L A B - F O C A L (V5C)
15 Nov 1982

The following interesting new things have been included in LDF-V5C:

1) 'Group GOTOs' are now allowed. GOTO, IF, JUMP and LOAD & GO commands may now meaningfully specify a 'group number', rather than a 'line number' as the branch point. Thus 'GO 5' is now a legitimate command which means: 'goto the lowest-numbered line in Group 5'; previously it was necessary to actually have the line '5.00' in order to use such a command. Other 'goto' type commands work the same way, including the error return option in file commands (but not the branch specified by QUIT or RETURN).

This feature may also be used with 'partial group' (sub-group) specifications: 'G-1.5' will transfer to line 1.5 (if it exists) or else to the first line with the next higher number in Group . The exception to this are lines specified by the QUIT command as 'error restart points' (in this case the 'negative' sign indicates a 'deferred' branch, rather than a 'sub-group' operation) and branches specified by the RETURN command.

This feature is particularly convenient in conjunction with the 'Load and Go' (L G) command: 'L G NXTPRG-3' will load the next program, starting at the first line in Group 3 (whatever it may be . This allows Group 3 to be modified without changing the call. Group labels are also nice in long IF commands, replacing a confusing mass of specific line numbers.

The only known 'drawback' to this feature is that it causes the error codes for a 'missing GOTO line' and a 'missing DO routine' to be identical. This error is now '03.40 - LINE IS MISSING'. In case it is not clear from the above, a command such as 'GO 1.7' (which uses a -specific- line number) requires that the line indicated be present; otherwise an error will occur. Group calls are marked by 'whole' or 'negative' line-number specifications.

2) Other optimization led to a few additional error code changes. (Note the exact error code is only important for 'batch-mode' operation, or for cases in which the 'on-line' error-message feature is omitted. Errors are normally followed by a brief explanation, hence their 'number' is not particularly relevant.) 'ZERO SIZE TOO BIG' is now 21.15 (was 12.05), while 'TOO MANY DIGITS IN NUMBER' is now 12.65 (instead of 22.65) and 'UNMATCHED PARENTHESES' are now 09.45 (rather than 09.40). Please correct any error-message sheets.

3) The highest available memory field is now stored in location '101' by the initialization code. This may be checked with 'FPAL(,101,6201,1601)', ignoring values greater than 7 (this location has other uses too).

4) The size of the symbol table may be found with FPAL(,1030,7040,2204). This is not a change - simply a convenient function call which was omitted from the documentation. The symbol table size is a function of the amount of available memory as well as the run-time environment (BATCH, etc.).

5) The HESITATE command now works properly when programs are run under BATCH. To 'fine-tune' the delay time (or to make drastic changes, if you like), simply change the value in location 10115 This can be done quite conveniently (for testing purposes) using 'FPAL(N,3115)'. This value is only used when a 'real-time clock' is not available, i.e. when running in a 'non-interrupt' environment, or on machines without this feature. The longest delay possible is 40.9 seconds (versus 214.7 for the RTC version).

6) Overlays for the PDP-12 and LAB8/e, the Tektronix 4010, incremental plotters and the 'expansion kit' are now available (at last!). Since LAB-FOCAL requires at least 16K, there are no '8K' or '12K' versions of these routines, and while the new overlays are more-or-less compatible with the UWF versions, there are also some significant changes.

7) The HESITATE command in the lab overlays now uses seconds and tenth-seconds, rather than milli-seconds, and there is no provision for changing the time-base (except with FPAL calls). The FQUE function may be used to set up event-driven subroutine calls using interrupts from the clock - OR from the Schmitt triggers! A really nice change to the pattern table now shows control characters with a bar on top of the corresponding symbol in place of an undecipherable pattern of dots! Try the following program to see this big improvement:

```
1.1 TYPE #,:20; FOR I='@,'_; TYPE #I
1.2 TYPE !!:20; FOR I='@,'_; TYPE #I+192
```

The error codes VIEW BUFFER FULL (14.15), BAD SSW NUMBER (14.50) and BAD EXT. LEVEL (14.55) may now appear on your screen. Also, to switch to the terminal temporarily, use the 'O T' command, or 'O E,T' to make the terminal the 'default' device. 'O 5' returns to the 'scope. 'Rubout' now works no matter whether the system is set for 'scope' or 'no-scope'. For general programming set RO=FPAL(,1000) and use 'TYPE #RO' to erase things.

8) The Tektronix routines now use the variables 'X.' and 'Y.' to report the position of the cursor (rather than XJ and YJ which were used by UWF), and the cursor position is obtained with the 'FCUR' function, rather than the 'FJOY' function.

9) The plotting routines now use the variables 'R.' (for rotation), 'S.' (for size), 'D.' (for direction) and 'M.' (for scale multiplier), in place of the variables '$R', '$S', '$D' and '$F' used in previous versions. The '$' prefix conflicted with the 'local variable' feature in LDF.

10) Suggestions for implementing two-letter USER commands may be found in the source file. If you add routines using the expansion kit', up to six additional 'U' commands can be added simply by patching the 'U' table.

11) The code for 'VT52' support has recently been completed. This overlay adds the 'KEYPAD' and 'PLOT' commands to the '8//e' versions for use with a 'VT-52' style terminal. The 'KEYPAD' command allows any of the 19 keys in the auxilliary keypad to be assigned to a designated 'DO' call. Thus a 'KEY ENTER=1.1' command instructs LDF to execute line '1.1' whenever the user hits the 'ENTER' key. This may be done either while a program is running (which allows one to check on the progress of a long calculation just by hitting a key), or while LDF is waiting in 'command mode' (which allows one to easily create 'menu-driven' application programs). The 'PLOT' command allows data to be displayed on the screen in a very simple manner (including the use of the special 'graphics characters'). Resolution is 80H by 168V. This overlay will also work with the 'Z19' terminal, allowing the five function keys (plus all the keypad keys) to be assigned to 'DO' calls. Unfortunately the Z19 has a different set of graphics characters which are not as useful for plotting data...

A 'VT-100' overlay (possibly without PLOT support) is also under consideration. However, since the VT-100 may be put in 'VT-52' mode (which is somewhat easier to program), the necessity of a special overlay is not entirely clear. Plotting in ANSI mode on the VT-100 is most easily done by using the 'ESCAPE' operator (as described in the release notes) hence a special 'PLOT' command seems almost superfluous. Comments, anyone?

12) The FRAN function has been changed. The new version fits in half the space, but appears to give equally good results, although testing has been rather limited at this point. The previous version was based on a solidly-documented algorithm while the new version is a bit more empirical. Both the 'flatness' and pair-wise correlation have been roughly checked however, and no obvious problems were found. The justification for this change is that FRAN is seldom used for 'serious' calculations, hence memory requirements were deemed more important than statistical 'accuracy'. This is not to imply that the new version is less 'accurate' but only that it has not been as extensively tested as the previous version.

13) The 'trig' functions (FSIN, FCOS, FATN) have been revised to work with ANY system of angular measurement. The default, as before, is RADIANS, but one may now use DEGREES or GRADS or REVOLUTIONS or 'whatever' as one wishes. The choice is made by means of the new 'FRDG' function, whose argument is: 'the number of angular units in one quadrant'. Thus to change to 'degrees' one should 'SET FRDG(90)', and to change back to radians: 'SET FRDG(PI/2)'. The accuracy of the functions is fully maintained - in fact the accuracy of the sine and cosine routines may be even slightly higher than before since the radian/degree conversion is entirely avoided (Note: angles should be limited to less than 128 'revolutions' (46080 deg) when calling FSIN/FCOS.) To test this feature, try a 'TYPE FATN(1)' command after setting 'radians', 'degrees' or 'grads' with the FRDG function (args 'PI/2','90','100' resp.).

14) The 'FEVM' function has disappeared, being replaced by a special 'FQUE' call. This change and the FRAN modification were necessary to make room for the FRDG function described above. The new method for changing the 'event mask' is to call FQUE with a null argument, followed by the desired value of the bit mask. Thus to DISABLE task '1', use 'SET FQUE(,-2048-1)', while to disable both tasks '3' and '4', use 'SET FQUE(,-512-256-1)'. The value '-1' by itself will ENABLE all tasks.

LDF - VERSION 5D

This version (dated 21-MAR-83) contains the following improvements:

1) A new command - INPUT CLOSE - has been added. The 'I C' command provides two needed services: (a) it makes the input file inaccessible, thus preventing unwanted (or worse, unexpected) changes from being made by the FRA function, and (b) it reduces the 'handler use count' for that device, allowing handlers that are no longer in use to be removed to make more space for the 'stack'.

If an INPUT CLOSE command is given without any option, the CURRENT input file will be closed; if, on the other hand, a command of the form: 'I C/n' is given, file 'n' will be closed. Once an input file has been opened it remains accessible until it has been closed. Even when the program reaches the 'end-of-file', the file remains 'active' and can be 'restarted' without any system overhead. (Since the handler is still available, all the 'O I,R' command has to do is reset the 'file pointers' to their initial values). The OUTPUT file also remains active after it has been 'closed', allowing it to be immediately re-used for input by an 'O I/0,R' command.

Since the FRA function permits both READ and WRITE access to any active file, once a file has been opened it is vulnerable to accidental modification, especially since the FRA function can also be used to simply store temporary data in one of the file buffers. By using a command such as 'I C/3', it is now possible to select buffer/3 for temporary storage without any danger that the data will be written into a previous file. Another use for the new 'I C' command occurs in connection with removable media: whenever a disk or tape is changed, all files associated with that device should obviously be closed.

Once an input file is closed it cannot be restarted by an O I,R command; any attempts to use the file for input without re-opening it will result in either an immediate EOF or an error. An error also occurs if one attempts to close input file/0 while an OUTPUT FILE is still open. (Output files can only be terminated by an 'Output Close' or 'Output Abort' command in order not to confuse the operating system.) Once the output file has been closed, however, it becomes an active input file, and at that point the 'I C/0' command may be used to remove the handler.

Note that since handlers are SHARED whenever possible, closing ONE input file may not actually remove that handler if it is also being used for other purposes. A given handler may be used for up to FIVE different operations: the 4 input files (or one output file and 3 input files), and the most recent LIBRARY operation. The handler is removed only when the 'use count' goes to zero. Handlers are initially loaded so as to leave as much space as possible for the stack, but as different handlers are used by the program, it can happen that not all space available can be used. To leave as much space as possible for the stack, programs should load the handler which will be used for the longest period of time FIRST, followed by more transient handlers.

Handlers come in three sizes: those that require no space at all, those that use one page of memory, and those that use two pages. LDF has 4 pages of memory available for handlers, making it possible to interact with as many as 5 different devices all at once (four with 1-page handlers, plus one using a '0-page' handler). The size of the stack area, as measured by the number of nested subroutine calls possible, is shown below as a function of the number of pages used:

pages used	nested calls
0	50
1	40
2	31
3	21
4	11

Operations on the SYSTEM device do not use up any handler space. This is also true of any device which is 'co-resident' with the system handler, for instance, DTA1: in a TD8E system, or RKB0: in a RK05 system. Unfortunately the normal floppy-disk handler can only access Drive 0 - hence any references to RXA1: require the use of the (two-page) non-system handler. For users wishing to improve their system performance, a version of the RX02 handler is available that can access both drives. An improved LINC-tape system handler which supports both SYS: and LTA1: is also available.

2) The FRA function has been fixed so that non-integer indices can be used with modes 2 and 4. Previous versions of LDF returned the wrong value if the index was not an integer in those cases. This revision changes the two error codes associated with the FRA function and also improves the access speed by about 3%.

3) The QUIT command now resets the 'task level' register so that normal operation will occur when the program is restarted. In previous versions if a QUIT command was executed by an external subroutine call, all lower-priority tasks were 'locked out' until after an error occured.

4) The KEYPAD command now -enables- keypad calls as well as defining them. The idea behind this change is that whenever the program begins execution of a subroutine as a result of pressing one of the keypad (or function) keys, it automatically locks out further keypad calls until the subroutine is finished. This was done to eliminate confusion caused by having the second keypad call interrupt the first one. (Unlike FQUE calls, keypad calls all have the same priority.) Since there ARE times when it is desirable to be able to use the keypad within a routine that was itself initiated by a keypad call, one can now clear the 'lock-out' switch that would normally prevent this by putting a KEYPAD command in the subroutine. This command need not define any keys - a null 'K' command is sufficient.

5) All subroutine calls (i.e. DOs, LIB DOs, FSUBs) now perserve the current input/output devices as well as the current output format. This change was made to accomodate external DO calls which needed to perform I/O operations but had no way to find out and/or restore the I/O devices being used by the main program. With this change the following program segment will type two values in %5.02 format on the terminal, and one in %3) on the lineprinter:

```
08.50 O T; Type %5.02,A; Do "; Type C

09.10 O L; Type %3,B
```

Note that line 9.1 does not bother to restore either the previous output device or the previous format; these parameters are automatically restored by LDF at the end of the subroutine call. While this example involves only INTERNAL calls (so the program flow is predictable if group 9 were called by an EXTERNAL event the same result would occur, namely the terminal (or whatever output device was selected at the beginning of the call) would be restored at

the end. Note that the status of the various 'echoes' is NOT preserved, nor (unfortunately) are things like the file selection or the trig mode. It is thus necessary to explicitly save the status of such parameters (but only if the external call changes them) so that the original values can be restored. The following program segment illustrates this problem:

```
07.60 O T;O I/3;Type :FRA(-1,4),FRA(100)
07.70 Type FRA(101)

08.10 O L;O I/2;Type :FRA(-1,1),FRA(i=i+1,FADC())
```

Which file will the value typed out in line 7.7 come from? If the program executes normally it will come from file/3. However, suppose an external call executes line 8.1 just before the program gets to 7.7. Then the data will come from file/2 because line 8.1 alters both the file number and the FRA mode, and these two parameters are not automatically restored. The solution to problems like this is to store the necessary information in a variable so that the FRA mode or file selection can be explictly restored by the external call:

```
07.60 O T;O I/N=3;Type :FRA(-1,M=4),FRA(100)
07.70 Type FRA(101)

08.10 O L;O I/2;T :FRA(-1,1),FRA(i=i+1,FADC()),:FRA(-1,M);O I/N
```

Another 'state' which may need to be preserved is the 'trig mode': if the main program uses 'degrees' while the external subroutine wants 'radians', appropriate calls to the FRDG function will be required. Note that FRDG returns the argument, hence the setting can be saved with a call such as TM=FRDG(PI/2).

6) The code for the internal routine 'PRINTC' was condensed so it would all fit in one field. This should improve performance when LDF runs in the background under RTS/8 as it eliminates a frequent source of trapped instructions.

7) More 'user space' has been left in Field 0 for patches to implement other 'internal' handlers - consult the listing for details. Many page-0 references have changed, hence be sure to check all such overlays for compatibility.

8) Group numbers now work correctly in the error return ('=') option used by some of the 'L/I/O' commands. If a group number is specified, the branch will go to the lowest-numbered line in the group.

9) A 'communications package' has been developed for V5D. This overlay adds the 'I/O/U-H' commands for communicating with a remote 'Host' machine, such as a VAX or a PDP10. The host is assumed to use the 'XON/XOFF' protocol for both sending AND receiving. Versions of this overlay are available for the PDP12 and the VT/78, both of which have well-defined communication ports. Custom versions built around the function-expansion package have also been developed.

The 'U H' (USE HOST) command is one of the most interesting features of this overlay. Executing this command links the terminal to the Host, allowing the user to communicate directly with the remote machine in order to 'log in', set up file transfers, etc. Typing CTRL/E returns the terminal to FOCAL. The user can thus interact with both FOCAL and the Host concurrently, using the INPUT HOST and OUTPUT HOST commands to transfer files. All I/O is done via the interrupt system, at speeds up to 9600 baud. Above 2400 baud, however, it is wise to disable the scope refresh on the '12 since the M707 receiver only buffers a single bit after the flag goes up (rather than an entire character), and the interrupt latency during a DSC instruction can thus cause a few 'glitches'.

One frequent use of this overlay is to list programs on the host's line-printer. A typical set of commands for doing this is shown below:

```
.R LDF                      run LAB-FOCAL
L L program                 load the desired program
U H                         connect terminal to the host
Username: xxx               login...
Password:                   password: don't echo
(login messages appear)
$ COPY TT: LPT:             set up the transfer
(type CTRL/E)               return to FOCAL
*O H;W;T #26;O T            Output to Host, Write, Send EOF
*U H                        Go back to the Host
$ SHOW QUEUE ...            Check print queue
$ LOGOFF                    logout
(type CTRL/E)               back to FOCAL
*                           go pick up the listing...
```

As can be seen from this example, at one moment the user is 'talking' to LDF, while at the next he or she is talking to the Host. The I/O/U-H facility thus provides a very synergistic programming environment. Automated transfers are possible simply by doing a 'O H' and then TYPEing whatever command is necessary. For instance, to get a directory listing one might use the following:

```
O H;T "COPY TT: LPT:"!;L A,E;T #26;O T
```

Data can also be read from the Host by using the 'I H' command to switch the source of input. There is a problem in trying to copy data directly from the host to an OS/8 file, however, since OS/8 handler calls always disable the interrupt system which thus causes characters sent from the host to be 'lost'. The solution to this problem is to perform such transfers using the VAX: (or TEN: or RSX:) handlers. The 'I H' command is, however, useful for sending the results of a complex calculation back to a FOCAL program for plotting or control purposes.

In addition to the restriction on the use of the 'I H' command mentioned above, there are a few other limitations the user should be aware of: (1) all LF's are suppressed on output. This was done so that CR/LF's didn't generate unwanted blank lines. (2) No EOF code is sent, hence it is the user's responsibility to send a CTRL/Z (#26) to close the file. Conversely, one can merge the listings of several different programs into a single file just by omitting the 'T #26' command until after the last one. (It is convenient to separate each listing by a FF, which is generated with a 'T #' command.) (3) All characters except CTRL/E and CTRL/F (and null) are passed to the host by the 'U H' command; typing CTRL/C will thus not get you back to OS/8. CTRL/E is the normal 'exit' character which returns to the command following 'U H'. It is useful to type a prompt such as "Back in FOCAL" to advise the user of his or her status when 'U H' commands are built into a program. Occasionally (especially at high baud rates on machines with M707 or KL8E interfaces), output will apparently 'lock up' so that the only way to return to FOCAL is to type CTRL/F. This is due to missing (for whatever reason) the signal from the host that requests more output. A 'U H' followed by an immediate 'CTRL/E' will clear up this condition. (4) The keypad on a VT52 or Z19 may be used by 'keypad' editors on the host as well as by LDF programs. Since the editor disables the keypad when it exits, however, it is necessary to restore 'keypad application mode' when you get back to LDF. TYPE $=" will do this.

1) Minimum memory size is 12K (was 16K); .ST after .LDF works
2) ASK or TYPE :-N works correctly with all I/O devices
3) ASK does not send FFs found in a file to the terminal
4) FBLK works correctly when used in filename expressions
5) FTRM clears the terminator, as well as returning it
6) FTAB returns the current 'tab position'
7) FQ replaces FQUE
8) X^(-Y) is computed as 1/(X^Y), instead of (1/X)^Y
9) The error 'Too Many Digits in Number' is gone -- numbers can now be any length. Non-significant digits are used to adjust the decimal point, so '987654321001234567890' is exactly equivalent to the number: '9876543210E+10'.

- - - - - - - - - - - Notes - - - - - - - - - - -

1) 12K provides 121 variables, 12 block programs (16K allows 677 variables, 15 block programs); BATCH operation is possible in 12K. The .START command now works after CCLK calls.

2) I F,E;O F;A :-N failed in V5D due to a conflicting use of the same memory location.

3) ASK previously echoed all FFs to the terminal (thus clearing the screen) regardless of the source of input. Only FF from the terminal is now echoed (this is chiefly for PDP12 users).

