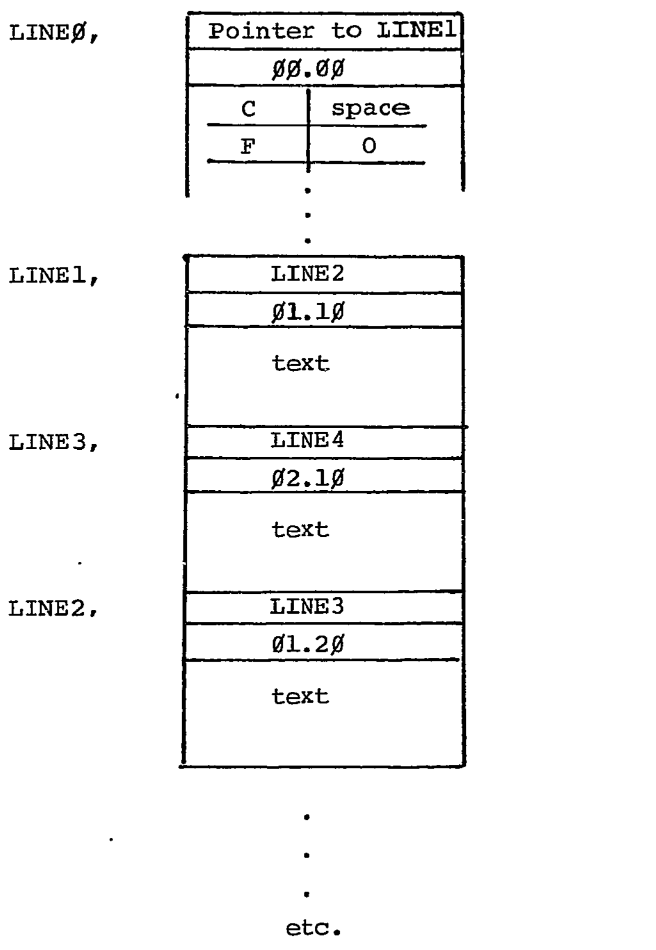


Figure 2 . Indirect Program Text Format.

SUBROUTINES AND INTERNAL ROUTINES
NEW INSTRUCTIONS

ROUTINE	CALLING SEQUENCE	EXPLAINATION
PUSHA	PUSHA	Put contents of accumulator on top of PDL.
POPA	POPA	Add top of PDL to accumulator.
PUSHF	PUSHF ADDRESS	Push three words (or four) of data onto PDL, starting at ADDRESS.
POPF	POPF ADDRESS	Put top three (or four) words of PDL into core starting at ADDRESS.
PUSHJ	PUSHJ SUBADD	Calls subroutine SUBADD recursively, putting return address on top of PDL
POPJ	POPJ	Return from a recursive subroutine, by returning to address on top of PDL.
GETC	GETC	Get next character into CHAR.
PACKC	PACKC	Pack next character (CHAR) into text buffer.
READC	READC	Read a character from INDEV into CHAR.
SORTC	SORTC LIST-1 RETRN1 RETRN2	Sort CHAR against LIST: return to RETRN1 if in list, return to RETRN2 if not in list. SORTCN is set to relative location in LIST.
SORTJ	SORTJ LIST-1 LISTG-LIST NOT IN LIST	Sort CHAR or C(AC) against list: if in list branch according to addresses in LISTG. if not in list return to call+3.
PRINTC	PRINTC	Print CHAR or AC on OUTDEV.
TESTC	TESTC RETRN1 /Term. RETRN2 /Numb. RETRN3 /F RETRN4 /Letter	Multiple return on CHAR attribute. Also ignores spaces before testing next CHAR. SORTCN set.
TESTN	TESTN RETRN1 /"." RETRN2 /Other RETRN3 /Numb.	Multiple return on CHAR attribute. SORTCN is set if number.
PRNTLN	PRNTLN	Print contents of LINENO
GETLN	GETLN	Unpack and form a LINENO, set NAGSW.
FINDLN	FINDLN	Search indirect text for LINENO. Set THISLN, LASTLN and TEXTP.
ENDLN	ENDLN	Insert line number pointers.
DELETE	DELETE	Remove old line number and siphon text.
TSTGRP	TSTGRP	Text contents of AC against LINENO. (SKP if =)
TSTLPR	TSTLPR	Skip if left-paren.
RTL6	RTL6	Rotate left 6 places.
SPNOR	SPNOR	Move past spaces and ALTMODEs.
ERROR	ERROR	Terminate execution and print error message.
EVAL	PUSHJ EVAL	Evaluate an arithmetic expression. Leave value in FLAC.
NXTARG	PUSHJ NXTARG	Move past a "," and evaluate an argument, return to call+2 if no "," found, call+3 otherwise.
ARG	PUSHJ ARG	Similar to NXTARG except truncate FLAC to integer.
INTEGER	JMS I INTEGER	Truncate FLAC to integer and return low order 12 bits in AC.
FPNTX	JMS I (FPNTX	Enter floating point package. Data field must be set to calling field. Floating Data Field = field of call.
FPNT	JMS I (FPNT	As above except Floating Data Field = field P.
XFIX	JMS I (XFIX	Set FLAC to integer value of AC.
SETFLAC	JMS I (SETFLAC	Set FLAC to integer value of AC and do function return.
EFUN3	JMP I EFUN3I	Function return.