4) FBLK used a memory location also used by the 'O 1' command.

5) FTRM previously just returned the last terminator from an ASK command. Since the terminator remained until the next ASK command, UNTIL loops looking for a special 'end' character could not be easily restarted. FTRM now clears the terminator after returning the current value.

6) FTAB replaces FPAL(,1576). To do a CR/LF only when there is output on the line, use 'IF(-FTAB())FOUT(0M)'. Since the same position counter is used for all output devices, FTAB (and/or the ':' operator) may not behave properly until after a CR has been sent to a device. (CR sets the counter to 0.)

7) FQUE does many things besides 'queing' tasks, hence FQ is a better (as well as a shorter!) name. (Requested by a user.)

8) Multiplication rounds off, division truncates; thus 1/(X^Y) is slightly more accurate than (1/X)^Y. Other changes to the '^' routine were also made which speed up this operation.

9) This change is long overdue. A recent need for reading double-precision output from a mainframe program prompted the effort. If there are digits to the right of the decimal point, the conversion is a bit slower than before; the binary result, however is more accurate since no approximate conversion constants are used. The largest decimal number that can be converted without any error is now 34225520639 (it was 34359738367).

The '@' character can now also be used in 'numeric' strings; it (and its lower-case counterpart: '`') are treated as '0', so '0.@@A' is the same as '0.001'. ('@' was previously treated as

a numeric terminator -- the documentation still implies this.)

&

COMMAND AND FUNCTION SUMMARY

```
@@ (*)  Ask      Break    Comment  Do       Erase    For      Goto     Hesitate  If
Jump    Key*     Modify   Next     On       Plot*    Quit     Return   Set       Type
Until   View*    Write    X(nop)   Yncrement(+-1)     Zero               * = optional

Input Buffer     Input Close      Input File       Input Host*      Input Terminal

Open Input       Open Output      Output Abort     Output Close     Output Dump
Output Buffer    Output Echo      Output File      Output Host*     Output Lineprnt
Output Plotter*  Output Scope     Output Terminal                   * = optional

List All         List Files       List Only        List Programs
Library Save     Library Erase    Library Load     Libr Comment     Libr Quit
Load In          Load ('n) Go     Load ('n) Do

Use Host*    (connects the terminal to a remote computer for interactive use)

FABS     FADC*    FAND     FATN     FBLK     FCOM     FCOS     FCUR*    FDAY     FDIN*
FEXP     FIN      FIND     FITR     FLEN     FLGS*    FLOG     FLS*     FMAX     FMIN
FMQ      FOUT     FPAL     FQUE     FRA      FRAC     FRAN     FRDG     FSGN     FSIN
FSQT     FSR      FRS*     FSS*     FTIM     FTRM     FVB*     FXL*     * = optional
```

&ATSIGN

@ [list of arithmetic expressions]

The @ command is used in some versions to open or close relays. The argument list specifies the relay number, using positive values to mean 'close' and negative values to mean 'open'. The @ commard is used in place of the KONTROL command in versions which also use the KEYPAD command. Additional information may be found in the KONTROL section.

@Examples:

@@ 1	Close relay #1
@@-1	Open relay #1
@@ 2,3,4	Close relays 2,3,4
@@ 1,-1	Create a short pulse using relay #1

&ASK

ASK [list of variable names, operators]

The ASK command reads ASCII data from the current input device. Decimal numbers are converted to binary and stored in the memory locations specified by the variable name. Prompting information may also be written to the current output device. Input is 'free format': integers, mixed decimal or power-of-ten ('E') notation may be used. Letters are converted to numbers (A=a=1, Z=z=26) and hitting the '=' key retains the current value of the variable. DELETE (RUBOUT) erases all previous digits and prints a '?' to indicate that the entire number must be re-entered. Use the ':-N' option to ignore 'N' characters when reading data from a file. Any non-numeric or non-alphabetic (A-Z,a-z) character terminates an input request, hence SPACES, COMMAS, CRs, Dollar signs, etc. may be used. The FTRM function returns the value of the last ASK terminator.

@Examples

```
ASK A,B,C       Gets values for three variables from current input device
ASK "Enter values for Time, Concentration " Time,Conc
                Prints prompt message, then gets values for Time and Conc
Input File; ASK :-2,X,:-3,Y; Input Terminal
                Gets values for X,Y from a file which reads: X=  5, Y=  6
ASK "YES or NO? "ANS=0YES
                Reads value for ANS.  Hitting '=' retains the value 'YES'
```

@Operators

```
!       Writes a CR/LF to the current output device
"..."   Writes quoted string to current output device
#       Writes a Formfeed to current output device
#expr   Writes character defined by the expression
$..."   Writes Escape Sequence to current output device
%expr   Changes output format (for later use by TYPE)
:+expr  Spaces to specified column on current output device
:-expr  Reads and ignores specified number of characters
```

&BREAK

BREAK [line or group number]

The BREAK command terminates a FOR or UNTIL loop causing the program to exit from the loop and continue with the command following the BREAK (or, if a non-zero line number is specified, to branch to that line). If more than one loop is active (nested loops), BREAK exits only from the current level; several BREAKS may thus be necessary. If NO loops are active when a BREAK is executed, the command becomes a NOP (except if a non-zero line number was specified, in which case the GOTO branch is performed). BREAK will exit from a loop even if it is encountered in a subroutine call; however the previous format and I/O device selection saved by the subroutine will not be restored in that case.

@Examples

```
FOR I=1,10; IF(I,LT,5)2; BREAK
                Stops the loop when I=5
UNTIL(A,GT,100); FOR I=1,10; IF(I,NE,6)3; BREAK; BREAK 2
                Exits from both loops when I=6 and branches to Group 2
```

&COMMENT

COMMENT [followed by other text]

Any command beginning with a 'C' is treated as the logical end-of-line. No further information is taken from that line, thus the text of a comment may contain any printable or non-printable characters terminated by a CR. Execution of a COMMENT causes the program to check for external DO calls. See also the description of the 'X' command, and note that 'blank' lines may be included in a program without marking them as 'comment' lines.

@Examples

```
FOR I=1,100; Create a short delay, interruptable by external DO calls
FOR I=1,100; Next; Create an un-interruptable delay
ASK X=-1; C Initialize X to -1; use the '=' key to signal End-of-Data
```

&CONTROL

CONTROL CODES

The following CONTROL CODES are recognized when entered from the keyboard:

CTRL/C [03] This causes LDF to return to the monitor system. The contents of memory are saved, hence one may restart the program by using the '.START' command. CTRL/C restores the KEYPAD on 'VT52' terminals to normal (numeric) operation. CTRL/C is NOT trapped during a USE HOST command and can thus be sent to the remote machine.

CTRL/E [05] This character has a special meaning ONLY during a USE HOST command. It causes a normal EXIT, returning to the command following the 'U H' command and restoring the CTRL/C trap and KEYPAD operations.

CTRL/F [06] This character interrupts the program and returns to 'Command Mode'. A '?' is printed, followed by the line number where the interrupt occured. If a 'U H' command is interrupted by CTRL/F, neither KEYPAD operations nor the CTRL/C trap will be restored, hence the user will be unable to return to the monitor system by typing CTRL/C. To re-enable these features, type 'U H (CR)', followed by CTRL/E.

CTRL/G [07] This character initiates selection of a new search character during a MODIFY or MOVE command. It rings the 'bell'.

CTRL/H [08] This is the BACKSPACE character; it has no special meaning in LDF. Typing CTRL/H will move the cursor left one position on most video terminals, causing the next character to replace the previous one on the display. This does NOT mean that the first character has been DELETED! It is still there, but 'hidden' by the second character, and can therefore cause confusing results.

CTRL/I [09] This is the horizontal TAB character; it has no special meaning to LDF, and is not recognized by all terminals. LDF has its own 'tab' operator which can be used to position text at any desired column. It is recommended that this operator (:) be used, rather than CTRL/I.

CTRL/J [10] This is the LINEFEED character; typing LINEFEED causes the currently entered command string to be retyped. This is especially useful on HARDCOPY terminals where deletions are echoed as '\'s, and the contents of the line are therefore confusing. LINEFEED is also useful on video terminals: typing CTRL/L followed by CTRL/J (LF) will re-display the current command string at the top of the screen.

LINEFEED is also used in a slightly different way during a MODIFY or MOVE command: in this mode it retypes the remainder of the line being changed and then exits from the command. This is the normal method for terminating a MODIFY or MOVE operation.

CTRL/L [12] This is the FORMFEED character; it is IGNORED during command input in the sense that it is not put in the command string. Most versions, however, detect FORMFEED and use it to clear the display. Typing FF, then LF, will display

the current command line at the top of the screen. CTRL/L also clears the screen during an ASK command (but is otherwise ignored).

During a MODIFY or MOVE command, CTRL/L causes the editor to look for the next occurrence of the search character. The screen is NOT erased in this case.

CTRL/M [13] This is the RETURN or CARRIAGE RETURN (CR) character. CR must be typed at the end of every command line before LDF will respond to the commands on the line. CR is also one of the ways to terminate input to the ASK command (SPACES, COMMAS and TABS also work).

During a MODIFY or MOVE command a CR means: 'end the line at this point'. Hitting RETURN instead of LF during this command will thus delete all text to the right of the current position.

CTRL/Q [17] This is the 'XON' character. It causes output to the terminal to be resumed after being stopped by CTRL/S.

CTRL/S [19] This is the 'XOFF' character. It causes output to the terminal to be suspended so that the operator can look at the display before lines at the top are removed.

Some terminals have a SCROLL key that sends XOFF the first time it is pressed and XON the second time. Other terminals have a similar key that displays exactly one more line of text each time it is pressed, again using XOFF and XON.

CTRL/U [21] Typing CTRL/U while LDF is in Command Mode deletes the entire command string and prints a new prompt (*). Typing CTRL/U during a MODIFY or MOVE command will delete all the characters that have been typed out (i.e. the beginning of the line); editing may then continue. CTRL/U serves only as a terminator during an ASK command - use the DELETE key.

CTRL/Z [26] This character is used throughout LDF as the END-of-FILE code. Typing CTRL/Z while LDF is in Command Mode causes a '?' to be printed, followed by another prompt (*). CTRL/Z is also used as the EOF code on most HOST machines, hence typing this character during a USE HOST command is one way to close a data file.

ESCAPE [27] This character is used by many terminals (and other I/O devices) to indicate the start of a character sequence which causes some special function to be performed. LDF itself gives no special meaning to this character, hence it will be included in the command string if it is typed in. 'ESC' is sometimes referred to as an 'altmode' key for monitor commands. ALTMODE is actually a distinct key that happens to be found in the same place (upper-left-hand corner) on older terminals. Terminals with an ESCAPE key do not have ALTMODE, and vice-versa. See also ESCAPE SEQUENCES.

ALTMODE [125] This key is used by the monitor system as a command option; it has no special meaning in LDF, and in fact, produces the code for a 'right brace' (}), which is a legal character in command strings.

PREFIX [126] This key is recognized by the monitor as yet another kind of 'altmode'. Terminals with a PREFIX key do not have ALTMODE or ESCAPE keys (and vice versa). This key generates the 'vertical bar' (|) and as such is a legal character in LDF programs.

DELETE [127] This character (called RUBOUT on some terminals) removes the previous character on the command line. On video terminals the character actually disappears from the screen, while on hardcopy terminals a '\' is echoed each time the DELETE key is used. DELETE works in the same way during a MODIFY or MOVE command that it does during Command Mode, with the minor difference that one cannot erase the line number of an existing line. Note that LDF ignores further use of the DELETE key when there are no more characters to be deleted.

Use of the DELETE key during an ASK command cancels the ENTIRE number, not just the previous digit. DELETE is thus very similar to CTRL/U (which is not recognized during an ASK command). A '?' is printed each time DELETE is typed, reminding the user to re-enter the entire number.

&DO

DO [list of subroutine identifiers]

The DO command executes a portion of the current program as a subroutine, returning to the next command when the call is completed. The anticipated range of the call is specified by the DO command: single lines, whole groups, or partial groups; the subroutine itself, however, may limit the actual range of the call by means of the RETURN command, or effectively extend it by means of further DO calls. The ON command and the Library DO command, as well as FSFs, FQUE and KEYPAD calls, all use the same 'DO' mechanism (sometimes with a restriction to positive arguments). DO calls always save and restore the current FORMAT specification, as well as the current INPUT/OUTPUT devices. Subroutines may thus change these settings, knowing that the previous values will be restored (with some restrictions) upon completion of the DO call. (Note effect of NEXT and BREAK commands.)

@Examples

DO	Executes the entire program, then does the next command
DO 0	Same - '0' refers to the entire program
DO-.01	Executes all of Group 0 (can't use 'DO 0' - see above)
DO 1.1	Executes line 1.10
DO 1	Executes all lines in Group 1 unless stopped by a RETURN
DO-1.5	Executes all lines in Group 1, starting with line 1.5
DO 1,2,3	Executes all lines in Groups 1,2,3; same as DO 1;DO 2;DO 3
DO .5	Executes line xx.50 in the current group
DO-.5	Executes lines xx.50-xx.99 in the current group
DO SUB	Executes subroutine identified by the value of a variable
DO 2+LINE/10	Executes a specific line in Group 2 (if LINE=0, then DO 2)

&ERASE

ERASE [line or group number]

The ERASE command removes one or more lines from the current program. Only a single parameter may be specified and the program returns to command mode following execution of the command. Lines removed are GONE FOREVER! There is no way to restore erased lines, hence be careful when typing 'W' (WRITE) that you do not accidentally hit the 'E' (ERASE) key instead!

@Examples

E	Erases the entire program
E 0	Same
Erase All	Erases portion of the program denoted by the value of 'AI'
E-.01	Erases all lines in Group 0 (can't use 'E 0' - see above)
E 1.1	Erases only line 1.1
E 1	Erases all of Group 1
E-1.5	Erases lines 1.50-1.99
E .1	Erases line 00.10 (from command mode)

&ESCAPE

ESCAPE SEQUENCES

The $ operator may be used in an ASK or TYPE command to generate an escape sequence. The appearance of this operator sends ESC to the current output device, followed by whatever characters are to be included in the sequence. The last character must be followed by a closing quote (") to indicate the end of the sequence. The SPACE character cannot be sent as part of the sequence; SPACES are used to delimit ARITHMETIC EXPRESSIONS, which are evaluated and then output under a temporary %-5 format. Characters, other than SPACES and those required for the sequence, may be included in the string.

@Examples

```
TYPE $\"          Exit from 'hold-screen' mode on a VT52 terminal
TYPE $="          Select 'alternate keypad mode' on a VT52 terminal
TYPE $H$J"        Clear the screen on a VT52 (or equivalent) terminal
TYPE $"#          Clear the screen on a Tektronix 4010 terminal (ESC FF)
ASK #13$I$K"      Clear previous line on a VT52 terminal (CR ESC I ESC K)
TYPE $\$($[?2l"   Turn a VT100 into a VT52 (last char is a lowercase 'L')
TYPE $[10;1H"     Position the cursor at (row=10,col=1) on a VT100 terminal
T $[ R; C H"      Position cursor at row,col location given by variables R,C
T $[ R; C H*"     Same, plus print a '*' at that loc. (VT-100 plot command!)
```

&FILENAMES

FILENAME EXPRESSIONS

Filenames used by the Library, List, Load and Open commands have the form:

DEVICE:FILENAME.EXTENSION[SIZE],option=value

where DEVICE: is a character string terminated by a colon which specifies the physical or logical name of a device. Only the first 4 characters are significant. The name may be user-assigned (i.e. IN:, OUT:, 1234:), or it may be the conventional name of a device. If no device name is specified, the name DSK: is assumed.

FILENAME is a character string containing an arbitrary sequence of letters and numbers which designates the name of the file. Only the first 6 characters are significant and lower-case letters are converted to upper-case. Filenames may be entirely numeric, and are not necessary when refering to non-directory devices. In LIST commands the filename may be replaced by a '*', which means 'all files with the specified (or assumed) extension'.

EXTENSION is a character string which must be preceded by a period. Only the first 2 characters are significant. The extension is commonly used to denote the file-type, i.e. Program, Data, Random Access, etc. Another common practice, especially when several people share the same disk, is to use the programmer's initials for the extension, allowing each person to reference his or her own unique set of programs. If no extension is supplied, the 'L' commands assume an extension of .FP, while the 'O' commands assume an extension of .DA. In LIST commands only, a '*' may be used for the extension to indicate 'all files with the specified name and ANY extension'.

[SIZE] is an optional parameter which consists of an arithmetic expression inside square brackets. It is used when OPENing a file to specify the maximum file size (which is used to optimize the location). The value of the expression is always taken modulo 255(10). If no [size] is specified, the system will generally put files in the largest available 'empty' area.

COMMA options indicate secondary input/output devices as well as other attributes peculiar to a given command. The available options are: E,L,T,N, R,S; the comma option is not programmable.

The EQUAL option processes an arithmetic expression (limited to +/- 31.99), which is used in different ways by various commands. The expression must be preceded by an '=' sign.

SPACES and 'case' are ignored in filename expressions: the name 'My File' is the same as 'MYFILE'. VARIABLE (programmable) expressions may be used in place of the FIXED expressions shown above. These are indicated by enclosing portions of the character string in parentheses. Characters inside parentheses are processed as arithmetic expressions whose value is inserted in the filename at that point. Either NUMERIC or CHARACTER values may be programmed. If the value of the expression is NEGATIVE, it is treated as a character code; if the value of the expression is POSITIVE, it is converted to a numeric string, using as many of the leading digits as possible. Both the ':' and '.' characters may be programmed, hence the entire 14-character filename expression can be specified at run-time. SPACES or '@-signs' may be used to fill in unused character positions, and multiple expressions may included within one set of parentheses, using commas as separators. Naturally only the integer part of the value is used.

@Examples

```
I s myprog              Short form of Library Save DSK:MYPROG.FP
O O MYFILE              Short form of Open  Output DSK:MYFILE.DA
O I RXA1:DATA.01=9.5    A complete filename, incl. error return option
O I DATA(I)             Specifies any file DATA0.DA to DATA99.DA
O I DATA(I,J)           Specifies any file DATA0(.DA to DATA99.DA
                        (This assumes that I & J are positive - negative
                        values could program non-numeric characters too)
o o data.(C1,C2)        If C1=-65 and C2=-66 the name becomes DATA.AB
O O JUNE(DA).(YR-1900)  Programs 2 digits of the name and the extension
List Only *.vZ          Lists all files having an extension of .VZ
List All LTA1:,E=4      Lists all files + 'empties' on LTA1: in 4 columns
```

&FOR

FOR variable=initial value, [increment,] final value; commands

The FOR command establishes a loop controlled by the value of the associated loop variable. The assumed increment is '1' (note that the increment is the 'middle' parameter). Parameters may be positive or negative, integer or non-integer. A non-integer increment may cause abnormal operation due to small arithmetic errors: e.g. 'I' is never EXACTLY equal to '0' in the loop 'FOR I= -1, .1, 1'. See also the NEXT and BREAK commands.

@Examples

```
FOR I=1,10; DO 5     Calls Group 5 ten times.  Exits with I=11.
FOR I=1,5,10; FOR J=I,J+4; TYPE J
                     Types the values 1,2,3,4,5,6,7,8,9,10
FOR I=1,5,10; TYPE I; NEXT; FOR J=I,J+4; TYPE J
                     Types the values 1,6,11,12,13,14,15
FOR I=1,10; FOR J=1,10; TYPE A(I,J); NEXT; TYPE !
                     Types the matrix A one row at a time
FOR I=J=1,K=5,L=10; TYPE I,J
                     Types the values 1,1,6,1 (J is not incremented)
```

&FORMAT

FORMAT OPTIONS FOR THE TYPE COMMAND

There are three different types of format options available: integer output, mixed decimal, and exponential (scientific) notation. The number of digits and the amount of space used for printing those digits is controlled by the argument given to the % operator. NEGATIVE arguments suppress all leading spaces, causing the output to be left-justified. INTEGER arguments produce integer output (no decimal point). An argument of XX.YY produces XX digits with up to YY decimal places (note that YY must always be a two-digit number). An argument of 00.YY produces YY digits in exponential format, covering the entire range of possible numbers (1E-615 to 9E+615). As a special case, an argument of 0 (or no argument) is equivalent to an argument of 00.10 (10-digit exponential format). No more than 10 significant digits will ever be output; leading spaces or trailing 0's are used to fill the remainder of the space. XX must be < 32 and YY must be <128. Positive formats always supply at least one leading space, plus a second one if the number itself is positive, otherwise a '-' sign. The sign 'floats' to the position just ahead of the first significant digit.