EXAMPLE OF A RECURSIVE FUNCTION.

FOCL/F'S USER DEFINED FUNCTIONS.

```
                /COMMON ENTRY F0-F9 FOCAL DEFINED
                /FUNCTIONS - LINE OR GROUP NUMBERS
                /ENTERED IN TABLE "LNTABL" BY LET COMMAND.

14266  1070  FNUM,   TAD PRIMEV       /SAVE START OF VARIABLES
14267  4533          PUSHA            /TO MAKE PRIMED VARS LOCAL.
14270  1012          TAD 12           /POINTER TO FNTABLEIS
                                      /AUTOXR 12 - LEFT SET BY S(
14271  1374          TAD (LNTABL-F0-1
14272  3012          DCA 12           /POINTS TO LNTABLE ENTRY
14273  1412          TAD I 12         /GET LINE OR GROUP NUMBER
14274  7450          SNA              /SKIP IF NOT A ZERO ENTRY
14275  4557          ERROR            /*GROUP 0 NOT ALLOWED IN U:
                                         FUNCTION.
14276  4533          PUSHA            /STASH LINE OR GROUP NUMBE
14277  7410          SKP              /FIRST ARG IS ALREADY EVAL
14300  4566  FNUML1, POPA             /PICK UP NUMBER ARGS SO FA
14301  7001          IAC              /INC AS ONE MORE COMMING
14302  3043          DCA FLAC-1       /TEMP STORAGE
14303  4534          PUSHF            /PUT NEXT ARGUMENT ON PDL
14304  0044           FLAC
14305  1043          TAD FLAC-1       /AND STORE NUMBER OF ARGS.
14306  4533          PUSHA            /ON STACK TO MAKE RECURSIVE
14307  4531          PUSHJ            /EVALUATE NEXT ARGUMENT IF
14310  2164           NXTARG
14311  7410          SKP              /NO NEXTARG=DONE!
14312  5300          JMP FNUML1       /GET MORE ARGUMENTS

                  /NOTE THAT PRIMEV WAS NOT MOVED UP SO THAT
                  /IN CASE THE ARGUMENTS CONTAINED PRIMED
                  /VARIABLES THEY WILL BE LOCAL TO THE
                  /CALLING ROUTINE AND NOT THIS FUNCTION.
                  /NOW THAT ALL ARGUMENTS HAVE BEEN EVALUATED
                  /WE MAY CHANGE PRIMEV TO FORCE ADDITION
                  /OF NEW VARIABLES.
                  /PRIMEV HAS ALREADY BEEN PUSHED INTO
                  /STACK FOR LATER RESTORATION.  NEW PRIMEV
                  /IS AT BEGINNING OF VARIABLES.

14313  1031          TAD FIRSTV       /SET UP FOR LOCAL VARIABLE
14314  3070          DCA PRIMEV       /LIMIT POINTER.
14315  4566          POPA             /GET # ARGUMENTS
14316  4553          RTL6             /WILL GIVE ...C',B',A' PACK
```

```
14317  1373  FNUML2, TAD (47          /THIS IS THE PRIME PART OF NAME
14320  3061          DCA ADD          /ADD CONTAINS VARIABLE NAME
14321  3772          DCA I (XSUBS     /CLEAR SUBSCRIPT
14322  4531          PUSHJ
14323  1513           GS2             /CREATE THE VARIABLE
14324  4535          POPF             /GET DATA
14325  0044           FLAC
14326  4771          JMS I (SPCL1     /STORE IT (THIS IS COPY ROUTINE)
14327  1061          TAD ADD          /UPDATE
14330  1370          TAD (-147        /VARIABLE NAME
14331  7440          SZA              /SKIP IF A' WAS LAST (=DONE!!)
14332  5317          JMP FNUML2       /LOOP TO CREATE MORE VARIABLES.
14333  4566          POPA             /GET LINENO
14334  3067          DCA LINENO       /STASH FOR GEXIT (A PART
                                         OF GETLN)
14335  4767          JMS I (GEXIT2    /SET UP NAGSW
14336  1066          TAD CHAR
14337  4533          PUSHA            /SAVE CHAR FOR L-PAREN TEST
14340  1054          TAD SORTCN
14341  4533          PUSHA            /ALSO SORTCN
14342  1073          TAD CCR          /CR TO SIGNIFY END OF LINE
14343  3066          DCA CHAR         /TO RETURN AFTER CALL
14344  4531          PUSHJ            /GO DO LINE OR GROUP
14345  0401           DO+1

14346  4566          POPA             /RESTORE CHAR AND SORTCN
14347  3054          DCA SORTCN       /FOR R-PAREN MATCH
14350  4566          POPA
14351  3066          DCA CHAR
14352  1366          TAD (3247        /VALUE OF FUNCTION IS "Z'"
14353  3061          DCA ADD          /SO PUT Z' IN ADD
14354  4531          PUSHJ            /FOR GETVAR LOOKUP (GS4 IS ENTRY
14355  1427           GS4+1           /FIND IT
14356  4407          FINT
14357  0430          FGET I PT1       /GET VALUE
14360  0000          FEXT
14361  1070          TAD PRIMEV       /ERASE LOCAL VARIABLES BY
14362  3031          DCA FIRSTV       /RESTORING FIRSTV.
14363  4566          POPA             /RESTOR LAST PRIMEV
14364  3070          DCA PRIMEV       /FOR PREVIOUS RECURSIVE
                                         FUNCTION
14365  5527          JMP I EFUN3I     /RETURN TO FOCLF. ALL DONE!!!
```

```
                    COMLST,             /COMMAND DECODING LIST

14400   0323        "S                  /SET
14401   1023        "E&77↑100+"S
14402   0306        "F                  /FOR
14403   2206        "O&77↑100+"F
14404   0311        "I                  /IF
14405   1111        "F&77↑100+"I
14406   0304        "D                  /DO
14407   2204        "O&77↑100+"D
14410   0307        "G                  /G0TO
14411   2207        "O&77↑100+"G
14412   0303        "C                  /COMMENT
14413   2203        "O&77↑100+"C
14414   0301        "A                  /ASK
14415   2601        "S&77↑100+"A
14416   0324        "T                  /TYPE
14417   3424        "Y&77↑100+"T
14420   0314        "L                  /LIBRARY
14421   1414        "I&77↑100+"L
14422   0305        "E                  /ERASE
14423   2505        "R&77↑100+"E
14424   0327        "W                  /WRITE
14425   2527        "R&77↑100+"W
14426   0315        "M                  /M0DIFY
14427   2215        "O&77↑100+"M
14430   0321        "Q                  /QUIT
14431   3021        "U&77↑100+"Q
14432   0322        "R                  /RETURN
14433   1022        "E&77↑100+"R
14434   0317        "O                  /ON - 3 WAY DO BRANCH
14435   2117        "N&77↑100+"O
14436   0326        "V                  /VARIABLE (ARRAYED)
14437   0426        "A&77↑100+"V
14440   2502        "R&77↑100+"B        /BRANCH-MUST BE SPELLED OUT
14441   1014        "E&77↑100+"L        /LET
14442   7777        -1                  /EXPANDIBLE
14443   7777        -1
14444   7777        -1
14445   7777        -1
14446   7777        -1
14447   7777        -1
14450   7777        -1                  /EXPANDABLE COMMANDS
                    /THIS LIST IS ENDID BY "JMS I (IGNORE"
                    /COMMAND ROUTINE ADDRESSES

11155   1032   COMGO,SET
11156   1032         SET
11157   1032         FOR
11160   1032         FOR
11161   7453         XIF
11162   7453         XIF
11163   0400         DO
11164   0400         DO
11165   0573         GOTO
11166   0573         GOTO
11167   0606         COMMENT
11170   0606         COMMENT
11171   1226         ASK
11172   1226         ASK
11173   1227         TYPE
11174   1227         TYPE
11175   3247         LIBRARY
11176   3247         LIBRARY
11177   2204         ERASE
11200   2204         ERASE
11201   5034         XWRITE
11202   5034         XWRITE
11203   1303         MODIFY
11204   1303         MODIFY
11205   0177         START   /RETURN TO COMMAND MODE VIA 'QUIT'
11206   0177         START
11207   3350         XRETRN
11210   3350         XRETRN
11211   7452         ON      /ON - 3 WAY DO BRANCH
11212   7452         ON
11213   3236         VARIABLE
11214   3236         VARIABLE