In order to output as many significant digits as possible, the format requested by the user may be temporarily suspended. Thus output in 'XX.YY' formats will drop decimal places in favor of leading digits, in the limit shifting to an integer (no decimal point) format. If the value cannot be printed in an integer format it will automatically be output using an exponential format, keeping the desired number of digits (XX) in all cases. An 'XX.YY' format which requests more decimal places (YY) than the number of digits (XX) automatically reduces YY to XX-1.

@Examples

TYPE %1	Print values <10 as integers with one or two leading spaces
TYPE %1.00	Same (the value is an integer, no matter how it is written)
TYPE %30	Print values <1e30 as integers with up to 31 leading spaces
TYPE %-30	Print values <1e30 as integers with no leading spaces
TYPE %6.04	Print values <1e6 as numbers with up to 4 decimal places
TYPE %-6.04	Print values <1e6 as mixed decimals with no leading spaces
TYPE %6.4	Improper format (6 digits, 40 decimal places) becomes %6.05
TYPE %10.1	Default format until changed by the user (equal to %10.09)
TYPE %30.1	Print values <1e30 with up to 10 decimal places, 21 spaces
TYPE %.05	Print values in scientific format with 5 significant digits
TYPE %-.05	Same, but left-justify the value by omitting leading spaces
TYPE %.50	Print 10 significant digits followed by 40 trailing zeros
TYPE %,	Same as %.1 (all 10 significant digits in exponential form)
TYPE %expr	Select the format according to the value of the expression

&FSF

FOCAL STATEMENT FUNCTIONS

Focal Statement Functions are user-defined functions written in the program and called by specifying the appropriate line or group-number. The differences between a FSF call and a DO call are that (a) an argument list may be passed to the subroutine and (b) a value can be returned by the function.

The form of a FSF call is: F(linenumber,arg1,arg2,arg3,arg4,arg5). Some or all of the arguments may be omitted and as they are not passed recursively, may not include other FSF calls that also have an argument list. All arguments are evaluated before passing control to the subroutine with the value of each being saved in one of the 'protected' variables: a,b,c,d,e. (arg1 goes in 'a', arg2 in 'b', etc.) The value of the LAST ARITHMETIC EXPRESSION processed by the subroutine is returned as the value of the function call.

Note: users with upper-case-only terminals may use the symbols (!,",#,$,%) in place of (a,b,c,d,e), remembering that these symbols are used as operators in ASK/TYPE commands, and must therefore be preceeded by a '+' sign to output their values.

The actual line or group-number is best replaced by a suitably-chosen variable whose value is defined at the start of the program; this permits modification of the function itself without having to change other parts of the program and provides added perspicuity as well (compare 'F(TAN,angle)' with 'F(3.1,angle)').

@Examples

```
3.1 SET FSIN(a)/FCOS(a)          F(3.1,angle) is the tangent function
4.1 SET (FEXP(a)+FEXP(-a))/2     F(4.1,X) is the hyperbolic cosine of X
5.1 SET FDAY(b*256+a*8+c-78)     F(5.1,MO,DA,YR) sets the system date
6.1 SET FEXP(FLOG(a)*b)          F(6.1,X,Y) computes X-to-the-Y-power
7.1 FOR I=$=1,!;SET $=$*I        F(7.1,N) returns N factorial
8.1 FOR I=N,-1,$=0;S $=$*!+C(I) F(8.1,X) evaluates a polynomial function
9.1 for d=c=0,4;set c=c+(a-8*fitr(a/8)*10^d
                                 F(9.1,N) computes the OCTAL value of N
```

&GOTO

GO [line or group number]

The GO (or GOTO) command causes an unconditional branch to the line or group specified. The branch causes the PDP-12 display to be refreshed. The IF, QUIT and RETURN commands as well as the Error Return option in 'L' and 'O' commands all perform 'GOTO' branches, and all (except QUIT and RETURN) may indicate a 'group' or 'sub-group' as the target, rather than a specific line. (See examples.)

@Examples

G	Starts program at lowest-numbered line
G 0	Same
G 1	Branches to the lowest-numbered line in Group 1
G 1.5	Branches to line 1.50 - gives error if missing
G-1.5	Branches to line 1.50 (or the next-higher line)
G .5	Executed in a program: branches to line xx.50
	Executed from command mode: starts at line 00.50
G 2+LINE/10	Branches to a computed line number

&HESITATE

HESITATE [number of seconds, to nearest (.1 second]

The HESITATE command creates a programmable pause during which external subroutine calls may occur (see FQUE). The display on a PDP-12 is also refreshed during a HESITATE command. This command normally assumes the presence of a Real Time Clock running with the interrupt system enabled. For machines without a RTC, or for non-interrupt operation (BATCH mode, RTS/8 background jobs), the HESITATE command is automatically patched to use a software timing loop. (Location 10115 controls the length of this loop which may be adjusted experimentally for 0.1 sec intervals.) The longest delay possible with a single HESITATE command is 204.7 seconds.

@Examples

H	Maximum of 1 clock-tick delay (typically 1/100 second)
H 0	Same (Gives FQUE calls and the scope refresh a chance)
H .5	Delays 1/2 second
H T	Delays T seconds, measured to the nearest 1/10 second
H 10	Create a 10-second pause for external DO calls, refresh

&IF

IF(expression)-,0,+ or IF(expression)-,0,1,2,3,4,5...

The IF command provides a conditional GOTO branch based upon either the SIGN or the VALUE of an expression. The expression must be enclosed by a pair of matching parentheses (any of the 4 types may be used - spaces preceeding the left member are optional). The list of target line numbers appears following the test expression, with the symbolic placement as shown above, i.e. the first line number is the target if the expression is NEGATIVE, the second if the expression is EXACTLY ZERO, and so forth. If more than 3 line-numbers are found, the INTEGER VALUE of the expression is used to select the desired branch. Line numbers may be OMITTED (or the value '0' used) if no branch is to occur on that condition. Line numbers may also be computed from expressions THAT DO NOT CONTAIN COMMAS (i.e. no double-subscripted variables or functions with more than one argument). RELATIONAL expressions may be used directly, with the '-' branch taken if the relation is TRUE, the '+' branch if it is FALSE. The PDP-12 display will be refreshed when the branch occurs.

@Examples

IF(A,LT,5)2	Branch to the lowest-numbered line in Group 2 if A is < 5
IF(X-Y).2,.3,.4	Branch to xx.20, xx.30 or xx.40, depending on sign of X-Y
IF(N),-2.5;	Branch to line 2.50 (or next higher line) only if N=0
IF(K),,5.1,6,,7	Branch to line 5.10 if K is greater than 0 but less than 2, or GOTO group 6 if K is between 2-3, or GOTO group 7 if K is between 4-5, else continue with next sequential command.
IF(I=I+1,eq,10)	Increment I and test to see if it is equal to 10. The '-' branch will be taken if this is true, else the '+' branch
IF(20-I)I=0	Test if I>20, and if so, reset I to 0. This command does not branch because a line-number of '0' means 'continue'
IF(20-I).*I=1	Test if I>20, and if so, reset I to 1. The '.*' converts the value '1' (or any arbitrary value) to '0' (no branch)
IF(9-X)FOUT('*)	Print a '*' if 'X' is greater than '9' (FOUT returns '0')

&ICMNDS

INPUT COMMANDS

The following INPUT commands are available:

Input Buffer Input File Input Host Input Terminal
Input Close

INPUT commands are two-word (two-letter) commands which change the Current Input Device (CID), and may, in addition, change the setting of the 'input echo' switch by means of the ',Echo' and ',Noecho' options. When the echo is enabled (as it is by default for the Terminal), each character read from the input device is automatically sent to the Current Output Device (COD). Input from devices other than the Terminal will not echo unless the ',Echo' option is specified. The available input devices are:

Buffer	Any one of the 4 internal buffers
File	Any one of 4 input files
Host	An optional communications line
Terminal	The default input device, restored after an error occurs

Input from a File or one of the Buffers is further selected by means of the /Buffer option. File input comes from the most recently selected file by default, which is File/1 until this setting is changed - either by an Open Input command or by another Input File command. Input from one of the Buffers is handled similarly, except that there is no default: the buffer desired must be explicitly selected to begin with. Selection without a buffer option will resume input from the next sequential location. Initially, however, the 'buffer pointer' is at the 'EOF' point.

@Examples

I T,N	Disable the keyboard echo; characters typed will not appear
I T	Re-enable the keyboard echo (CTRL/F also enables the echo)
I F	Select the previous file as the CID; do not echo input
I F,E	Same, but echo each character to the Current Output Device
I F/3	Select Input File #3 as the Current Input Device
I F/expr,Echo	Select the file designated by the expression & enable echo
I B	Resume input from the previously selected buffer
I B,E	Same, but with the echo enabled
I B/0,E	Reset input to the beginning of buffer/0, echoing each char
I H,E	Select the HOST as the CID, echoing all characters received

The INPUT CLOSE command serves two purposes: (1) it severs all connections between the program and the designated file and (2) it selects the terminal as the CID. An input file can always be restarted by an 'O I,R' command until it is CLOSED, and can be used by the FRA function. CLOSING an INPUT file thus protects it from inadvertent uses.

@Examples

I C	Close the current input file
I C/3	Close input file/3 (makes buffer/3 the default)

&JUMP

JUMP line- or group-number

The JUMP command is a conditional GOTO which branches to the specified line UNLESS a character is waiting in the terminal-input buffer. The effect is that the program loops, waiting for the operator to hit ANY key on the keyboard. The JUMP command does nothing further with this character, leaving it in the input buffer for other commands. The FIN function combined with the IF/ON commands may be used to read and test the character, or else an ASK :-1 command may be used to discard it. Note that external DO calls can occur during JUMP loops and the PDP-12 display will also be refreshed each time the branch occurs.

@Examples

```
3.9 JUMP 3        Branch to the Group 3 unless a key has been struck
5.1 JUMP .1       Hang on this line for a key, checking external DO calls
9.5 SET FMQ(FADC(FRS())); JUMP .5
                  Display the A/D channel (selected by the Right Switches
                  on a PDP-12) in the MQ register, until a key is struck
```

&KEYPAD

KEY [list of keypad assignments of the form: label=DO call]

The KEY command associates a subroutine (DO) call with one of the keypad keys on a VT52, WT78, VT100* or Z19 terminal. Defining any key puts the terminal in 'alternate keypad mode', while executing a 'KEY ZERO' command (or returning to the monitor with CTRL/C) restores 'numeric keypad mode'. (*VT100 terminals must be put in 'VT52 mode' first: use TYPE $\$($[?2l".)

Only one KEYPAD subroutine call is normally permitted at a time - further calls are locked out. To permit keypad calls 'inside' a subroutine that was called from the keypad you must execute another KEY command (a 'null' command may be used for this purpose). The 'cursor' keys are treated as an independent group: they continue to provide cursor-movement commands until at least one of them has been assigned; they can then be turned on or off with a simple FPAL call. (The 'function' and 'color' keys remain active once they have been assigned since they have no other usage.) All labels must be entered from the main keyboard (not from the keypad), and except for the function keys, can be abbreviated to a single letter. The 'down-arrow' label is the letter 'vee' (upper or lower case). Note that the 'color keys' vary on different terminals: B,R,G on a VT52; Y,B,R on a WT78; B,R,W on a Z19, and PF1-PF4 on a VT100.

Keypad calls may specify only a single line or group; negative values (a 'partial group' call) are not permitted (the sign is ignored), while the value '0' (normally associated with the 'whole program') is used to disable that particular key.

@Key Labels:

0	1	2	3	4	5	6	7	8	9
.	F1	F2	F3	F4	F5	^	v	(	)
Zero	Enter	Red	White	Blue	Yellow	Gray		-	,

@Examples

```
K                 A null command which allows nested keypad calls to occur
K Z               Remove all keypad assignments and restore normal operation
K 0=0             Remove the assignment for the '0' key, leaving all others
K Z,1=1,2=2       Define the '1' and '2' keys to call their respective groups
K enter=9.1       Use the ENTER key to execute line 9.1
K ^=.1,v=.2       Define (and enable) the 'up' and 'down' keys for this group
K -=11,,=12       Define the '-' and ',' keypad keys on a VT100 terminal
SET FPAL(0,3167)Disable the cursor keys (does not affect any of the others)
SET FPAL(4,3167)Reenable the cursor keys (preserves previous assignments)
TYPE $)"          Disable all keys (except the 'color' and 'function' keys)
TYPE $="          Reenable all keys, preserving their previous assignments.
```

&KONTROL

K [list of arithmetic expressions]

The KONTROL command is used by the PDP12 and LAB8E versions to open & close relays. On the PDP12 the argument is interpreted as a 6-bit value which is stored in the 'relay register' and thus sets all six relays simultaneously. On the LAB8E the argument list specifies bits in the 'digital output register', where positive values mean 'set', and negative values mean 'clear'. This version also allows two additional constructs: (1) groups of bits may be set and/or cleared together by enclosing a list of arguments in parentheses, and (2) the lower 4 bits of the register can be loaded by a single expression which is preceeded by an '=' sign. The 'bit names' used in this version run from 1-12, not from 0-11, which allows the value '0' to be used to clear all bits. Since the processing time for a simple numeric argument is typically several milliseconds, short pulses may be created by closing and then immediately opening the same relay.

@Examples:

K	Open all relays (PDP12)
K-1	Close all relays
K 5	Close relays 3 and 5, open 0,1,2,4
K expr	Set the relay register to value of expression
K	Clear all bits in the Output Register (LAB8E)
K,1	Clear all bits, then set line #1 (bit '0')
K-1	Clear bit #0, leave others as they were
K 2,3,4	Close lines 2,3,4
K(-2,-3,-4)	Open lines 2,3,4 all at once
K 1,-1	Create a short pulse using bit #(
K(1,-1)	Set bit #0 ('ON' has precedence over 'OFF')
K=5	Set bits '9' and '11' (lines '10' and '12')

&LCMNDS

'L' COMMANDS

The following 'L' commands are available:

Load commands:	Load In	Load Go	Load Do
Libr commands:	Lib Erase	Lib Load	Lib Save
	Lib Comment	Lib Quit	
List commands:	List All	List Only	
	List Files	List Programs	

&LIBRARY

Library Command [filename+other options]

The LIBRARY commands perform operations such as saving & deleting (erasing) programs, as well as miscellaneous things such as returning to the monitor (L Q) or changing the header line (for listing purposes only). The Library Load command is identical to the Load In command. Note that these commands assume a '.FP' extension, hence to erase a data file one must explicitly specify the extension (.DA).

Please refer to the section on FILENAMES for additional information.

Lib Comment	Changes the comment (header) line in the program
Lib Erase*	Removes the file specified from the directory
Lib Load	Loads a program from the device specified (none=DSK:)
Lib Quit	Exits from the program to the monitor (used under BATCH)
Lib Save*	Names & saves the current program on the device specified

*Note: These two commands close the output file before they are executed.

@Options

=line number	Specifies error-return point for the Library Erase command If the file is missing, and a non-zero line number was specified, the program will branch to this line.

@Examples

L C 12(PI,2+2)	Places the name '1234' in the Header line - this is an easy way to test out programmable file names
L E MYPROG	Removes the name MYPROG.FP from the directory of DSK:
L E MYDATA.DA	Removes the name MYDATA.DA from the same directory
L L NUPROG	Reads NUPROG.FP into memory, then returns to Command Mode
L Q	Recalls the Keyboard Monitor (the '.') - the same as typing CTRL/C. Programs which run under BATCH must use the 'L Q' command to return to the monitor level; the program can be restarted after a L Q by use of the monitor .START command.
L S TEST01	Saves the current program on DSK: with the name TEST01.FP; The name TEST01 and the current date are automatically put in the header line to identify the program when it is later recalled by a L L (or L I) command.
l s rka1:junk	Saves the text buffer on device RKA1: with the name JUNK.FP

&LIST

List Command [filename+other options]

The LIST commands search the directory of a specified device (default=DSK:) reporting the files present and (optionally) the amount of free space left. Commands are provided for showing all files or only a selected subset. The LIST ONLY command, for example, searches for a particular file, which is a useful thing to do before saving a program with a name you may have already used! Wildcards (*) may be used for listing all programs with a given name or extension.

List All	Lists programs on selected device, starting with filename
List Only	Lists only programs of the form given (default is .FP)
List Programs	Lists all .FP programs, starting with filename
List Files	Lists all .DA files, starting with filename

@Options

,Empties	Lists empty areas as well as files
=columns	Changes the number of columns used (default=last choice)
* (wildcard)	Match any name or extension (? marks are not permitted)

@Examples

L A,E	Lists all files on DSK:, including the empty areas
l a,e=8	Same, but displays 8 columns across (132-col term)
L A JUNK.TM,E	Lists files starting with JUNK.TM, plus 'empties'
l o myprog	Checks for the existence of MYPROG.FP on DSK:
L O RXA1:*.VZ	Lists all files with the extension .VZ on RXA1:
L O X,E	Lists empty areas on DSK: (if X.IP does not exist)
L F	Lists all the data files (*.DA) found on DSK:
L F LTA1:,E	Lists all the data files (and empties) on LTA1:
L P,E	Lists all programs (*.FP) on DSK:, plus 'empties'

&LOAD

Load Command filename[+options]

The LOAD commands read a program (.FP) file into memory. The program can also be started at a specified line (Load and Go calls), or a part of the program can be executed as a subroutine (Load and Do calls), after which the 'calling' program is automatically reloaded. In the case of subroutine calls, a program that has not been saved since it was last modified will be saved automatically on DSK: with the name FOCAL.JM before the 'L D' call is executed. (Note: this 'save' operation also closes any open output file.)

Load In	Loads the specified program and then returns to command mode
Load Go	Loads the program and automatically starts at specified line
Load Do	Loads the program and executes the specified subroutine, then reloads the calling program, continuing with the next command

@Options

=line or group	Specifies the starting point (L G) or subroutine call (L D). Omitting this option (or specifying '0') executes the entire program.

@Examples

L I PROG01	Loads PROG01.FP into memory from DSK:
L I DTA1:MYPROG	Loads MYPROG.FP from DTA1:
L G BANGUP	Loads BANGUP.FP into memory and starts at the first line
L G SETUP=2	Loads SETUP.FP and starts at the first line in Group 2
L D SUBPRG	Executes SUBPRG.FP, then returns to calling program
L D INVERT=3	Loads INVERT.FP and executes Group 3, then returns
L D SUBS=SubNo	Loads SUBS.FP and executes the routine specified by 'So'

&MODIFY
&MOVE

MODIFY line1 or MOVE line1,line2

The MODIFY/MOVE commands provide a limited-funtion editor with which the user can conveniently make changes to existing programs without having to re-type any of the unchanged part of the code. These commands also allow selected lines of a program to be copied (with or without any changes) in order to insert additional lines in their proper logical sequence or else to create several variations of a particular command string. For information on moving an entire group of lines, see MOVGRP.

The MODIFY command is normally executed from Command Mode, and always returns to command mode upon completion. Input comes from the Current Input Device, which is normally the Keyboard, but can just as well be a file or one of the four internal buffers. 'Self-modifying' programs, for instance, may output a command string to one of the buffers and then execute a MODIFY or MOVE command after selecting that buffer as the Current Input Device.

The only difference between MODIFY and MOVE is whether a single line or two lines are specified. In the case of a MOVE, line #1 remains unchanged. After the command has been typed (including the (R), the editor waits for a 'search character' to be entered. There is no prompt for this, so if nothing seems to be happening, try typing a second CR. This will cause the entire line to be written out as the editor searches for a CR. To modify the line at a certain point, pick a search character which is at or just beyond that point. If the character is not unique the editor may stop before getting to that point, in which case you may type CTRL/L to repeat the search. When the search character is found, you may add new commands just by typing them in, or delete unwanted commands by using the DELETE (or RUBOUT) key. You may also change the search character, or simply skip to the end of the line without making further changes. While the editor is waiting for input the following CONTROL KEYS may be used:

CTRL/F	Abort the entire command without making any changes
CTRL/G (bell)	Change search character; type CTRL/G plus new character
CTRL/L (FF)	Find the next occurrence of the search character
CTRL/J (LF)	Keep the rest of the line as it was without any changes
CTRL/M (CR)	Terminate the line at this point (delete to end-of-line)
CTRL/U	Delete from current position to the beginning-of-line
DELETE	Removes previous character (does not remove line number)

@Hints

A semicolon is a good search character when there are multiple commands on one line. Typing CTRL/L will advance one command at a time. You can also initially select a semicolon as the search character and then, when you are near the part of the line being changed, select a different character.

Remember to type LINEFEED, not RETURN, when you finish with changes in the middle of a line. RETURN truncates the line at that point, thus requiring a second MODIFY to restore the commands that were deleted.

CTRL/L is not echoed back to the terminal, hence no screen-erasure or page-eject will occur on terminals that respond to a FF. The line number of the line being changed (line #2 in the case of a MOVE) is normally displayed on video terminals, but not on hard-copy terminals. This leaves more room for changes on terminals that do not remove deleted characters. Disabling the INPUT ECHO suppresses printing of the line as it is modified, but does not suppress printing the line number. Attempts to modify a non-existent line are treated as a NOP, not an error; this allows the MOVGRP routine to work.

&MOVGRP

USING THE 'MOVGRP' COMMAND FILE

The MOVGRP routine was written to automate the process of moving an entire group of lines all at once. This routine generates the necessary 'Direct Commands' to move every possible line in the group, placing these commands in one of the buffers which is then selected as the Current Input Device. The default input device is re-selected upon completion of the moves. Note that this procedure requires the use of BUFFER/2 and BUFFER/3.

To execute the MOVGRP command file from Command Mode, do the following:

```
*Open Input/3 MOVGRP; Input File <CR>
Move Group? xx To Group? yy <CR>
(line numbers will appear on the terminal)
*
```

To move a second (or third ...) group without the overhead associated with the initial Open Input command, do the following:

```
*Open Input,Restart; Input File <CR>
Move Group? xx To Group? yy <CR>
(line numbers will appear on the terminal)
*
```

In 'short-hand' notation this is: 'O I,R;I F', followed by answers to the questions. A curious observation: terminating the response to the second question with a SPACE, rather than a CR, causes the line numbers to be output on separate lines, rather than all on one line.

&NEXT

NEXT [line or group number]

The NEXT command marks the logical end of a FOR or UNTIL loop, causing the next cycle of the loop to be started as well as identifying the location in the program where execution will resume after the loop runs to completion. The NEXT command may have an optional line or group number which causes a GOTO branch at the conclusion of the loop. Note that NEXT commands may occur in subroutines called by the loop (this a common occurrence, in fact). A NEXT command executed in this context causes an automatic RETURN from the subroutine call without, however, restoring either the format or I/O device selection saved by the subroutine (DO) call. If a NEXT command is executed when no loop is in progress the command is ignored unless it specifies a non-zero line number, in which case the GOTO branch is executed. Note that NEXT commands which are not executed by the loop on its final pass are also ignored. This can happen with UNTIL loops (which skip the loop entirely if the test expression is negative) as well as with FOR loops that have conditional branches. (See UNTIL for syntax requirements when used with NEXT.)

Loops may be written either with or without NEXT commands - the use of NEXT is entirely optional, allowing loops with different end-points to be placed on a single line, and to facilitate the programming of multiple-line loops. A given NEXT is automatically matched with the inner-most loop existing at the time it is executed. Thus nested loops with the same logical end point require 'nested NEXTs'.

@Examples

FOR I=1,10;NEXT;TYPE !	Types only a single CR/LF
FOR I=1,10;DO-.2;NEXT 2	Executes the block of statements following the
(subroutine)	loop, then branches to Group 2 when I=11.
FOR I=1,10;DO-.2	This is similar, however in this case the NEXT
(subroutine;NEXT)	command is in the subroutine itself, bypassing
	restoration of any format and I/O device changes

&ON

ON(expression)-,0,+ or ON(expression)-,0,1,2,3,4,5...

The ON command provides a conditional subroutine call based upon either the SIGN or the VALUE of an expression. The expression must be enclosed in a pair of matching parentheses (any of the four kinds may be used, and spaces preceeding the left member are optional). The list of subroutines follows the test expression with the order shown symbolically above, i.e. the first subroutine is called if the test expression is NEGATIVE, the second if the expression is EXACTLY ZERO, and so forth. If more than 3 subroutines are given, the INTEGER VALUE of the expression is used to pick the desired one. If no call is desired for a given condition, simply omit the corresponding identifier (or use the value '0'). Subroutine calls may be computed from expressions THAT CONTAIN NO COMMAS (i.e. no double-subscripted variables or functions with more than one argument are allowed). RELATIONAL expressions may be tested, with the '-' call taken if the relation is TRUE, and the '+' call taken if it is FALSE.

@Examples

ON(A,LT,5)2	Call Group 2 if A is less than 5, then continue program
ON(X-Y).2,.3,.4	Call line xx.20, xx.30 or xx.40, depending on sign of X-Y
ON(N),-2.5;	Execute the subroutine starting at line 2.50 only if N=0
ON(K),,5.1,6,,7	Execute line 5.10 if the integer part of K is greater than 0 but less than 2, or execute group 6 if K is between 2-3, or execute group 7 if K is between 4-5, then continue with the next sequential command.
ON(I=I+1,eq,10)	Increment I and test to see if it is equal to 10. The '-' call will be executed if this is true, else the '+' call

&OCMNDS

'O' COMMANDS

The following 'O' commands are available:

File Initialization:	Open Input	Open Output	
File Termination:	Output Abort	Output Close	Output Dump
Device Selection:	Output Buffer	Output Echo	Output File
	Output Lineprnt	Output Scope	Output Terminal
(Optional)	Output Host	Output Plotter	

&OPEN

OPEN COMMANDS

The OPEN INPUT and OPEN OUTPUT commands do the work of setting up the files used by the Input File/Output File commands. The form of these commands is (xxxx=Input or Output):

OPEN xxxx [/buffer] [filename] [options]

where the three major parameter groups are indicated. The '/buffer' option selects one of FOUR possible INPUT FILES (0-3), each file being associated with a given buffer. It is meaningful only for OPEN INPUT commands, since output files always use 'buffer/0' no matter what value is specified. When no '/buffer' option is given, the default is the PREVIOUS VALUE (initially '/1'). Note that Input File/0 is the same as the Output File (if there is one); the FRA function takes advantage of this. Simultaneous reading and writing of file/0 is prohibited; however once the Output File is CLOSED, it can be immediately opened for input with an 'Open Input/0,Restart' command.

When a file has been attached by either of the OPEN commands it remains accessible to the program until either it is replaced by another file, or an INPUT CLOSE command is used to remove it. Note especially that a file is still 'attached' following an OUTPUT CLOSE command; while it can no longer be used for output, it can still be used for input! The only way to 'disconnect' a file once it has been OPENED is to use the INPUT CLOSE command.

The form of the filename is discussed in the FILENAME section. Note that the SQUARE BRACKET [size] option is often useful in OPEN OUTPUT commands.

@Options:

,Restart — This switch resets the 'file pointers' to the beginning of the file. Thus all previous output is lost and input commences with the first character of the file. NO FILENAME should be specified when this option is used. A file may be RESTARTED at any time until it is finally 'disconnected' by an INPUT CLOSE command.

=linenumber — This option allows the program to recover from an error condition by executing a GOTO branch to the line specified. If this option is omitted (or has the value '0') any OPEN error will interrupt the program and print a short diagnostic message. Common errors are: an OUTPUT file is already open, or the Output Device is Write Protected, or the INPUT file can not be found.

OUTPUT COMMANDS

There are two classes of OUTPUT commands: those dealing with system files (Output Abort, Output Close and Output Dump), and those which only change the Current Output Device: Output Buffer, Output File, Output Echo, Output Host, Output Lineprinter, Output Plotter, Output Scope and Output Terminal.

Output Abort [size] This command 'closes' the output file without writing an EOF mark. If no size is specified the tentative file is deleted, otherwise it is made permanent using the size specified. This provides a way to create 'dummy' files. '0 A' commands also reset the COD; they are ignored unless there is actually a file open.

Output Close [size] This command closes the output file in a normal manner, including writing an EOF mark (CTRL/Z). The optional size parameter allows the user to increase (or decrease??) the size of the file. An O C command always resets the COD device to that selected by the '0 E' command. If no file is open this is all the O C command will do.

Output Dump This command dumps the current contents of buffer/0 to the file device. It does not change the COD. For certain devices (such as the LQP printer) this provides a convenient method of achieving interactive (line-by-line) output in spite of the internal buffering.

Output Buffer [/bufno] This command selects one of the four buffers as the COD. If the '/buffer' option is included, output will be directed to the beginning of the buffer; otherwise it will continue from the previous end-point. Note that Buffer Output is NOT the same as File Output. Any of the four buffers may be used in this way for 'scratch-pad' output. Each buffer holds a maximum of 255 characters. The 'echo option' is not recognized by the '0 B' command; output to a buffer cannot be echoed to another device.

Output Echo [,options] This command selects and/or defines the ECHO device. The Echo Device is the default output device restored by the '0 A' and '0 C' commands as well as by the error-recovery routine. It is normally equivalent to the TERMINAL, however it may also be set to the PDP12 SCOPE, or the LINEPRINTER. The user may thus choose which device is to be used for error messages. '0 E' selects the Echo Device as the COD. '0 E,L' selects the lineprinter as the Echo Device and then makes it the COD device as well. Available options are: L,S,T (Lineprinter, Scope and Terminal).

Output File [,options] This command selects the OUTPUT FILE as the COD. Options are available for echoing output to the FILE on some other device (Echo,LPT,or Terminal). The '=' option provides an error return in case

there is no file open.

Output Host [,options]	This is an optional command which selects a HOST machine as the COD. Output to the Host may also be echoed to the Terminal, Lineprinter or Echo device by specifying the appropriate option (T,L,E).
Output LPT [,options]	This command selects the user's Lineprinter as the COD. If desired, output sent to the Lineprinter can be echoed to either the Echo Device (,E) or to the Terminal (,T). To send output to a file, AND to the Lineprinter AND to the terminal (3 places), use the sequence: O L,T;O F,L.
Output Plotter	This command selects the incremental plotter as the COD. No echo options are available, and output to the plotter is further controlled by the PLOT command (which sets the initial position on the paper) and the two 'Plot Variables' (S., D.) which control the Size and Direction of the characters.
Output Scope	This command selects the PDP12 (or LAB8/E) scope as the COD. If neither of these is available the O S command selects the Terminal instead.
Output Terminal	This command selects the Terminal as the COD. No echo options are recognized, hence if output to more than one device (including the terminal) is desired, the terminal should be selected as the secondary output device: 'O L,T' sends output to both the LPT and the Terminal; 'O T,L' does not.

&OPS

Arithmetic, Relational and ASK/TYPE Operators

Arithmetic Operators: (All may serve as command delimiters)

=	A=B=	Variable on left receives value on right
+	+A+B	Addition or Sign indicator
-	-A-B	Subtraction or Sign indicator
/	A/B	Division (lower priority than multiply)
*	A*B	Multiplication
^	A^B	Raise A to the (+,-) integer power of B
() <> [] $		Perform inside operations first
'	'A	Value of character following the '

Priority: (low) =, +, -, /, *, ^, ()-<>-[]-$, ' (high)

Relational Expressions: (must be enclosed, TRUE=-1, FALSE=+1)

LT	lt	Less than	(A,lt,B)
LE	le	Less than or equal	[a,LE,b]
EQ	eq	Equal to	<A,eQ,b>
NE	ne	Not equal to	a,Ne,B$
GE	ge	Greater than or equal	[A,GE,B$
GT	gt	Greater than	a,gt,b]

ASK/TYPE operators:

!	Start a new line
"..."	Output literal string
#expr	Convert expression to ASCII character
$..."	Output an Escape Sequence (see ESCAPE)
%expr	Change output format (see FORMAT)
:+expr	Tab to specified column (if not beyond)
:-expr	Read (and ignore) specified # of chars

&PLOT

PLOT X,Y,LINE,MARK or ILOT HORZ,VERT,CHAR

The optional PLOT command has two forms: one for use with an incremental plotter (Calcomp, Houston Instruments, Benson, etc.), and one for use on VT52/VT78 video terminals. There are also two related VIEW commands for displaying (plotting) data on Tektronix terminals, the PDP12 or a LAB8/e 'scope.

For info on incremental plotters, see PLOTXY. For info on the VT52/VT78 plot command, see VTPLOT. See TEKTRONIX and SCOPE for info on the VIEW commands.

&PLOTXY

INCREMENTAL PLOTTERS

Machines equiped with an incremental plotter are supported by the PLOT command and the OUTPUT PLOTTER routines. Plotters with a resolution of 0.01" or 0.25mm are considered 'standard', however plotters with 0.005" or 0.10mm step size can also be used, with a corresponding improvement in line quality, but no increase in resolution. The maximum plotting area is +/-20.47" (or +/-51.1cm). Plotter operations overlap calculations, providing smooth, continuous motion. A choice of 8 different axis orientations is provided.

The command used is 'PLOT X,Y,L,M' where 'X' and 'Y' are the coordinates of the desired pen position, specified in some -arbitrary- set of units. The default is 'inches' for 'english' plotters and 'cm' for 'metric' plotters. The actual values of 'X' and 'Y' are scaled by the value of the variable 'M.' ('Multiplier.') before being sent to the plotter routines. Thus the program can change the scale factor at any time just by changing the value of 'M.'. (The default value is '1'; setting M. to '-1' inverts the axes!)

The value of 'L' determines the type of line drawn between two consecutive points. If L=0 the pen is raised and NO line is drawn. If L=1 the pen is lowered, drawing a straight line (approximately) between points (to within the limit of the plotter step size). If L=-1 the pen is raised during the move, and the position reached at the end is taken to be the new origin. Calls with L=-1 are used to advance the paper in order to begin a new plot.

The value of 'M' is used to select the type of MARK to be drawn upon reaching the end point. 16 different marks are available, with several possible variations of each type. Mark #3, for instance, is a 'tick mark' which can be used to divide an axis into convenient segments. Mark #0 is a 'null' - other marks provide 'open' or 'filled' symbols for identifying data points:

1 dot	5 cross (x)	9 f. triangle	13 f. diamond
2 dash	6 o. circle	10 o. square	14 asterisk
3 tick	7 f. circle	11 f. square	15 5-point star
4 plus	8 o. triangle	12 o. diamond	16 Maltese cross

The labeling of the 'X' and 'Y' axes is essentially arbitrary; commonly 'X' moves the paper, while 'Y' moves the pen. To achieve this orientation the variable 'R.' ('Rotation.') should be set to '0'. Many programmers prefer to rotate the axes clockwise 90 degrees, so that 'X' moves the pen and 'Y' moves the paper. This makes it easier to view the plots as they are being drawn, and is achieved by setting the variable 'R.' to the value '2' (this is the default value). Other values of 'R.' may be used to rotate the axes in increments of 45 degrees; values are modulo 8, hence 'R.=-2' and 'R.=6' are effectively the same. For best results the value of 'R.' should not be changed unless the pen is at the origin. Odd values of 'R.' have a special effect: all motions are 'stretched' by the square root of 2 (approximately 1.414), due to the difference between 'diagonal' steps and 'axis' steps.

All other 'plot directions' are relative to the orientation of the axes set by the value of 'R.'. Within this framework, motion along the '+Y' axis is considered to be 'Direction 0', motion in the '+X' direction is regarded as 'Direction 2', and so on (clockwise) in steps of 45 degrees. The value of the variable 'D.' controls the direction (orientation) of the MARKS and any TEXT output on the plotter. If 'D.' is equal to '2', then annotation will be parallel to the 'X' axis - no matter if this is the 'paper-feed' or 'pen motion' direction.

The last special 'plot variable' is 'S.' ('Size.'), which controls the size

of ASCII characters drawn on the plotter. (The size of the MARKS is fixed, except for the effects of a 45 degree rotation.) The basic character cell is 6 units wide by 10 units high (0.06" by 0.10" on an 'inch' plotter). Of this 6x10 area, only 4x6 is used for the character. Characters using this basic grid are scaled by the value of 'S.', in increments of 0.5. Thus if 'S.=1.5', the character cell becomes 9x15, with 6x9 used for the symbol and the remainder for spacing.

Characters sent to the plotter by the OUTPUT PLOTTER command are thus drawn at a position determined by the PLOT command, with a size and direction set by the values of 'S.' and 'D.'. These values may be changed by the user at any time, however no changes will be made to the plotter routines until the next PLOT command in order to prevent 'shifting' the origin.

In addition to the full upper and lower-case character set (with decenders) the plotter also responds to CR, LF and BS. The ability to overprint thus makes possible composite symbols, for instance 'plus-or-minus' can be drawn with the commands: 0 P;T "+"#8"_";0 T. ('#8' is 'backspace').

Sample programs which illustrate features of the incremental plot routines are PLDEMO, XYAXIS and PLOTER.

&QUIT

QUIT [optional line number]

The QUIT command stops execution of the program, with an optional RESTART feature. The restart option may also be used for error recovery purposes.

@Examples

QUIT	Stops the program and returns to Command Mode
QUIT 0	Same
QUIT 1.1	Stops the program and then restarts it at line 1.1
QUIT-1.1	Does NOT stop the program, but remembers line 1.1 as the 'error-restart' point. If ANY error occurs, the program will automatically be restarted at line 1.1.

&RETURN

RETURN [optional line number]

The RETURN command forces an exit from any form of subroutine call: DO, ON, Library DO, FSF's, FQUE and KEYPAD calls. The normal return is to the command immediately following the subroutine call, however DO, ON and 'Lib DO' calls (only) can branch to a line given by a NON-ZERO line number. (This option is ignored by FSF's, FQUE and KEYPAD calls). If a RETURN command is executed when there is nothing to return to (no subroutine calls are in effect), it behaves just like a QUIT and returns to Command Mode. It is thus recommended that programs be terminated by a RETURN command (rather than by QUIT) so they can be used as subroutines (via the 'Lib Do' command) if desired. The RETURN command can also be used to mark the end of FOR or UNTIL loops; such usage is not common, however.

Subroutines are not required to end with a 'RETURN' (as they are in FORTRAN or BASIC) since the 'DO' call itself specifies the 'range' of the routine; use of this command is thus largely a matter of convenience for terminating calls with conditional code. The 'ranges' permitted are: single line, partial group, entire group, or the entire program. The RETURN command serves to shorten the specified range, and/or to provide (by means of the optional line number) an alternate return point.