11215   3635         XBRANCH /NOT ABREVIATED
11216   3247         LIBRAR  /LET FOR USER DEFINED FUNCTIONS
11217   7527         ERROR5  /*COMMAND NOT IMPLEMENTED. EXPANDABLE COMMANDS
11220   7527         ERROR5  /*COMMAND NOT IMPLEMENTED.
11221   7527         ERROR5  /*COMMAND NOT IMPLEMENTED.
11222   7527         ERROR5  /*COMMAND NOT IMPLEMENTED.
11223   7527         ERROR5  /*COMMAND NOT IMPLEMENTED.
11224   7527         ERROR5  /*COMMAND NOT IMPLEMENTED.
11225   7527         ERROR5  /*COMMAND NOT IMPLEMENTED.
```

```
                /LIST OF CODED FUNCTION NAMES

        4000    FNTABL=.
14000   2533    2533    /ABS
14001   2650    2650    /SGN
14002   2636    2636    /ITR
14003   2565    2565    /DIS
14004   2517    2517    /ADC
14005   2630    2630    /RAN
14006   2572    2572    /ATN
14007   2624    2624    /EXP
14010   2625    2625    /LOG
14011   2654    2654    /SIN
14012   2575    2575    /COS
14013   2702    2702    /SQT
14014   2631    2631    /NEW
14015   2567    2567    /COM
14016   1140    1140    /FIN
14017   2672    2672    /FOUT
14020   0316    0316    /FN
14021   0330    0330    /X
14022   1056    TAD EFOP        /F (NULL)
        4023    F0=.    /USER DEFINED FUNCTIONS
14023   0260    260     /0
14024   0261    261     /1
14025   0262    262     /2
14026   0263    263     /3
14027   0264    264     /4
14030   0265    265     /5
14031   0266    266     /6
14032   0267    267     /7
14033   0270    270     /8
14034   0271    271     /9

                /LIST ENDED BY JMS OUTCRLF
/NOTE THAT NAMES MAY BE HASHED VIA
/"A↑2+B↑2+S↑2" WILL GENERATE SAME CODE
/AS ABOVE FOR "ABS" 2533.
/NOTE:  ALL FUNCTION NAMES ARE NOT UNIQUE!!!
```

```
                /LIST OF FUNCTION ADDRESSES

          4200  FNTABF=.
    14200 2006  XABS      /ABS
    14201 2002  XSGN      /SIGN PART
    14202 3622  XINT      /INTEGER PART
    14203 7512  XDYS      /DISPLAY AND INTENSIFY
    14204 7527  ERROR5    /*FADC FUNCTION NOT AVAILABLE.
    14205 7434  XRAN      /RANDOM NUMBER
    14206 7434  ARTN      /ARC TANGENT
    14207 7434  FEXP      /EXPONENTIAL
    14210 7434  FLOG      /LOG
    14211 7434  FSIN      /TRIG FUNCTIONS
    14212 7434  FCOS
    14213 7434  XQRT      /SQUARE ROOT
    14214 7527  ERROR5    /NEW/*FNEW FUNCTION NOT IMPLEMENTED.
    14215 7527  ERROR5    /COM /*FCOM FUNCTION NOT IMPLEMENTED.
    14216 5040  FIN       /FIN
    14217 5042  FOUT      /FOUT
    14220 7527  ERROR5    /FN /*FN FUNCTION NOT IMPLEMENTED.
    14221 3535  FX        /X
    14222 3126  FNULL     /F(NULL)
    14223 4265  FNUM      /F0-F9
    14224 4265  FNUM
    14225 4265  FNUM
    14226 4265  FNUM
    14227 4265  FNUM
    14230 4265  FNUM
    14231 4265  FNUM
    14232 4265  FNUM
    14233 4265  FNUM
    14234 4265  FNUM



       1760   TERMS=.     /TERMINATOR TABLE FOR 'EVAL' AND 'GETVAR'
 11760 0240               240     /SPACE  0
 11761 0253               253     /+      1
 11762 0255               255     /-      2
 11763 0257               257     //      3
 11764 0252               252     /*      4
 11765 0336               336     /UP ARR 5
 11766 0250               250     /(      6 L-PARS
 11767 0333               333     /[      7
 11770 0274               274     /<      10
 11771 0251               251     /)      11 R-PARS
 11772 0335               335     /]      12
 11773 0276               276     />      13
 11774 0254               254     /,      14
 11775 0273               273     /;      15
 11776 0375               375;376 /ALTMOD 17 BOTH FLAVORS
 11777 0376
 12000 0215               215     /C.R.   17
 12001 0275               275     /= TO END GETARG FROM 'SET'
```

/EXAMPLE OF A FIELD T FUNCTION

```
                      /XRAN1. A FIELD T RANDOM NUMBER GENERATOR FOR FOCL/F
                      /CALLED BY FRAN(); GIVES 23 BIT NUMBER IN RANGE 0-1
                      /FROM (2↑17+3)*X(N-1), MODULO 36;
01600   1270   XRAN1,     TAD LSPR
01601   7104              CLL RAL
01602   0224              AND CONST2
01603   3271              DCA TEMP1
01604   1267              TAD MSPR
01605   0177              AND [177
01606   1271              TAD TEMP1
01607   7006              RTL
01610   7006              RTL
01611   7004              RAL
01612   3271              DCA TEMP1
01613   1270              TAD LSPR
01614   0177              AND [177
01615   7006              RTL
01616   7006              RTL
01617   7004              RAL
01620   1267              TAD MSPR
01621   3272              DCA TEMP2
01622   7430              SZL
01623   2271              ISZ TEMP1
01624   7400    CONST2,   7400


01625   1270              TAD LSPR
01626   7104              CLL RAL
01627   7430              SZL
01630   2272              ISZ TEMP2
01631   7410              SKP
01632   2271              ISZ TEMP1
01633   7000              NOP
01634   7100              CLL
01635   1270              TAD LSPR
01636   3270              DCA LSPR
01637   1267              TAD MSPR
01640   7004              RAL
01641   7430              SZL
01642   2271              ISZ TEMP1
01643   7000              NOP
01644   7100              CLL
01645   1272              TAD TEMP2
01646   3267              DCA MSPR
01647   1266              TAD MAXSPR
01650   7004              RAL
01651   1266              TAD MAXSPR
01652   1271              TAD TEMP1
01653   3266              DCA MAXSPR
01654   1266              TAD MAXSPR
01655   7110              CLL RAR
01656   6213              CDF CIF P
01657   3433              DCA I FTFLAC+1
01660   1267              TAD MSPR
01661   7010              RAR

01662   3434                      DCA I FTFLAC+2
01663   3432                      DCA I FTFLAC
01664   5665                      JMP I .+1
01665   2011                       EFUN3
                         /
01666   0000     MAXSPR,  0
01667   0000     MSPR,    0
01670   0001     LSPR,    1
01671   0000     TEMP1,   0
01672   0000     TEMP2,   0
```

```
                /FX FUNCTION DECODER

13535  4453  FX,     JMS I INTEGER
13536  454Ø         SORTJ
13537  3463          FXLIST-1
1354Ø  Ø114          FXGO-FXLIST
13541  4557  FXNO,   ERROR                  /*FX FUNCTION NOT AVAILABLE

       3464  FXLIST=.
13464  ØØØ1          1
13465  ØØØ2          2
13466  ØØØ3          3
13467  ØØØ4          4
1347Ø  ØØØ5          5
13471  ØØØ6          6
13472  ØØØ7          7
13473  ØØ1Ø          1Ø
13474  ØØ11          11
13475  ØØ12          12


       36ØØ  FXGO=.
136ØØ  3671          FCOR    /FX(1,
136Ø1  3646          FXCT    /FX(2,
136Ø2  35Ø4          FAND    /FX(3,
136Ø3  4646          FX4     /FX(4,
136Ø4  4651          FX5     /FX(5,
136Ø5  4654          FX6     /FX(6,
136Ø6  3541          FXNO    /FX(7,
136Ø7  3454          OCTANX  /FX(8,
1361Ø  3541          FXNO    /FX(9,
13611  3453          DECNX   /FX(1Ø,