@Examples

R	Exit from the subroutine call or return to Command Mode
R 0	Same - the RETURN command always looks for a line number
R 5.1	Exit from the call and branch (GOTO) line 5.1 if the call was made by a DO, ON or Lib Do command. If the call was made by a FSF or an 'external' source such as the KEYPAD, the branch option is ignored and the normal return occurs.
R expr	Selects the type of return from the value of the expression

&SET

SET [list of expressions]

The SET command evaluates Arithmetic and/or Relational Expressions for various purposes. One purpose is to save the result of such expressions in a memory location specified by the name of a VARIABLE; another purpose is to execute functions which perform some sort of I/O operation. In this case the value returned by the function may or may not be of interest. Whether a result is saved or discarded depends on whether or not the expression includes the name of a variable followed by an assignment (=) operator. When a program is being run in TRACE mode, the SET command is equivalent to the TYPE command (including interpretation of the symbols !,",#,$,%,: as operators), hence the result of each expression is output as it is evaluated.

@Examples

S A=1,B=2,C=3	Assign different values to three variables
S A=B=C=1+2+3	Assign the value '6' to each of three variables
S A=1+B=2+C=3	Assign the value '6' to A, '5' to B, and '3' to C
S FMQ(FSR())	Display the switch-register setting in the MQ
S FRA(-1,4)	Select the desired Random Access storage mode
?S PI?	Type out the value of PI (3.14...) (Trace mode)
S-1	Put '-1' in the Floating Accumulator as a FSF result

&SCOPE

PDP-12 and LAB8/E DISPLAY ROUTINES

The OUTPUT SCOPE command selects the refreshed display on a PDP12 or LAB8E. The screen is refreshed entirely by routines in IDF and hence represents a compromise between frequent updates and the desire to use the CPU for running the user's program. Refreshing occurs whenever the program is waiting for keyboard input, on any branch or subroutine call (as well as when the program merely 'falls into' the next line), and during a HESITATE command. The VIEW command can be used to 'turn off' all refreshing until the program either returns to Command Mode or the user enables refresh mode again.

The text display on the PDP12 provides 32 lines with 85 characters per line plus automatic 'wrap-around' (the '8/e has only 16 lines of 64 characters). All 128 characters are displayable, with control characters shown as lowercase letters with a bar across the top. Approximately 750 characters can be displayed at one time; CTRL/S and CTRL/Q can be used to control the rate at which text scrolls off the screen.

Data points as well as text may be displayed on the screen. The 'VIEW X,Y' command puts points in the 'view buffer', which is completely separate from the 'text buffer'. Thus either buffer can be cleared without affecting the other one. On the PDP12, in fact, the two buffers are displayed on different 'channels', hence the selector switch must be set to '1+2' in order to see both images.

Sending a FORMFEED (CTRL/L) to the 'scope will clear the text buffer, while a 'VIEW-1' command will clear the data buffer. Interestingly enough, however, a 'VIEW-3' command causes the first 2 data points to reappear! Here is how the VIEW command works:

VIEW +1	Enables refreshing of the display
VIEW [0]	Disables refreshing of the display
VIEW -1	Displays 0 points of the 'view buffer'
VIEW -N	Displays N-1 points of the " "
VIEW X,Y	Places a new point in the view buffer

To plot points on the screen the VIEW command is called with TWO arguments: the first is the horizontal coordinate (0-1023); the second is the vertical coordinate (-512 to +511). (The actual resolution of the PDP12 display is only 9 bits (-256 to +255), but to simplify displaying 10-bit A/D readings, an extra factor of 2 was included.) The buffer can be cleared by a 'V-1' command, and then instantly restored by a 'V-N' command. Of course any new points which are 'viewed' after clearing the buffer will overwrite previous values. The 'FVB' function can also be used to read or change data in the View Buffer. The maximum number of points that can be displayed is roughly 1400 (380 on a 16K machine).

&TYPE

TYPE [arithmetic, relational expressions, operators]

The TYPE command evaluates arithmetic/relational expressions and converts the resulting binary values to decimal values which are formatted and sent to the Current Output Device. In addition the TYPE command recognizes six OPERATORS which are used for various control purposes. The SET command is equivalent to TYPE when a program is being TRACEI.

@Examples

T 1+2+3	Evaluates the expression and writes result to the COD
T "2+2="2+2	Same, but outputs an identifying label for the result
T ?2+2?	Same, using the TRACE operator tc provide identification
T I=I+1	Increments I and outputs new valte
T 1,2,3,4,5!	Outputs 5 values, followed by a CR/LF
T H=.5,4-H,4+H	Sets H to 1/2, outputs values of .5, 3.5 and 4.5
T "Ready?":-1	Prints prompt, then waits for one character from the CID

@Operators

!	Writes a CR/LF to the current output device
"..."	Writes quoted string to current output device
#,	Writes a Formfeed to current output device
#expr	Writes character defined by the expressicn
$..."	Writes Escape Sequence to COD (see ESCAPE)
%expr	Changes output format (see FORMAT)
:+expr	Spaces to specified column on current output device
:-expr	Reads and ignores specified number of characters

&TRACE

TRACING PROGRAMS

Programs which do not appear to be working correctly can be TRACED. This means that each command will be listed on the Current Output Device as it is executed, which allows one to find the point where the wrong BRANCH is taken. In addition, all SET commands are automatically converted to TYPE commands when the 'trace switch' is set, consequently intermediate values which are computed by a SET command will also be output without having to make any other changes to the program.

The TRACE SWITCH is set whenever a '?' is encountered in the program (unless it is part of a quoted string or following the character-value operator). A second '?' then turns off the trace switch, a third turns it on again, and so forth. To trace the entire program, simply start it with a 'GO?' or 'DO?' command. To trace selected portions of a program, bracket the desired commands with a pair of '?'s.

This feature can also be used as a 'shortcut' for generating labels in an ASK or TYPE command by 'tracing' the name of a variable as it is used. A disadvantage to this scheme is that these labels are NOT output during an actual trace because of the toggling nature of the trace switch.

@Examples

```
DO 1,2?,?3,4              Traces only subroutine 2
X?This is a comment?;     Prints the string 'This is a comment'
TYPE %4.03,?A=?5,? B=?6 Prints 'A=  5.000 B=  6.000'
ASK ?A,B,C ?              Prompts for the variable by printing its name
FOR I=1,5;?SET I          Prints the values 1,3,5 (every other pass)
```

&TEKTRONIX

TEKTRONIX SUPPORT

Plotting on a Tektronix 4010 terminal is done by the 'VIEW X,Y,Z' command, where 'X' is the horizontal coordinate (0-1023) and 'Y' is the vertical coordinate (0-780, approximately). The 'Z' value may be -1, 0, or +1 according to the desired type of line:

Z= -1	Draw a 'dot' at the current point
Z= [0]	Move without drawing a line
Z= +1	Draw a line from the previous point

On Z-19 'Graphics-Plus' terminals which can also 'undraw' lines and fill or erase areas, the following additional Z values are permitted:

Z= +2	Un-draw a line from the previous point
Z= +3	Fill the rectangular area between points
Z= +4	Erase the rectangular area between points

Interaction between the VIEW and TYPE commands: the 'home' position of the cursor corresponds to VIEW 0,767. The character cell is 14 (wide) by 22 (high), including the space between characters and lines, and the character is drawn with its lower-left-hand corner at the position last specified by a VIEW command. To clear the screen, use a TYPE $"# command.

For additional information, see ESCAPE and FCUR.

&UNTIL

UNTIL(expression); commands

The UNTIL command starts a conditional loop which executes the commands on the remainder of the line until the value of the expression is NEGATIVE. If the expression is initially negative, none of the commands are executed and the program skips to the next line. UNTIL loops may be terminated by a NEXT command, but the user should be aware that if the loop is not executed the commands following the NEXT command will not be executed either. Also, when UNTIL is used with a NEXT command that specifies a branch, the right-hand parenthesis MUST be followed by a SPACE. UNTIL commands that test relational expressions loop until the expression is TRUE (-1); a null expression creates an endless loop that can only be interrupted by typing CTRL/F.

@Examples:

```
U();S FMQ(FSR())          An endless loop which displays the SR in the MQ
Z I;U(FTRM(),EQ,'/');A X(I=I+1)
                          A loop to read an array of X values ended by '/'
Z I;U(C(I=I+1)=FIN(),EQ,13);N
                          A loop to read in a character string ended by CR
UNTIL(-FKEY()) ;NEXT .5 Wait for a key (note the SPACE preceding the ';')
```

&UCMNDS

'U' COMMANDS

The only two-letter 'U' command currently available is USE HOST. This command connects the TERMINAL to a communications line linking a remote HOST (such as a VAX, PDP10, or PDP11 system) to the PDP8 or PDP12. After executing the 'U H' command, all characters entered from the keyboard, with the exception of CTRL/E and CTRL/F are sent to the Host, while all characters received from the Host are sent to the ECHO DEVICE (which is also made the Current Output Device). All I/O to the Host is done through the interrupt system, using the 'XON/XOFF' protocol to limit the rate of transmission. Baudrates as high as 9600 may be used, though some problems have been noted with the KL8E and M707 interfaces at high baud rates.

The USE HOST command allows the programmer to run programs on both the PDP8 (or PDP12) and the Host machine concurrently. File transfers can be set up interactively and programs developed in a concerted fashion, using the best features of both systems. The USE HOST command can be built into programs, or it can be used from Command Mode. The normal EXIT is by means of typing CTRL/E; note that CTRL/C is NOT trapped during a USE HOST command, allowing this character to be sent to the remote machine. The USE HOST command also temporarily disables the KEYPAD in versions which have this feature so that keypad editors can be used on the Host machine. A CTRL/E exit will restore both the CTRL/C check and the Keypad facility, however the user may need to reset 'Alternate Keypad Mode' upon returning to LDF (TYPE $=" will do it).

It is helpful to prompt the user with messages such as 'You're on the VAX' and 'You're back in LDF' before and after the 'U H' command. That way the programmer can mentally adjust to the command set of the machine he or she is currently using.

&UWF

CONVERSION OF UWF PROGRAMS TO LDF

The following steps must be taken to convert UWF programs so they will work with LDF:

Step 1: Convert UWF Program ('.FC') Files to ASCII. This is done with the following set of commands:

```
.R UWF
*L C program
*O O program.LS; WRITE; ) C
```

Any extension may be selected - '.LS' is only a suggestion. Since many of the changes below involve 'search and replace' operations, it is probably most convenient to use your favorite editor to work on this ASCII file, rather than using the MODIFY command later on. MODIFY can be used, but it is often tedious and error-prone. After all the changes have been made, you can then do

Step 2: convert this ASCII file back into a LDF Program File ('.FP' file). This is done with the following series of commands:

```
.R LDF
*O I program.LS; I F
*L S program [;ERASE]
```

The ERASE command is only necessary if you want to immediately repeat this process with a second program.

Step 3: All places where '!' is used in connection with matrix operations (Double-subscripted arrays, row and column loops) must be changed to '@'.

Step 4: All places where '#', '$' and '%' are used in FOCAL Statement Functions must be changed to either 'a', 'b' and 'c' (if you have a LC terminal), or else '!', '"' and '#' (if you have an UC terminal, or else prefer the 'punctuation marks'). FSF's with 4 or 5 arguments (using the first variables defined by the program) must be changed to use the variables 'd' and 'e' (or else '$' and '%'). FSF's with more than 5 arguments can not be converted for use in LDF (but also do not work in versions of UWF with a hashed symbol table either).

Step 5: Variable names longer than 2 characters must be checked for uniqueness. UWF uses the first two characters of the name, LDF uses the first+last.

Step 6: Variables whose names begin with a special symbol (other than the 'secret' variables) will now be treated as 'local' variables. The use of such variables must thus be determined.

Step 7: Expressions that test character values must be changed from '8-bit' ASCII to '7-bit' ASCII. This is done by subtracting 128 from the value of all constants. Expressions that use the 'character-value' operator (') do not need to be changed.

Step 8: JUMP commands must be examined. Those which are used as 'computed DOs' (or in earlier versions, 'computed GOTOs') must be changed to extended 'ON' or 'IF' commands. This requires changing the command word (letter) and adding two commas after the closing parenthesis.

JUMP commands used only for keyboard branches need not be changed.

Step 9: Due to the reversal of values for 'true' and 'false' the COMPARISON OPERATOR must be changed in all Relational Expressions, except any used in UNTIL commands. ('True' is now -1', 'false' is '+1'.) An alternative in IF/ON commands with relational expressions is to reverse the branch options. The first branch is now 'condition true' rather than 'condition false', which is MUCH easier to think about.

Step10: UNTIL commands must be changed for the same reason given in Step 9. The UNTIL command now loops until the test expression is Negative, not Positive, thus UNTIL commands using relational expressions need NOT be modified because the two sets of changes cancel each other.

Step11: All 'X' (eXecute) commands must be changed to 'Set' commands. 'X' is now a 'nop'.

Step12: ASK or TYPE commands using the '#' operator must be examined. In versions which use '#' to clear the screen (PDP12, LAB8E) this operator MUST be followed by a comma (which was optional before). In versions which use '#' to generate a CR-without-LF, the value '13' (followed by a terminator) must be added i.e. '#' becomes '#13,'.

Step13: ASK or TYPE commands using Negative Formats to specify the number of digits printed in Scientific Notation must be changed to use a format of the form '00.xx'. This change affects only very old versions of UWF.

Step14: ASK or TYPE commands which use the '$' operator to dump the symbol table must be removed; there is no way to do this anymore - the '$' operator now sends Escape Sequences.

Step15: The argument of HESITATE commands must be decreased by a factor of 1000. Hesitate times are now in SECONDS rather than milliseconds, with the smallest interval a TENTH-second. PDP12 (or LAB8E) versions can no longer use the 'H' command for programming the clock frequency - it is fixed at 100 Hz (but can be changed with an FPAL function if absolutely necessary).

Step16: Calls to the FBUF function (PDP12, LAB8E) must be changed to FVB. Calls to the FTRG function must also be changed. The FLGS function can be used (with an appropriate setting of the Event Flag Mask) or else an 'external DO' call can be set up by the FQUE function.

Step17: If you have a Tektronix terminal you must change FJOY to FCUR; also all references to 'XJ' and 'YJ' must be changed to 'X.' and 'Y.'.

Step18: If you have an Incremental Plotter, all references to the variables '$F','$R','$D' and '$S' must be changed to 'M.','R.','D.' and 'S.'; also change OPEN OUTPUT PLTR: commands to OUTPUT PLOTTER commands.

Step19: OUTPUT DATE commands must be changed to FDAY calls ('SHOW FDAY()').

Step20: OUTPUT BUFFER commands must be changed to OUTPUT DUMP commands.

Step21: OPEN OUTPUT TTY: or 'O O' commands must be changed to 'O T' (OUTPUT TERMINAL) commands.

Step22: OPEN INPUT TTY:,ECHO or 'O I,E' commands must be changed to INPUT TERMINAL commands. 'O I' commands WITHOUT the ',E' option must be

changed to 'I T,N' commands. (The echo sense has been reveresd.)

Step23: ONLY LIST commands must be changed to LIST FILES.

Step24: OPEN OUTPUT LPT: commands using UWF's lineprinter handler must be changed to OUTPUT LINEPRINTER (O L) commands. Those that called the SYSTEM handler must now be followed by an OUTPUT FILE command.

Step25: OPEN OUTPUT filexx commands must be followed by OUTPUT FILE[,ECHO]. Opening an output file no longer automatically makes the file the Current Output Device; conversely an 'O D' command immediately following an 'Open Output filexx' command should be removed. If an error return option is included in such commands, it MUST be preceded by an '=' (which can replace the 'space' used in UWF).

Step26: OPEN RESUME OUTPUT commands must be changed to OUTPUT FILE[,ECHO].

Step27: OPEN INPUT filexx commands must be followed by 'INPUT FILE[,ECHO]'. Opening a file for input no longer automatically makes the file the Current Input Device. Conversely, the 'D I,E' command that often follows 'Open Input filexx' commands should be removed. If an error return option is included in such commands, it MUST be preceded by an '=' sign (which can replace the 'space' used in UWF).

Step28: OPEN RESUME INPUT commands must be changed to INPUT FILE[,ECHO].

Step29: OPEN SECOND commands must be changed to OPEN INPUT/2 followed by an INPUT FILE command. Switching between two input files requires the use of the '/buffer' option.

Step30: OPEN RESUME SECOND commands must be changed to INPUT FILE/2.

Step31: OUTPUT ECHO LPT: commands must be changed to 'O E,L'.
OUTPUT ECHO TTY: commands must be changed to 'O E,T'.

Step32: LIBRARY BRANCH commands must be changed to JUMP commands. ('L B' is used only in rather old versions.)

Step33: LIBRARY NAME commands must be changed to LIBRARY COMMENT commands. Note: 'L N' commands set the 'program changed' flag, causing 'L G' commands to save the program; 'L C' commands do not set this flag.

Step34: LIBRARY EXIT commands must be changed to LIBRARY QUIT commands.

Step35: LIBRARY DELETE commands must be changed to LIBRARY ERASE commands. If such commands use an error-return option, it MUST be preceded by an '=' sign (which may replace the 'space' used by UWF).

Step36: LIBRARY GOSUB commands must be changed to LIBRARY DO commands. If such commands specified a line- or group-number, this expression MUST be preceded by an '=' sign.

Step37: LIBRARY RUN commands must be changed to LIBRARY GO commands. If such commands specify a line-number, it must be preceded by an '='.

Step38: LIBRARY LIST commands must be changed to LIST PROGRAM commands.

Step39: LIBRARY CALL commands (used in 'Command Mode' to load a program) should be replaced by LOAD IN or LIBRARY LOAD (your choice).

Step40: Notes:

&VIEW

VIEW [X,Y,Z] or \IEW [X,Y]

There are two flavors of the VIEW command: one for 'Tektronix' terminals, and one for the PDP12 display. In addition there is a version of the PLOT command for 'viewing' data on a VT52 (or WT78) terminal. Check TEKTRONIX, SCOPE, and VTPLOT (respectively) for additional help.

&VTPLOT

VT52 GRAPHICS SUPPORT

Very effective graphics displays can be produced on VT52 or VT78 terminals by using the PLOT command. PLOT will also work on VT100 terminals if they are first set to 'VT52 mode' (TYPE "$\$($[?2l" does this); to plot in 'ANSI mode' one can use a programmable escape sequence (see ESCAPE for details).

The PLOT command has the form: PLOT HORZ,VERT,CHAR, where each of the three parameters may be an arithmetic expression. HORZ values range from 0-79, VERT values from 0-23, and CHAR values from 0-127. Parameters may be omitted, in which case the value '0' is assumed. The origin for HORZ and VERT values is at the bottom-lower-left, hence all coordinates lie in Quadrant I. The PLOT command allows any character (including the special graphics characters) to be displayed at any position on the screen. Resolution is thus limited to full character cells (80x24), however the 'scan line' characters can be used to interpolate vertically within the cell.

CHAR values from 32-126 generate the normal 'printing' character set of the terminal. These characters may be specified either by an arithmetic expression, or often more conveniently, by including the desired character in the PLOT command, preceded by the 'character value' operator ('). Values from 1-31 (normally control codes) are mapped into the special graphics characters. The value '127' is ignored by the terminal. The following table indicates the graphic symbol produced by a given code (on VT52s, not VT100s):

0 (null)	8 plus-minus	16 scan3	24 sub-3
1 c/d	9 right arrow	17 scan4	25 sub-4
2 solid block	10 ellipsis	18 scan5	26 sub-5
3 1/	11 divide sign	19 scan6	27 sub-6
4 3/	12 down arrow	20 scan7 (bot)	28 sub-7
5 5/	13 scan0 (top)	21 sub-0	29 sub-8
6 7/	14 scan1	22 sub-1	30 sub-9
7 o (degrees)	15 scan2	23 sub-2	31 paragraph

The scan-line characters allow one to increase the vertical resolution from 24 to 196 (120 on a VT100). The command for doing this looks like:

```
PLOT X,Y=...,20-Z*FRAC(Y
```

where 'X' is the HORZ coordinate, 'Y' is the VERT coordinate (NOT an integer!) and 'Z' is '7' for a VT52 ('4' for a VT100). Try the example below with Z=0, then repeat with Z=7 (or 4) to see the difference:

```
@Example        FOR X=0,79;PLOT X,Y=X/5,20-Z*FRAC(Y)
```

Interaction of the PLOT and TYPE commands: the PLOT command puts the cursor at the desired screen position and adjusts the 'tab' counter appropriately; a pair of 'PLOT...TYPE' commands can thus put text anywhere on the screen. A null PLOT command moves to the bottom of the screen, hence any additional output (for instance the next command prompt) will scroll the display up one line. Disabling the INPUT ECHO suppresses the prompt; one can also move the cursor to the upper-left-hand corner with a 'PLOT,23' command. The VT52 ignores commands to move the cursor 'off-screen', hence if either coordinate is 'out-of-bounds', the cursor will move in only one direction. The easiest way to clear the screen is to send a FORMFEED to the terminal, either with a 'TYPE #' command or simply by typing CTRL/L on the keyboard. (Formfeeds are ignored during keyboard input.)