                /FIELD T FUNCTION TABLE

Ø2ØØØ  4427          FTERR   /*UNDEFINED FIELD T FUNCTION
       2ØØ2  FABLE=.+1
Ø2ØØ1  16ØØ          XRAN1
Ø2ØØ2  15Ø3          FTATN
Ø2ØØ3  Ø767          FTEXP
Ø2ØØ4  1Ø55          FTLOG
Ø2ØØ5  12Ø6          FTSIN
Ø2ØØ6  12ØØ          FTCOS
Ø2ØØ7  14ØØ          FTSRT
Ø2Ø1Ø  2ØØ1          FABLE-1
Ø2Ø11  2ØØ1          FABLE-1
Ø2Ø12  2ØØ1          FABLE-1
Ø2Ø13  2ØØ1          FABLE-1
```

```
                    /EXAMPLE OF ADDED COMMAND

                /CODING FOR 'BRANCH' COMMAND
                /         "BRANCH #" WILL BRANCH TO THE INDICATED LINE #
                /         IF THE LAST INSTRUCTION EXECUTED BY THE FX(2,...
                                                                   )
                /         FUNCTION DID NOT PRODUCE A SKIP.

13635  1245  XBRANC, TAD SKPFLG       /CHECK SKIP FLAG
13636  764Ø          SZA CLA          /DID LAST FX(2,...) CAUSE A SKIP?
13637  5777          JMP I (GOTO      /NO - TREAT BRANCH AS GOTO
1364Ø  4541          SORTC            /YES - IGNORE UP TO TERMINATOR
13641  1373           TLIST           /TLIST CONTAINS TERMINATORS
13642  5776          JMP I (PROC      /TERMINATOR FOUND=EXIT TO PROC
13643  4536          GETC             /MOVE PAST CHARACTER
13644  524Ø          JMP .-4          /AND TEST NEXT.
3645   ØØØØ  SKPFLG, Ø

                    /EXAMPLE OF FX FUNCTION.


13646  4531  FXCT,   PUSHJ   /GET FIRST ARG AFTER FX(1,-
13647  2164           NXTARG
1365Ø  4557          ERROR   /*NO ARGUMENTS IN FX(2,X,X).
13651  4453          JMS I INTEGER    /FIX INSTRUCTION
13652  4533          PUSHA            /SAVE FOR ARG CALL
13653  4531          PUSHJ            /EVALUATE 2ND ARGUMENT
13654  3476           ARG
13655  3Ø46          DCA FLAC+2       /NO 2ND ARGUMENT
13656  4566          POPA             /RECALL INSTRUCTION
13657  3262          DCA .+3          /PREPARE TO EXECUTE IT
1366Ø  3245          DCA SKPFLG       /CLEAR SKPFLG
13661  1Ø46          TAD FLAC+2       /FETCH AC
13662  ØØØØ          Ø                /EXECUTE INSTRUCTION
13663  2245          ISZ SKPFLG       /INDICATE NO SKIP
13664  3Ø46  XFXIT,  DCA FLAC+2       /SAVE AC VALUE
13665  3Ø45          DCA FLAC+1       /CLEAR HIGH ORDER PART
13666  1375          TAD (27
13667  3Ø44          DCA FLAC         /SET NORMALIZATION CONSTANT
1367Ø  5527          JMP I EFUN3I     /TERMINATE FUNCTION CALL



                    /FDIS(X,Y) DISPLAY FUNCTION

17512  4453  XDYS,   JMS I INTEGER        /12-BIT INTEGER PART
17513  4533          PUSHA                /SAVE TO MAKE RECURSIVE
17514  4531          PUSHJ                /EVALUATE Y ARGUMENT
17515  2164           NXTARG              /"," SHOULD BE NEXT
17516  4557          ERROR                /*NO Y VALUE IN FDIS(X,Y)
                                          .
17517  4453          JMS I INTEGER        /MAKE 12 BITS
1752Ø  6Ø63          6Ø63                 /LOAD Y (1Ø BITS)
17521  73ØØ          CLA CLL              /CLEAR AC TO LOAD X
17522  4566          POPA                 /RESTORE X VALUE
17523  6Ø57          6Ø57                 /LOAD X AND INTENSIFY
17524  73ØØ          CLA CLL
17525  5527          JMP I EFUN3I         /RETURN TO FOCLF
```