&VARIABLES

Variable Names

There are four classes of Variables available: (1) protected variables; (2) global variables; (3) local variables; (4) double-subscripted variables.

PROTECTED variables are those that are not automatically 'wiped out' by the ZERO command. There are 7 such variables: @,PI,a,b,c,d,e; the last 5 (a-e) may also be referenced by the names: !,",#,$,%. (These are the punctuation symbols used as operators in ASK/TYPE commands. This is for the benefit of users with upper-case only terminals). These 7 variables are used for many special purposes and hence should not be used without some thought on the part of the programmer.

The variable @ is used by the double-subscripting routine; @ is assumed to have been set to the maximum value of the first subscript.

The variable PI is initialized to 3.14159..., correct to approximately one part in 1E11. Users who desire the most accurate value of expressions involving this constant should therefore not set PI to something else! (All decimal approximations of PI are less accurate than the internal value.)

The variables (a-e) are reserved for passing FOCAL Statement Function arguments, however the user is free to employ them for other non-conflicting purposes ('scratch-pad' variables, etc. - the variable '$' (d) for example, is convenient to use in summations.)

All the remaining variables created by the program are unprotected from the ZERO command. They are stored in whatever space is available for user variables and hence will be effectively set to '0' by that command. Variables defined by the user's program are identified by a two-character name and a single 12-bit subscript. The internal name is abstracted from the name in the user's program by keeping only the FIRST and LAST letters. Thus 'TEMP' is reduced to 'TP'. Similarly the internal subscript may differ from the subscript(s) appearing in the program. Note especially that variable names (unlike other symbols in LDF) are NOT 'case-blind': upper- and lower-case variable names are DIFFERENT, hence the names 'KW', 'Kw', 'kW' and 'kw' represent 4 different variables. As discussed below, the first two are classified as GLOBAL variables, while the latter two are considered to be LOCAL variables.

GLOBAL variables are defined as those whose names begin with an upper-case letter or those with an explicit subscript, regardless of their name. The seven PROTECTED variables are also considered to be GLOBAL. Such variables can be uniquely referenced from any part of the program, i.e. 'globally'.

LOCAL variables are members of the remaining set they must have names that begin with a lower-case letter (other than a-e), and no explicit subscript: the variable 'i', for instance. Such variables are UNIQUE to a particular section (group) of the program. The variable 'i', for example, may be used in two completely different ways by parts of the program that interact with each other. This is accomplished by setting the internal subscript of such variables equal to the GROUP NUMBER of the line in which it appears. From this it should be clear that the LOCAL variable 'i' in Group 12 is the same as the GLOBAL variable 'i(12)'. Local variables can thus be accessed from any part of the program just by adding an explicit subscript.

As mentioned above, all variables have an internal subscript, even if they are not subscripted in the user's program. Un-subscripted variables with Uppercase names are given an internal subscript of '0'. (Thus there is NO

DISTINCTION between the variables 'A' and 'A(0)'.) Unsubscripted variables with lowercase names are given a subscript equal to the current group number, with those defined in Direct Commands considered to be in Group 0.

Finally, variables may also be defined with TWO subscripts. This greatly simplifies the programming of matrix operations. Since only a single subscript is used internally for each variable, however, an algorithm must be employed to uniquely transform the two indices into a single 12-bit number. The method chosen allows the second subscript to have any value whatsoever, while the first subscript is constrained to be less than some maximum value which is saved in the protected variable '@'. Thus '@' is often called the 'double-subscripting constant', or the 'dimension constant'.

Programs using two-dimensional subscripts generally involve manipulation of 'conformal arrays', that is, arrays which all have similar dimensions. In this case the 'dimension constant' can be selected without difficulty, and in fact, it is both expedient and wise to use the symbol '@' throughout the program for all loop limits and subscript calculations. Written this way, the entire calculation can be scaled for a different size array merely by changing the value of a single variable. Programs involving non-conformal arrays cannot be generalized quite so easily, since in this case the value of the dimension constant must be set to satisfy the array with the largest 'row' index.

To summarize the use of TWO-DIMENSIONAL arrays: The variable '@' must first be set to a value equal to (or greater than) the maximum value of the FIRST subscript. This must be done BEFORE any double-subscripted variables are used in the program. Changing the value of the dimension constant later on will alter the internal subscript calculation so that an entirely different variable will be accessed even though the subscripts in the program are unchanged. For reference, the algorithm used to convert the subscripts (I,J) to a single value is: @*I+J-@. From this it is apparent that the variables 'A(1,0)' and 'A(0,1)' are identical to 'A(0)', which is identical to 'A'.

&WRITE

WRITE [list of line or group numbers]

The WRITE command converts selected portions of the program from an internnal form to standard ASCII text. A simple 'W' command will list all of the program. The listing may be stopped and started by typing CTRL/S...CTRL/Q. Output goes to the Current Output Device, which can be changed by using one of the OUTPUT commands. Several sections of the program can be output with a single WRITE command: WRITE 1,2 will list all of Groups 1,2. Blank lines are added between each section to improve the readability. No error occurs if a designated section is missing: what you see is what you've got!

@Examples:

W	List the entire program on the Current Output Device
W 0	Same - an argument of '0' means 'all'
WRITE ALL	List section of the program selected by the variable 'AL'
W-.01	List all lines in Group 0 (can't use 'W 0' - see above)
W 1,-2.5	List all lines in Group 1, and any lines from 2.50-2.99
W 9.1	List only line 9.10
F I=10,31;W I	List all lines in Groups 10-31

&X

X [followed by any text, delimited by a ';' or CR]

The 'X' command is a 'NOP'. It may be used to 'patch out' one or more commands to quickly test variations of a program without having to delete (and then possibly later restore) long command strings. It is unlike a COMMENT command in that the text following the 'X' is scanned, looking for a 'semicolon'. If a ';' is found, the 'X' command is terminated, and the program then continues with the next command.

@Examples

```
FOR I=J=1,10;XET J=I;TYPE I,J       Types (1,1),(2,1),(3,1),...(10,1)
XOR I=J=1,10;SET J=I;TYPE I,J       Types previous value of I, twice
FOR I=J=1,10;SET J=I;XYPE I,J       No output; exits with I=11, J=10
```

&YNCREMENT

Y [list of variable names, +,- signs]

The YNCREMENT command provides a convenient way to INCREMENT or DECREMENT a list of variables: 'Y I' is the same as 'S I=I+1', while 'Y-I' is the same as 'S I=I-1'.

Examples

Y I,J,K	Increment three variables
Y-I,-J,K	Decrement two variables and increment the third

&ZERO

ZERO [numeric argument] or [list of variables]

The ZERO command is used to set ALL, or just a few, of the user's variables to zero. When called without an argument (or with an argument of '0'), the entire 'symbol table' is cleared, thus resetting ALL user-defined variables to zero. (Note that this operation DOES NOT clear any of the 'protected' variables.) To 'zero' only selected variables, the command is used with a list of variable names: 'ZERO I,J' will set those two variables to zero.

The amount of memory set aside for user variable storage is dependent on a number of different hardware and software considerations which are resolved by the initialization routine each time LDF is loaded into memory. In the absence of other complicating factors, each FIELD of memory, beginning with Field 3, has room for 682 variables. On a 32K machine this suggests that there ought to be room for 5*682 or 3410 variables. The actual number is only 3405, due to the space reserved for the protected variables.

Other factors, such as the presence of a bootstrap ROM, space required for BATCH or for routines such as the incremental plotter or the PDP12 display, reduce the maximum size of the symbol table. To provide for alternate uses of this rather large memory space, the ZERO command can dynamically reduce or expand the active size of the symbol table, up to a maximum size set by the initialization code. The format for such commands is: ZERO (N), where (N) is a numeric parameter (or an expression enclosed in parentheses), that indicates the number of FIELDS desired. Values above '5' are illegal, as is a 'Z 2' command on a 16K machine. 'Z 0' is equivalent to 'Z' and merely clears the table without changing its size. To determine the current size of the symbol table you can use this little FPAL function:

```
TYPE FPAL(,1030,7040,2204); C GET SIZE OF SYMBOL TABLE.
```

&FABS

ABSOLUTE-VALUE

FABS(arg) returns the absolute value of the argument.

&FADC

ANALOG-TO-DIGITAL CONVERSION

FADC(channel#) returns the value of the signal connected to the specified channel. Channels are numbered from '0'. The LAB8E has 8 channels (0-7), while the PDP-12 has 16, the first 8 of which are the 'pots' on the front panel. The value returned by the function ranges from -512 to +511 (for a 10-bit ADC). The scale factor depends on the input range selected - it should be checked by measuring a known voltage.

Some versions allow calls of the form 'FADC(channel,#samples)', where the (optional) second argument specifies the number of times to read the A/D converter. The readings are taken as rapidly as possible and the average value is returned by the function. This feature improves the accuracy of the reading by roughly the square-root of the number of samples, e.g. the average of 4 samples of a 10-bit converter gives approximately 12-bit accuracy (if the signal is sufficiently 'noisy'! - 'clean' signals are not improved by this technique, due to the quantized nature of the converter). The number of samples is limited to 2^23/(size of signal).

@Examples

Type FADC()	Check the value on channel '0'
For Ch=0,7;T FADC(Ch)	Check the readings on channels '0-7'
Type FADC(2,1000)	Get the average of 1000 readings on Channel 2

&FAND

LOGICAL-AND

The FAND(arg1,arg2) function returns a 24-bit logical-AND of two arguments. The arguments are adjusted to have equal exponents before being ANDed with each other. The FAND function is useful for extracting data from a packed format, for instance separating the MONTH, DAY and YEAR bits from the system date word, or decoding the information saved in SKED files. (SKED files use a 24-bit data word.) Another common use is with FIN to convert lower-case input to upper-case. This is done by masking with the constant '95'.

@Examples

TYPE FAND(1,3)	Returns the value '1'
TYPE FAND(FDAY(-1),7)	Returns the current year offset
TYPE FAND(FSR(),512)	Checks bit '2' of the switch register
FAND(A,FAND(B,C))	Returns all bits common to A, B, and C
IF(FAND(FIN(),95),eq,'Y)	Branch if the answer is either 'Y' or 'y'

&FATN

ARC-TANGENT

FATN(arg) returns the angle whose tangent is 'arg'. Positive arguments return values in Quadrant I, Negative arguments return values in Quadrant IV. The actual value returned depends on the 'trig mode' selected by the FRDG function. In 'radian mode' (the default), FATN(1) returns 'PI/4', while FATN(-1) returns '-PI/4'. In 'degree mode', these values would be +45 and -45, respectively.

&FBLK

BLOCK NUMBER OF INPUT FILE

FBLK() returns the location of the first block of the current input file. This value may be used for later 'direct access' calls. FBLK has no argument; to get the value for a given file one must first use an 'O I/buffer' or 'I F/buffer' command to select the desired file. Note that selecting buffer/0 will return the starting-block number of the output file.

&FCOS

COSINE FUNCTION

FCOS(arg) finds the cosine of the argument, using the 'trig mode' selected by the FRDG function. The default mode is 'radians', thus FCOS(PI/2)= '0'.

&FCUR

TEKTRONIX CURSOR POSITION

The FCUR() function is part of the 'Tektronix' overlay. It determines the current cursor position, either in 'alpha' or 'Graphics Input' mode, allowing the user to interact with the display. Information obtained by FCUR is returned in two ways: (1) as the value of the function and (2) as the value of two special variables 'X.' and 'Y.' (the '.' stands for 'position').

FCUR() calls with a null or '0' argument turn on the 'GIN mode' cursor and wait for the user to position it with the cursor controls. Hitting any key on the keyboard ends the call and returns the character code of the key as the value of the function. In addition the variables 'X.' and 'Y.' are updated to reflect the new position. ('X.' is the horizontal position, while 'Y.' is the vertical position. The range of values is 0-1023 for X., 0-780 for Y.)

FCUR() calls with a non-zero argument set the variables 'X.' and 'Y.' equal to the current text position and return the value of the terminal's STATUS as the value of the function. (Refer to the terminal User's Guide for the interpretation of status information.) Function calls with a non-zero argument DO NOT enable the 'GIN mode' cursor, nor wait for the user to strike a key.

@Examples

```
TYPE FCUR(),X.,Y.
```

Turns on the cursor, waits for a key to be hit and types the value of the key and the (X,Y) positions

```
UNTIL(Z=FCUR()); VIEW X,Y; VIEW X=X.,Y=Y.,FSGN('N-Z)
```

An endless loop that gets the cursor position, and does one of three things, according to the key hit:

L = draws a LINE from the previous point
N = draws NO line
P = makes a dot at this POINT

&FDAY

SYSTEM DATE FUNCTION

The FDAY() function provides three services: (1) it prints the current system date as an ASCII string; (2) it returns the binary value of the current date; (3) it allows the user to change the current system date.

FDAY calls with a '0' or 'null' argument write the current date as an ASCII string to the Current Output Device. The string has the form 'MM/DD/YY' or alternatively, 'DD.MM.YY' or 'YY.MM.DD' - depending upon whether the user wants 'American', 'European' or 'International' dates. (This is an assembly option, not something that can be easily changed.) The value returned by the function is always '0', hence it can be included as part of a TYPE command by using it in a 'tab' expression.

@Example TYPE "Today is ":FDAY()!

FDAY(-) calls with a Negative argument return the current date word. The date is encoded as follows: the 4 most significant bits are the MONTH, the next 5 bits are the DAY, and the last 3 bits are the YEAR, counting from a BASE YEAR which is either 1970, 1978, 1986 or 1994.

FDAY(+) calls with a Positive argument change the system date and also generate a new date string. Such calls are the usual means for the program to ensure that the files it creates have the proper date. FDAY(+) calls always return the value '0'. The argument should be computed in the following way:

MONTH*256 + DAY*8 + FAND(YEAR-1970,7)

@Examples

SET FDAY(FDAY(FDAY(-1)+8)) Bumps the date by 1 day and then prints it (does not account for end-of-month effects)

```
ASK "MO/DA/YR ",MO,DA,YR :FDAY(MO*256+DA*8+FAND(YR-70,7))
```

This command requests input from the user in the form '3/21/83' and uses this input to set the system date

&FDIN

DIGITAL INPUT

The FDIN() function is part of the LAB8/E overlay. It returns a (masked) value of the Digital Input Register. When called with '0' (or null) argument, FDIN() returns a number equal to ALL BITS set in this register. The interface then resets any bits which were strapped to be EDGE-SENSITIVE. (Level-sensitive bits cannot be reset.)

A MASK can be created to test just certain bits by specifying one or more non-zero arguments. These arguments may be either POSITIVE or NEGATIVE: a positive argument selects a specific bit, while a negative argument selects all bits EXCEPT that one. The bits are numbered from 1-12, instead of from '0-11' so '0' can be used for 'all'. A 'FDIN(1)' call will thus test only the MSB in this register, reseting it if it is edge-sensitive. The value returned is weighted according to the bit position, i.e. if bit '1' was set the value returned by FDIN(1) would be '2048'. Only the bits specified are tested, hence the remaining bits are unchanged by this call.

More complicated masks can be constructed by specifying multiple arguments: FDIN(1,2,3) will test the 3 high-order bits, while FDIN(-1,-2,-3) will test all BUT the three high-order bits. Only the bits which are tested will be reset and then only if (a) they were set prior to the instant when they are tested, and (b) if they are 'edge-sensitive', not 'level-sensitive'.

&FEXP

BASE-e EXPONENTIAL

The FEXP(X) function returns a value for e^X (e=2.71828...). Calling the function with an argument of '1' thus returns an accurate value for 'e'. To avoid arithmetic overflow, the maximum value of the argument should be less than approximately 1400.

&FIN

CHARACTER INPUT

The FIN() function (called with a 'null' argument) reads one character from the Current Input Device and returns its ASCII value (0-127). This function is often called following a JUMP command to determine which key the user typed. It is also used for reading character-strings such as file-names or 'comment' lines.

The following routine may be called to read a string of characters ended by a CR. It provides the usual editing functions (delete and restart) usually associated with keyboard input routines. The 'length' of the string is returned in the 'zeroth' element of the array (C(0)=C):

```
xx.10 Z C;C READ LINE OF TEXT ENDED BY CR; IO RUBOUTS AND RESTARTS
xx.20 IF(127-#=FIN()),.3;IF(#-0U),.1;IF(#-0M),.4;S C(C=C+1)=#;G .2
xx.30 IF(C-C=C-FSGN(C)),.2;T #8#32#8;G .2;CM=0M, CTRLU=0U, DEL=127
xx.40 RETURN; NOTE: THE TEXT IS IN C(1)...C(C), I.E. 'C' IS LENGTH
```

&FIND

CHARACTER SEARCH

The FIND() function reads characters from the Current Input Device, seeking one which matches the argument. If the INPUT ECHO is enabled, each character processed by the FIND() function is passed to the Current Output Device with the exception of the search character. If the Input Echo is turned off all characters read by the FIND() function are effectively discarded.

FIND() normally returns the value of the search character as the value of the function. However, if an End of File mark (CTRL/Z) is found before the search character, FIND() returns the value '0'. Conversely, searching for a 'null' is equivalent to searching for the EOF since 'nulls' are always ignored by the input device.

By appropriate use of the INPUT ECHO, the FIND function can be used to copy some, but not all, of one or more input files to an output file, i.e. to do a MERGE operation. Since FIND never echoes the search character (even when the Input Echo is enabled), one can search for the EOF mark without writing it to the output file. Conversely, to put the search character in the output file, one must use a call such as: FOUT(FIND(xx)).

Common search characters used with the FIND function are: 0J (LF), 0L (FF), 0M (CR) and 0 (EOF). Searching for a LF is usually a good way to copy an entire line (including the preceding CR). Some input devices, however, use a CR to mark the end of a line, not a CR/LF. This is true of the terminal as well as of input from any of the four internal buffers (I B/N).

&FITR

INTEGER PART

The FITR() function returns the InTegeR part of the argument (the name has historical origins). Numbers that cannot be represented as 35-bit integers are not affected by this function call. The SIGN of the argument does not change the result, i.e. FITR(-3.9) is -3. This may be contrasted with the 'entier' function found in some languages which returns the 'integer less than or equal to the argument'. For negative arguments such functions give a result that is one less than that returned by FITR - unless the argument is already an integer, in which case the results are the same!

The user should note that LDF converts many arguments to integer form automatically. Subscripts, values used as indices or placed in hardware registers are always converted internally before being used - hence the use of FITR() in such cases is just a waste of time.

&FLEN

FILE LENGTH

The FLEN function provides a way to determine the length of either an INPUT or OUTPUT file. Calls with '0' or a null argument return the number of unused blocks remaining for the Output file. Calls with an argument of '1' return the size of the Current Input File. (Associate '0' with 'O'utput, '1' with 'I'nput.)