FOCL/F
******

COMMANDS

ASK [X,Y,"TEXT",:,!]	ACCEPT INPUT VALUES
BRANCH <L1>	GOTO L1 IF LAST FX(2) DID NOT SKIP
COMMENT	IGNORE REST OF LINE
DO [G1]	SUBROUTINE CALL
ERASE [VARIABLES]	DELETE NORMAL VARIABLES
ERASE <TEXT OR G1>	DELETE ALL TEXT OR LINES
ERASE ALL	ERASE BOTH TEXT AND VARIABLES
FOR <X=E1[,E2],E3>;	COMMAND REPEATED !(E3-E1)/E2!+1 TIMES
GOTO [L1]	START PROGRAM EXECUTION AT L1
IF (E1)[L1,L2,L3]	CONDITIONAL GOTO
LET <FØ=G1>	ASSIGN USER DEFINED FUNCTION
LIBRARY BREAK, <G1>	ASSIGN USER DEFINED INTERRUPT
LIBRARY ECHO,<DEVICE>	SWITCH ECHO DEVICE
LIBRARY INPUT,<DEVICE>	SWITCH INPUT DEVICE
LIBRARY OUTPUT,<DEVICE>	SWITCH OUTPUT DEVICE
MODIFY <L1>	EDIT LINE L1
MOVE <L1,L2>	EDIT LINE L1 INTO L2; KEEPING L1
ON (E1)[G1,G2,G3]	CONDITIONAL SUBROUTINE CALL
PROGRAM <--->	PS/8 FILE COMMANDS
QUIT	STOP PROGRAM EXECUTION
RETURN	TERMINATE DO SUBROUTINE
SET <X=E1>	ASSIGN VALUE TO VARIABLE
TYPE [E1,"TEXT",!,:,$,%]	OUTPUT
VARIABLE <--->	ARRAY VARIABLE SETUP
WRITE [G1]	LIST PROGRAM OR LINES

< > ENCLOSE REQUIRED TERMS. [ ] ENCLOSE OPTIONAL TERMS.
ONE OR TWO LETTER ABBREVIATIONS MAY BE USED FOR COMMANDS.
"X" REPRESENTS A VARIABLE.
E1,E2, AND E3 ARE ARITHMETIC EXPRESSIONS.
L1,L2, AND L3 ARE LINE NUMBERS.
G1,G2, AND G3 ARE LINE OR GROUP NUMBERS.
L1,L2,L3,G1,G2, AND G3 MAY BE REPLACED BY ARITHMETIC EXPRESSIONS
WHICH DO NOT BEGIN WITH A NUMBER OR AN "A".
FØ IS A USER DEFINED FUNCTION (FØ,F1,...,F9 ALLOWED).

ADVANCED COMMANDS
****

```
      NON-PS/8 VERSION                          PS/8

                     TELETYPE              TELETYPE
LIBRARY ECHO,        HIGHSPEED             FILE
                     NONE                  NONE

LIBRARY INPUT,       TELETYPE              TELETYPE
                     HIGHSPEED             FILE

                      TELETYPE             TELETYPE
LIBRARY OUTPUT,       HIGHSPEED            FILE
                      BOTH                 BOTH

LIBRARY INPUT, REWIND
                                           INPUT [DEV:]<NAME[.EX]>
                                           OUTPUT [DEV:]<NAME[.EX]>


VARIABLE LIMIT,<# VARIABLES>               VARIABLE MAKE,<# BLOCKS>,[DEV:]<NAME>
VARIABLE OPEN,<X>,<START>                  VARIABLE OPEN,<X>,[DEV:]<NAME>
VARIABLE CLOSE,<X>                         VARIABLE CLOSE,<X>
VARIABLES KILL                             VARIABLES KILL
```

PS/8 PROGRAM COMMANDS

```
PROGRAM CALL,[DEV:]<NAME>[;COMMAND STRING]  .. SUBROUTINE CALL
PROGRAM DELETE,[DEV:]<NAME>  ................... DELETES FILES
PROGRAM EXIT  ..................... TERMINATE "PROGRAM CALL"
PROGRAM GET,[DEV:]<NAME>  ...................... LOAD PROGRAM
PROGRAM RUN,[DEV:]<NAME>[;COMMANDS]  .......... PROGRAM CHAIN
```

FUNCTIONS

```
FSIN(A)          FITR(A)          FX(1,A1,A2) .... CORE EXAMINE
FCOS(A)          FSGN(A)          FX(2,A1,A2)........... EXECUTE
FATN(A)          FDIS(X,Y)        FX(3,A1,A2)...... LOGICAL AND
FEXP(A)          FRAN()           FX(4,N,SKP,CLR)...  INTERRUPT
FSQT(A)          FIN()            FX(5,N).....INTERRUPT EXAMINE
FABS(A)          FOUT()           FX(6,N).... USER BUFFER FETCH
F(ARGS)              ....         FX(8,A)..... DECIMAL TO OCTAL
F0(ARGS)         F5(ARGS) .       FX(10,A).... OCTAL TO DECIMAL
F1(ARGS)         F6(ARGS) .
F2(ARGS)         F7(ARGS)  ............USER DEFINED FUNCTIONS
F3(ARGS)         F8(ARGS) .
F4(ARGS)         F9(ARGS) .
                     ....
```

SYMBOLS AND CHARACTERS

↑ * / + -	EXPONENT, MULTIPLY, DIVIDE, ADD, SUBRACT
()[]<>	ENCLOSURES FOR ARITHMETIC EXPRESSIONS
!	OUTPUTS A CARRIAGE RETURN/LINE FEED - TYPE OR ASK
"	OUPUTS "STRING" - TYPE OR ASK
:	TABULATION CONTROL - TYPE OR ASK
$	"TYPE $" OUTPUTS SYMBOL TABLE.
%	FORMAT CONTROL - TYPE
?	TRACE SWITCH
;	COMMAND TERMINATOR
,	ARGUMENT SEPARATOR
ALTMODE	LINE CONTINUATION CHARACTER: STATUS OF SPACE
CTRL/Z	END-OF-FILE CHARACTER
CTRL/G	CHANGES SEARCH CHARACTER IN "MODIFY" OR "MOVE"
CTRL/L	SEARCH FOR NEXT OCCURRENCE OF SEARCH CHARACTER
LINE FEED	RETAIN REST OF LINE IN MODIFY
RETURN	ENDS COMMAND LINES

*