To determine the length of a specific input file one must first select this file as the Current Input File by using the appropriate Open Input or Input File command. Note that with respect to File/0, the value of FLEN() and FLEN(1) are both meaningful: the first returns the number of unused blocks still available for further output while the latter returns the size of the original 'empty' area (which for an output file is its effective 'length').

@Examples

```
O I/0;T FLEN(),FLEN(1)  Type remaining length, empty size
O I/1;T FLEN(1)         Type the length of File/1
F i=,3;O I/i;T FLEN(1)  Type the length of all files
```

&FLGS

EVENT FLAGS

The FLGS() function is part of the PDP12 and LABLE overlays. It monitors event flags which have not (yet) caused a program interrupt. High-priority subroutines can thus use the FLGS function to see if some other action is necessary before they exit. Event Flags are cleared as soon as the program responds to that event; however if the corresponding bit in the 'Event Flag Mask' is cleared, no response will ever occur. Thus the FLGS function can be used to watch for 'non-interrupt' events too.

The Schmitt Triggers provide a good example: it is not always convenient to have S.T. events call an external subroutine. To disable this feature one can use the following commands:

```
SET FQUE(0,-8);FOR I=1,3;SET FQUE(9+I,0,I)
```

This sets the 'Event Flag Mask' to '7770' (octal), thereby disabling events 10,11,12. It then equates S.T.#1 with event 10, S.T.#2 with event 11, etc. These three bits will thus be set whenever one of the Schmitt's detects a signal, but because corresponding bits in the Event Flag Mask are cleared, no 'external DO' call will occur. The FLGS function, however, can be used to monitor the Event Register for any Schmitt Trigger events. (Note: the Event Register is always cleared when the program returns to Command Mode.)

FLGS() may be called with or without an explicit argument. If no argument (or '0') is used the call will return the weighted sum of all pending event flags (but will not reset any of them). If a non-zero argument is used, only bits in the event flag register which match those in the argument will be returned and in that case those same bits will be reset so that the program will see a given event only once. Thus the call FLGS(-1) will return all pending bits (just as a FLGS(0) call does), but in this case the entire register will be cleared. In the example above, to check and reset the bit corresponding to S.T.#1, one would use FLGS(4). To check this bit WITHOUT resetting it, one would use FAND(FLGS(),4).

&FLOG

NATURAL LOGARITHM

The FLOG() function returns the natural (base e) logarithm of the argument. A value of '0' generates an error. Negative arguments, while undefined for the 'real' logarithm function, are accepted by FLOG() and give the same result as positive arguments. This unusual behaviour allows this function to be used for extracting roots of both positive and negative numbers. To convert natural logarithms into 'common' (base 10) logarithms, just divide the result by FLOG(10).

@Examples

```
TYPE FLOG(1)              '0' is the answer to this problem
TYPE FLOG(100)/FLOG(10) '2' is the answer here (the common log of 100)
TYPE FEXP(FLOG(-27)/3)  '3' is the result - but cube root of -27 is -3,
                        which can be corrected by multiplying by FSGN()
```

&FLS

LEFT SWITCH REGISTEI

The FLS() function is peculiar to the PDP-12 version as only the PDP-12 HAS a Left Switch Register. The value is returned as an UNSIGNED number, i.e. a value between 0-4095. There is also of course, a companion FRS() function, which is identical to the usual FSR() function, but with a different name.

&FMAX

MAXIMUM VALUE

The FMAX(arg1,arg2) function returns the argument having the algebraically largest value. Only two arguments can be compared, hence a simple loop is required to sort an entire array for a maximum value.

@Examples

```
TYPE FMAX(1,2)          '2' is the answer
TYPE FMAX(1,-2)         '1' is the answer
SET MAX=N(1); FOR I=2,LAST; SET MAX=FMAX(MAX,N(I))
                        This loop will return the largest value
                        in the array N(1)...N(LAST).
```

&FMIN

MINIMUM VALUE

The FMIN(arg1,arg2) function returns the argument having the algebraically smallest value. Only two arguments can be compared, hence a simple loop is required to sort an entire array for a minimum value.

@Examples

```
TYPE FMIN(1,2)          '1' is the answer
TYPE FMAX(1,-2)         '-2' is the answer
SET MIN=N(1); FOR I=2,LAST; SET MIN=FMIN(MIN,N(I))
                        This loop will return the smallest value
                        in the array N(1)...N(LAST).
```

&FMQ

LOAD MQ REGISTER

The FMQ function is provided for machines with this hardware register which also have a 'front panel' so that the contents of the register can be seen. The least-significant 12 bits of the argument are stored in the MQ register by this function, which provides a very convenient way to display changing values from devices such as A/D converters. Note that LDF tries very hard to preserve the contents of the MQ register even though it is used whenever possible for its intended functions (multiplication and division). Values placed in the MQ by the user will thus generally appear to remain there although the lights may dim a bit during heavy computation.

@Examples

U();S FMQ(FSR())	An endless loop to display the SR in the MQ
F I=1,99;S FMQ(I);DO...	Displays the current loop index in the MQ, allowing the user to monitor progress of a calculation
U();S FMQ(FADC(FSR()))	Displays selected channel of the A/D, allowing interactive adjustment of gain and offset controls

&FOUT

CHARACTER OUTPUT

The FOUT() function is the converse of the FIN() function. It evaluates an arithmetic expression, converting this value to a single character which is sent to the COD. The FOUT() function and the '#' operator in ASK/TYPE commands perform very similar, but not quite identical functions. FOUT() will output a 'null' character if the expression is '0', while the '#' operator generates a FORMFEED in that case; FOUT(13) produces a CR/LF and resets the 'tab counter', while #13 generates only a CR and does not change the tab counter. FOUT also keeps track of printing characters, whereas the '#' operator treats all characters as though they were non-printing.

@Examples

```
SET FOUT(FIND(13))      Searches for a carriage-return, and outputs it
F I=1,80;S FOUT(C(I))   Writes a line of text, even if some chars are '0'
```

&FPAL

MACHINE-CODE ROUTINES

The FPAL function provides the user with an extremely convenient way to put short machine-code functions in his or her program. Such functions allow the direct control of unique hardware features as well as providing access to the 'raw' machine for operations that must be performed as rapidly as possible.

FPAL routines may contain up to 31 machine-code instructions per call. As many separate FPAL routines as desired may be included in the program; each call may pass a 24-bit value to the routine, 12-bits of which automatically appear in the AC. The value of the AC at the end of the routine is then returned to the user as the value of the FPAL function. It is also possible to return 24-bit or larger values to the program, but such programming requires some additional knowledge on the part of the user.

The first argument is always treated as an arithmetic expression. Thus it may contain other functions, loop indices, etc. This is the value passed to the function in the AC. The remaining arguments are treated as octal numbers and are assumed to be valid machine instructions. No checking whatsoever is done on these numbers. The final user-supplied instruction is followed by a 'return' instruction which ends the function call and (normally) returns the value of the AC to the program.

FPAL routines load in Field 1 at 17601 through 17640; the first instruction is thus at 'page+1', the second at 'page+2', etc. The floating accumulator is in locations 44-47 in this field: location 47 always contains '0'; location 46 contains a copy of the AC (lower 12-bits of the first argument) and location 45 contains the upper 12-bits of the first argument. To preserve the contents of the floating accumulator, end the routine with a '5555'; to return the C(AC) as an unsigned number, end with the instruction '5554'.

Since FPAL routines are called with the interrupt system enabled, it is always possible to exit from unexpected loops (for instance loops waiting for a non-existent hardware flag) just by typing CTRL/F. One thus has the very best of both the 'low-level' and 'high-level' worlds at one's fingertips. The primary use for FPAL routines is to control special laboratory equipment. FPAL routines also provide a very convenient way to communicate with a time-sharing system (such as MULTI8) which uses special IOTs to implement system calls.

@Examples

```
FOR Channel_#=0,7; TYPE FPAL(C#,6531,6532,6534,5203,6533)
                   This function reads the LAB8/E A/D converter, using the
                   AC to select the appropriate channel.  It is entirely
                   equivalent to the normal FADC() function except for exe-
                   cution speed.
TYPE FPAL(,6770)   Get the current Time-of-day from the MULTI8 monitor
```

&FQUE

MULTI-TASKING OPERATION:

The FQUE function serves many purposes, and is therefore somewhat complicated. It may have 1 to 4 arguments which fit the following general description:

Argument 1:	The TASK NUMBER
Argument 2:	The associated SUBROUTINE CALL
Argument 3:	The TIMER or SCHMITT TRIGGER number
Argument 4:	The time period or interval

The major purpose of the FQUE function is to implement the idea of EXTERNAL subroutine calls. Such calls are in most respects similar to an ordinary DO call, however they are executed independently of the 'main' program (though synchronized with it) and do not 'return' to any 'calling point'. When an external subroutine call is completed it just vanishes from the scene, having served some purpose in the meantime.

Up to 12 such calls may be active at any one time. Each call has an associated PRIORITY value which is used to determine the order in which it will be executed. This priority value is, in fact, just the TASK NUMBER assigned by the FQUE function (Argument #1 in the list above). Tasks are numbered 1 to 12, with #1 having the highest priority and #12 the lowest. Lower still than #12 is the MAIN program, which will always be temporarily suspended in favor of servicing one of the External Subroutine Calls.

EXTERNAL subroutine calls can occur from at least 4 sources (more may be added by the user if her or she so desires): (1) the MAIN program can issue an external call. Thus in addition to the normal DO call, the main program can call a subroutine which then runs IN PARALLEL with it, returning not to the calling point when it is done, but simply disappearing. This method is convenient for testing routines that are to be called by some other means later on. It also allows tasks to be split into a 'high-priority' part and a 'low-priority' part, with the 'high-priority' part merely requesting that the 'low-priority' part be executed at a later time, instead of waiting for it to be done immediately at the expense of other 'high-priority' tasks.

(2) Secondly, an external subroutine can be called by any one of 4 internal TIMERS. These timers can be programmed to issue subroutine calls as often as 10 times per second, or as infrequently as once every three weeks. Furthermore, such calls can be either repetitive, or one-time events which occur after a prescribed delay time.

(3) Thirdly, the PDP12 and LAB8E versions allow the Schmitt Triggers to request external calls. Thus an arbitrary electrical signal can be connected directly to a specific high-level subroutine.

(4) And lastly there are the KEYPAD calls. These are also 'external' calls however they are implemented independently of the FQUE routines and have a priority just above that of the main program. Their purpose is basically convenience, not real-time programming.

With this outline in hand we will now step into the details of programming the FQUE function. It is important to note at the beginning that external DO calls do not use the subroutine number itself, but rather the TASK number associated with that call. An external call is started by placing its task number in an EVENT REGISTER (see the discussion of the FLGS function). This alerts the scheduler to the fact that there is something to do besides running the main program. The task with the highest priority at any given

decision point is then allowed to run, and it is at this time that the subroutine call itself is extracted from the 'task table'.

One of the primary tasks of the FQUE function is thus to create the 'task table' which equates TASK numbers with the corresponding SUBROUTINE calls. Entries in this table are made by FQUE calls with TWO ARGUMENTS: the first is the task number, and the second is the associated DO call. In order to simplify the table, a small restriction is placed on such calls: only single lines and single groups are allowed. Thus calls to execute the entire program or to execute just part of a group cannot be stored in the table. If such calls are necessary, they can be included in the code executed by the external call.

@Examples

SET FQUE(1,12)	Identify TASK 1 as a 'DO 12' call
SET FQUE(2,1.2)	Identify TASK 2 as a 'DO 1.2' call
SET FQUE(3,.2)	Identify TASK 3 as a 'DO xx.2' call
SET FQUE(4,-5.6)	Attempt to identify TASK 4 as a 'DO-5.6' call. The 'sign' is ignored, turning TASK 4 into a 'DO 5.6' call WITHOUT signalling an error.
SET FQUE(5,0)	Attempt to identify TASK 5 as a 'DO 0' (entire program) call. This is likewise not allowed, with the effect that TASK 5 becomes a 'NOP'.

From the last example we see that the way to remove a task assignment from the table is to assign it a '0' subroutine call. This is an important feature because it allows tasks to 'shut themselves off' by putting a '0' in the table. On a more global scale, calling the FQUE function with a single NEGATIVE argument will clear the entire task table - thus disabling in one command ALL external subroutine calls. A 'SET FQUE(-1)' call at the beginning of a program thus 'cleans the slate'.

This brings up another issue: entries are made in the task table under the assumption that such routines actually exist in the program currently in memory. What happens if this program executes a 'Lib DO' or 'Lib GO' call so that a completely different program is brought into memory? Clearly the subroutine calls stored in the task table are no longer appropriate, hence external calls are automatically disabled whenever the contents of the program buffer are changed.

This can occur as a result of a program swap, or just because the main program reaches a stopping point and returns to Command Mode. It has not been explictly stated so far, but in fact, external calls are only allowed while the main program is running. The reason for this is that it is almost impossible to perform direct commands such as editing, listing programs, etc. if external calls can occur at any time.

So there is an internal mechanism for 'turning off' FQUE calls. There is also an explicit way to do this in the program as well: calling the FQUE function with a 'null' or '0' argument temporarily disables all external calls. Clearly this is a useful thing to be able to do, but how does one enable external calls initially, or re-enable them after a 'FQUE()' call? The answer is that ANY FQUE call with a POSITIVE, non-ZERO task number enables the scheduler. Thus a call which sets up an entry in the task table also enables external calls.

Once an entry has been made in the task table, that subroutine will be run whenever the corresponding EVENT FLAG is set. And the simplest way to set this flag is to again use the FQUE function, but this time with a SINGLE, POSITIVE, non-ZERO argument. The call 'SET FQUE(1)' will set the flag for

task #1 and PRESTO! it will be called as soon as the scheduler can arrange for it. Notice that such calls also satisfy the conditions for ENABLING external subroutines; thus even if the scheduler were previously turned off (either by an explicit FQUE(0) call, or because the main program returned to Command Mode), setting the Event Flag with a FQUE call will ensure that the task will be executed.

While this suggests the idea of actually initiating an external call simply to enable the scheduler, one would hope there was a better way. Of course, one could simply set the flag of an undefined task; this turns on external calls but is otherwise a 'nop'. A somewhat better way is to set flag '13'. Since 'task 13' does not exist the only effect is to enable external calls.

The discussion thus far covers all the essential details needed for calling the FQUE function with either 1 or 2 arguments. We now consider calls with 3 or 4 arguments, examining three-argument calls first. Only versions with the FLGS function (PDP12, LAB8E, and others) recognize such calls.

To allow Schmitt events to create an External Subroutine Call, we must tell LDF which Schmitt goes with which Call. This is accomplished using a FQUE call with 3 arguments, the first two of which store information in the task table as discussed above, and the third of which identifies the appropriate Schmitt. Thus a 'SET FQUE(1,2,3)' call provides the following information:

The TASK NUMBER is 1 (highest priority)
The SUBROUTINE is Group 2
The Schmitt Trigger is #3

Once a Schmitt Trigger has been associated with ANY task, signals detected by that trigger will cause that task to be scheduled. The only way to DISABLE Schmitt events is to re-define the task as the 'null' task by using '0' as the second argument, or else to clear the corresponding bits in the Event Flag Mask. (See the FLGS section for more information on this.)

Finally we come to FQUE calls with FOUR arguments. Such calls are reserved for setting up the 4 internal timers that can be used to initiate External Subroutine Calls; these timers are completely independent of the timer used by the FTIM function, although all timers are ultimately driven by the same source of periodic interrupts and are thus synchronized with each other to some extent. The FQUE timers are decremented 10 times per second and whenever they 'count down' to zero, they can be enabled to set one of the EVENT FLAGS. Thus up to 4 event flags can be set from timed interrupts, with a frequency of up to 10 'events' per second. The FQUE call which makes this all happen looks like:

FQUE(task, subroutine, timer#, time interval)

where the first 3 arguments should look familiar - the timer number simply replacing the Schmitt Trigger number discussed above. Timers are numbered '0-3', whereas the Schmitt Triggers are numbered 1-3. The fourth parameter is then the number of SECONDS that the timer should wait before setting the event flag. This number can be specified to the nearest 0.1 second. The call: SET FQUE(1,2,3,4.5) has the following meaning:

The TASK number is 1
The SUBROUTINE is Group 2
The TIMER is #3
The time INTERVAL is 4.5 seconds

Time intervals may be either POSITIVE or NEGATIVE numbers; Positive values

are interpreted as REPETITIVE intervals, i.e. once the specified interval has elapsed, the counter is automatically reset so that the event flag is set over and over again. Negative intervals are interpreted as 'one-time' events: after the first time interval has elapsed the counter continues to run without being reset, and thus causes no further interrupts for 2^24/10 seconds (approximatey 19.5 days).

Thus it is very, very simple to set up a periodic interrupt which executes a subroutine once every minute: SET FQUE(1,2,3,60). Since this call also satisfies the requirments for enabling the scheduler, nothing further needs to be done except to wait for the time period to elapse.

There are some other considerations, which we will now discuss. The first is: how fast can external subroutine calls really be executed? Since such calls are no different (once they are scheduled) from ordinary DO calls, the time scale is clearly MILLIseconds, not MICROseconds. In fact, modest subroutines might even require TENTHS of seconds to execute - which is the reason that the timers have only 0.1 second resolution. There is no way to limit the rate of Schmitt Trigger events, of course, but it would obviously behoove the user to think in terms of 10-20 interrupts per second as an upper limit.

What happens if external calls are requested faster than they can be executed? Answer: some calls are simply missed. The EVENT FLAG mechanism is a rather primative one - there is no queue of requests, only a single bit, so if a second interrupt comes along before the first one has been serviced it will simply set the same bit again. External calls are thus best suited to applications with modest interrupt rates (say 10/second), which occur for a reasonably long period of time (hours to a few weeks).

In order to allow the program some 'fine control' over the scheduling process, a call has been provided for temporarily disabling some of the event flags WITHOUT changing the task table. This method involves modifying the EVENT MASK which is used by the scheduler when it examines the EVENT FLAGS. If this mask is set to 7777(8) then all tasks will eventually be scheduled. If some of the bits in the mask are '0', however, those event flags will never be seen by the scheduler, hence those tasks will never be executed, and since the flag bits are only reset when the task is actually run, those bits will remain set until the program returns to command mode or the flag register is read (and reset) by the FLGS function.

The method for changing the EVENT MASK is to call the FQUE function with 2 arguments, the first of which is '0'. You will recall that calls with '0' as the first argument DISABLE the scheduler, hence it is convenient to add an option to only partially disable things. The second argument in such a call is the desired event mask, with '-1' enabling all events, and '0' disabling them all. To disable 'task 1', use FQUE(,-2048-1), etc.

Here is a summary of all the different kinds of FQUE calls:

FQUE(-1)	Clears the entire task table
FQUE(0)	Inhibits all external calls
FQUE(1) thru FQUE(12)	Sets the Event Flag for that task
FQUE(13)	Enables external calls
FQUE(0,mask)	Changes the Event Flag Mask
FQUE(1,0) to FQUE(12,0)	Disables a particular task
FQUE(1-12,subroutine)	Defines one of the twelve tasks
FQUE(1-12,subr,Schmitt)	Defines a Schmitt Trigger call
FQUE(N,subr,timer,time)	Defines a timed-event call

Synchronization: this subject has been avoided until now. The question is: how flexible is the scheduler? If an interrupt occurs which sets the Event Flag for Task #1, just how soon will this task be executed? In an ideal machine the answer would be VERY SOON, and at the hardware level this ideal is rather closely approximated. Even at that level, however, it is necessary for the machine to finish the CURRENT INSTRUCTION before the interrupt is acknowledged.

The basic reason for SYNCHRONIZATION is to avoid saving thousands of pieces of information when perhaps only a short time later it would be sufficient to save half a dozen. If interrupts could occur at totally arbitrary times then all 'unfinished business' at that point would have to be saved so that the state of the machine could be restored at the end of the interrupt. For hardware interrupts this would require knowledge of the state of all flip-flops, etc. For the sort of 'high-level' interrupts implemented by the FQUE function, allowing external calls to occur at any time would require that the state of all internal routines be saved, each of which has a return address and a number of parameters. Thus hundreds of memory locations would have to be saved in this case, when, if one simply waits until the program comes to the beginning of a NEW LINE, the number of things that must be remembered is VASTLY reduced. In fact, switching to an external task at that point is exceedingly simple, amounting to little more than substituting one new line number for another and remembering the fact that we have been interrupted and need to eventually get back to what we were about to do.

Thus the need for simplicity comes in conflict with the desire for 'instant response', leading to the result that External Subroutine Calls are only recognized when the program is about to start a new line because of a GOTO, a DO call or simply 'falling into' the next sequential line of the program. Another way to state this is that the CURRENT LINE always has the highest priority, even if it is associated with the lowest priority task (i.e. the MAIN program). If the main program were waiting for the user to respond to an ASK command, for example, no external subroutine calls could occur, even though a high-priority event flag was set by one of the Schmitt triggers.

There is one other moment when it is realtively easy to respond to external DO calls, and that is during a HESITATE command. Thus while the program is doing little more than counting clock ticks, it is possible to also acknowledge external subroutine requests. These are the only two places where it is easy, but as a practical matter, most programs come to the end of a command line rather often, hence the time required to respond to an external call is typically on the order of 1/50 sec. The only particularly troublesome case is that of long 'FOR loops' which are written all on one line. But by being aware of potential problems like this the user can easily make corrective changes to his or her program to avoid long 'dead' periods. For example, simply ending the FOR loop with a 'Comment' command will allow the loop to be interrupted. ('Comments' are treated like a 'new line'.)

@Examples

```
1.1 SET FQUE(-1),FQUE(1,.3,1,1)
1.2 JUMP .2;C HIT A KEY TO QUIT
1.3 TYPE "Tick ";SET FQUE(1,.4)
1.4 TYPE "tock"!;SET FQUE(1,.3)

2.1 SET FQUE(-1),FQUE(5,.3,1)
2.2 U();C TYPE CTRL/F TO STOP
2.3 TYPE "SCHMITT #1 FIRED!"!

3.1 SET FQUE(-1),FQUE(1,.3,1,5)
3.2 TYPE FTIM();HESITATE 1;G .2
```

3.3 QUIT; STOPS AFTER 5 SECONDS

Notice in each case that the 'main' program is little more than an endless loop - this is typical of programs using external subroutine calls. Everything really happens via interrupts!

&FRA

RANDOM ACCESS

The FRA function provides Random Access data storage using any of the four possible files. Four data modes are provided: unsigned integers, signed integers, double-precision integers and floating-point, which is the default. Random Access files may be as large as the largest system device, hence it is possible to randomly access over a million different 12-bit values per file, or the same number of floating-point values split into four different files.

The FRA() function is easy to use: to retrieve the value of the 'Nth' data point in a file takes only two commands: OPEN INPUT filename; TYPE FRA(N). Calls with ONE argument return the data value stored at that file location, while calls with TWO arguments store the value of the second argument at the specified location. FRA() works just like the FCOM function, but with data stored in a file rather than just in memory.

FRA always uses the Current Input File (CIF). This is independent of, and to be distinguished from, the Current Input Device (CID). To change the CIF without changing the CID, one uses the OPEN INPUT command, whereas one uses an INPUT FILE command (or some other INPUT command) to change the CID. The OPEN INPUT command may be used with or without a file name. Initially it is used WITH a file name in order to locate the desired file. If only one file were necessary, this would be the only OPEN INPUT command in the program. If several files were required, each one must be OPENed with a different '/buffer' specification.

@Examples

```
Open Input file1          The CIF becomes 'FILE1.DA'
O I/1 file1               Same, but with an explicit /buffer specification
O I/2 file2               Selects 'FILE2.DA' as the CIF, using buffer/2
```

If these examples were actually carried out, we would have at this point 2 different files 'attached' to the program. The CIF is 'file/2' since it was the last one selected. Values obtained by the FRA function would thus come from FILE2.DA. To select 'file/1', one would use an 'O I/1' command. Notice that NO FILENAME is required: filenames are only necessary when new files are 'looked up', hence there is no 'system overhead' associated with OPEN INPUT commands that merely specify a new buffer. The only effect of such commands is to change a few memory locations so that the FRA(), FBLK() and FLEN() functions know which file is the Current Input File.

@Example O I/1; TYPE FRA(100); O I/2; TYPE FRA(100)

Two values will be typed out: one from 'file/1' and one from 'file/2'; this idea can obviously be extended to all four files (/0,/1,/2,/3) if need be.

DATA MODES: FRA can access data in any of 4 different 'modes'. The default mode is 'floating-point', which preserves all significant bits and requires the least conversion time between file storage and memory storage. In this mode there are 64 values per block. Other modes pack 2 to 4 times as much data in the same file space at the expense of a smaller range of values. In double-precision mode, for example, there are 128 values per block, with a data range of +/-8388607; in single-precision mode there are 256 values per block with a total range of 4096.

The data mode is changed by a call such as 'SET FRA(-1,2)' where the first argument is Negative (clearly not a 'legal' index), and the second argument specifes the desired mode (0,1,2 or 4):

FRA(-1,0)	UNSIGNED integer mode (0 to 4095)
FRA(-1,1)	SIGNED integer mode (-2048 to +2047)
FRA(-1,2)	D.P. integer mode (+/-8388607)
FRA(-1,4)	Floating Point mode (+/-1E615)

Once changed, the mode setting applies to all subsequent FRA() calls, hence if one is using files with different storage modes one must change the mode each time one changes the CIF. Obviously it would be nice if the 'mode setting' were remembered along with the other file information, but it is not.

STORING DATA: as mentioned above, to STORE data in a file one simply calls the FRA() function with two arguments. However, you may be wondering just how this works, since it has also been emphasized that FRA() uses the four INPUT files, while 'storing' data seems to imply some sort of OUTPUT operation. The answer is that the FRA() routines can both READ & WRITE a given block of data, while the INPUT routines can only READ it. Much of the code is the same however, which is why the FRA function and the OPEN INPUT command are so intimately connected.

The other question that usually arises at this point is: how do I initially open a file to put data in? All the discussion so far has been in terms of files that ALREADY EXIST, whereas when I want to STORE data it seems to me that I need a NEW file, don't I?

The answers here are slightly more complicated, and involve the design of the operating system to some extent. In this system there can be only ONE 'output' file per device at any one time. This is because files are stored in consecutive blocks, hence until a file is 'closed' it is not possible to know where to start a second one. The chief difference between an 'input' file and an 'output' file is thus that one has a known, fixed size, whereas the other has only a possible maximum size.

When using the FRA function to store experimental data one has two choices: one can create a 'dummy' input file with a fixed length that can be safely predicted to be large enough for all the data, or one can use the OUTPUT file (treated as Input File/0), if one needs to store an unknown amount of data, with a limit set only by the maximum number of blocks available. If one uses 'method A' (fixed-length input file) then one can use the normal output file for other purposes, whereas if one uses 'method B' this option is not available, and in general one must be a bit more careful while the file is 'open' so that nothing happens to accidentally destroy the data you have put there.

Method A: Suppose the experiment generates 256 data points every 10 minutes or so. If you plan to work all day on this experiment, and the data can be adequately represented as double-precision integers, then you would need a 100-block file in which to store the data. This file can be created by the following two commands:

```
Open Output DATA01.DP; Outpu  Abort 100
```

This file can now be used as FRA File/3 by saying 'Open Input/3 DATA01.DP'. Note that it is customary to give Random Access files a different extension from ordinary 'data' files so that one does not attempt to use them for the wrong purpose. '.DP' is common for 'Double Precision' files, 'UN' for 'Unsigned' files, etc. A 'List Only *.DP' command can then be used to list all the double-precision files.

The disadvantage of 'Method A' is that if you have to stop after collecting 10 sets of data, you will have a file with a lot of 'wasted space' in it.

There is no 'anxiety-free' way to solve this problem, but the following may be used:

```
Lib ERASE DATA01.DP; 0 0 DATA01.DP; 0 A 20
```

Thus one simply deletes the file and then creates a new one with a shorter length. The problem here is GUARANTEEING that the 'new' DATA01.DP file will really be just the first 20 blocks of the 'old' DATA01.DP file. IF (and I repeat), IF no other files have been created on this device in the meantime then everything will be OK. If, however, other files have been added (or deleted), then the 'allocation algorithm' may well decide to put the 'new' DATA01.DP file in a different place. You can, of course, watch for this by comparing the value of the FBLK() function before and after (or just by doing a 'List All,E'), and you can also eliminate the problem almost entirely by using the 'square bracket' option in the OPEN OUTPUT command.

Method B: The problems above can be eliminated by using Method B. In this case you separate the two 'file creation' commands, opening the output file at the beginning of the experiment, and leaving it open all day long. Then at the end of the day when you know exactly how many data sets were collected, you do the 'Output Abort' (NEVER NEVER use 'Output Close' with a FRA() file!), specifying just the right length. These steps are shown below:

```
morning:        Open Output DATA01.DP; 0 I/O
all day:        SET FRA(N,data), N=N+1
afternoon:      Output Abort (N/128+1)
```

The problem here is that an 'open' file is unprotected from system crashes; if something happens during the day there will be no record in the directoy that this area of the disk has valuable information on it. An even worse feature is that you cannot make any program changes without the certainty of destroying your file, for if you thoughtlessly decide to do a 'Lib SAVE' of the new version, your output file will be CLOSED (not ABORTED!), with a length of 1 block which will contain all zeros!

This discussion is not meant to deter anyone from using FRA files, but only to indicate the problems that arise when one attempts to make the size of a FRA file 'just right'. If data is stored on a large disk (RK05, RL01, etc.) then there can be problems in attempting to adjust the file size. On the other hand, in this case 'wasting' a few blocks now and then is not so much of a disaster. If data is stored on a DECtape or floppy disk, then it is worthwhile to make the files only as long as necessary. In this case, however the directory structure is so much simpler that problems with the file 'going in the wrong place' are practically non-existent.

One final point with regard to storing data with the FRA function: each new data value is obviously NOT immediately written out, but rather put in the designated buffer until the entire block of data has been filled. At any time, however, one can FORCE this data to be written out by using a FRA(-1) call. This is somewhat akin to an 'Output Dump' command; in practice it is only necessary to do this just before exiting from the program, since all other file reading and writing operations are handled automatically. Naturally if you have been storing data in more than one FRA file, you should be sure that the last block of each file has been written out.

To summarize the possible FRA calls:

```
FRA(I)          Reads 'Ith' data point of the CIF
FRA(I,data)     Stores 'Ith' data value in CIF
FRA(-1,mode)    Changes data mode (mode=0,1,2,4 only)
```

FRA(-1) Writes contents of data buffer into CIF

FRA() calls are fully recursive, hence FRA(I,FRA(J)) can be used to shuffle data around. This can become quite inefficient, however, hence it is much better to copy a block of data into the FCOM area and then write it out all at once so that different data blocks are not swapped in and out of memory on every FRA call.

One final point: the FRA() function can be used WITHOUT a file! This allows any of the four buffers to be used for temporary storage in addition to the FCOM area. To use FRA in this manner the desired buffer should be selected by an INPUT CLOSE/buffer command, instead of by an OPEN INPUT command. The 'I C' command ensures that data in the buffer will never be 'accidentally' written into a file.

@Examples:

```
I C/3;S FRA(-1,0);F I=0,255;S FRA(I,-I)
```
This stores 256 integers in buffer/3 as the values '0', 4095, 4094, etc.

```
I C/2;S FRA(-1,4);F I=0,63;S FRA(I,FSQT(I))
```
This stores the square root of the first 63 integers in buffer/2

```
O B/1;T "This is a char string";O T;I C/1;S FRA(-1,1);F I=1,25;T FRA(I)!
```
This uses the FRA() function to analyze a character string

&FRAC

FRACTIONAL PART

The FRAC(arg) function returns the 'fractional' part of the argument - the part that FITR discards. Since this involves subtracting two nearly equal values from each other, the result may have only limited precision! In general the value returned by FRAC() should NEVER be tested for an 'exact' result, but only for values within a certain tolerance.

@Examples

```
TYPE FRAC(PI)              Answer:   0.141592654
FOR I=10,19;T FITR(10*FRAC(I/10))
                           Answer:   0,0,1,3,3,5,6,7,8,9
FOR I=10,19;T FITR(10.1*FRAC(I/10))
                           Anser:    0,1,2,3,4,5,6,7,8,9
```

&FRAN

RANDOM NUMBERS

The FRAN() function returns an arbitrary random number from a pseudo-random sequence. The sequence can be modified by calling FRAN() with an argument, however this is not normally necessary because the initial random number is set by the keyboard input routine and hence is almost always different each time LDF is loaded into memory (4095 possibilities). The range of values returned is 0-1 (exclusive); other ranges may be created by simple scaling. No statistical bias has been found, nor is there any obvious pair-wise correlation between values. This is not to imply that FRAN() is statistically 'perfect', however, and a report of any deficiencies would be appreciated.

@Examples

```
For N=R=0,99;Set R=R+FRAN();Next; Type R/N
                Check the average of 100 calls
For i=pi=0,399;If(fran()^2+fran()^2,le,1).*pi=pi+1;Next;Type pi/100
                Compute approximate value of PI (to within +/- 0.2)
```

&FRDG

ANGULAR MEASURE

The FRDG() function allows the programmer to choose a completely arbitrary system of angular measure for use by the three 'trig' functions (FSIN,FCOS FATN). The common choices are Radians, Degrees and Grads (hence the name 'FRDG'), with the default being RADIANS (for consistency with other implementations of FOCAL). The desired set of units is selected by a call of the form:

SET FRDG(units in a quadrant)

where the argument is equal to the number of units in a 'quadrant'. This value is also returned by the function and should be saved in one of the Protected Variables so that it can be easily referred to by other routines which need to know the current system of units. This value can be found (approximately) from the expression 2*FATN(1).

@Examples

SET FRDG(PI/2)	Sets Radian Measure
SET FRDG(90)	Sets Degree Measure
SET FRDG(100)	Sets Grad Measure
SET FRDG(1/4)	Sets Revolution Measure

Accuracy: The accuracy of the trig functions is obviously directly related to the accuracy of the argument in the FRDG function. The typical accuracy with the arguments shown is a few parts in 1E10, with 'degrees' and 'grads' tending to be slightly more accurate than 'radians' due to the use of integer arguments and the elimination of scaling in FSIN/FCOS calls.

&FSGN

SIGN VALUE

The FSGN(arg) function returns the value '-1','0' or '+1', depending on the sign of the argument. An expression that returns '+1' for both '0' and '+' numbers is: '(arg,LT,0)'. This relational expression has the value '-1' if the argument is negative, and '+1' ('false') if it is zero or positive.

@Examples

TYPE FSGN(-PI)	Answer: -1	TYPE (-PI,LT,)	Answer: -1
TYPE FSGN(PI-PI)	Answer: 0	TYPE (PI-PI,LT,)	Answer: +1
TYPE FSGN(PI)	Answer: +1	TYPE (PI,LT,)	Answer: +1

&FSIN

SINE FUNCTION

FSIN(arg) returns the cosine of the argument usi:g the 'trig mode' selected by the FRDG function. The default mode is 'radi:ns', thus FSIN(PI/2)= '1'.

&FSQT

SQUARE ROOT

The FSQT() function returns the Square Root of the argument (positive arguments only). Accuracy is +/-1 in the 10th place.

@Examples

```
TYPE FSQT(2)              Answer:  1.414213562
TYPE(2,eq,FSQT(2)^2)      Answer: -1 (true)
```

&FSR
&FRS

SWITCH REGISTER
RIGHT SWITCHES

The FSR() function returns the current value of the console Switch Register as a SIGNED number (-2048 to +2047). For symmetry with the FLS() function, the PDP12 version uses the name FRS() for this same function. The state of SW0 (the Most Significant Bit) can be tested by checking the SIGN while the FAND() function can be used to test other individual bits.

@Examples

```
U(FRS());S FMQ(FADC())  Display A/D readings in the MQ until SW0 is raised
T FAND(FSR(),120)/120   Type the value of bits 5-8 taken as a 4-bit entity
```

&FSS

SENSE SWITCHES

The FSS(N) function returns the value of the Nth Sense Switch on the PDP12, where 'N' is 0-5 (any other values result in an error message). The value returned by the function is '-1' if the switch is OPEN, and '+1' if it is CLOSED. (These are just the reverse of 'truth values' assigned to logical expressions, hence conditional commands effectively test 'not switch xx'.)

@Examples

```
FOR I=0,5;T %-1,FSS(I)  Output status of all 6 Sense Switches
UNTIL(FSS());T #7       Ring the bell until switch '0' is reset
IF(FSS(1)).1            Branch to line xx.10 if switch 1 is OFF
```

&FTIM

ELAPSED TIME

The FTIM() function keeps a running counter which is incremented 100 times per second. Thus successive calls to this function can be used to measure the total elapsed time to the nearest 0.01sec. A 24-bit counter is used, which, if left undisturbed, will overflow approximately every 23.5 hours. The counter can be reset to '0' at any time, or preset to any integer number of seconds. The FTIM() function is particularly useful for measuring execution time of critical program steps since it can be started (reset) under program control. Here is a summary of FTIM() calls:

FTIM(-1)	Resets counter to '0'
FTIM()	Returns counter value divided by 100
FTIM(+N)	Presets counter to 'N' seconds

@Examples

```
TYPE "HIT A KEY":FTIM(-1):-1,FTIM()
                        Measure reaction time
FOR I=FTIM(-1),999;NEXT;TYPE FTIM()
                        Measure how long it takes to do 1000 loops
                        (Note: FTIM(-1) returns the value '0'.)
```

&FTRM

ASK TERMINATOR

The FTRM function returns the character code of the LAST TERMINATOR found by an ASK command. This information may be used to create an 'indefinite' input loop, which continues to request data until a special terminator is found that indicates 'The End'. Since ASK commands can be terminated by almost anything EXCEPT the digits '0-9' and the letters 'A-Z' (or 'a-z'), imaginative programmers have many possibilities to choose from. A few suggestions: any arithmetic operator may be used, or any special symbol (such as a '$' or '!'). A second decimal point also acts as a terminator, and of course, all control characters do too. In particular, one may check for an End-of-File condition by testing for CTRL/Z (code 26).

It is helpful when writing such input loops to know that all input devices return an endless number of CTRL/Zs once they have reached the EOF. Thus ASK commands which input more than one value at a time will not get 'stuck' if the file ends in the middle of the command: each successive CTRL/Z will terminate data input (leaving the value '0') until the command is finally satisfied, at which point the program can check to see if the FTRM function has the value '26'. Checking for the last value in this way is much more convenient than using a supposedly 'false' data point (such as the all-too-common '999' or '-1' tests).

@Example

```
ZERO I;UNTIL(FTRM(),eq,'$);ASK D(I=I+1)
```

Reads in successive 'D' values until ended by a '$' (A problem here is that if FTRM() is initially equal to a '$' this loop will exit immediately - there is no way to preset the terminator except by ASKing for something.)

&FVB

VIEW BUFFER

The FVB() function appears in the LAB8E and PDP12 versions. It provides a way to access or change data stored by the VIEW command. FVB() calls are similar to FCOM or FRA calls: one argument means 'read from this location', while two arguments mean 'store new value in this location'. Locations are numbered from '0', with all even locations holding 'X' (horizontal) values, and all odd locations holding 'Y' (vertical) values. Playing with the data in the view buffer thus allows the display to be changed dynamically.

@Examples

```
TYPE FVB(),FVB(2)       Check the first two 'X' values
TYPE FVB(1),FVB(3)      Check the first two 'Y' values
VIEW FVB(),FVB(1)       Double-intensify the first point
```

&FXL

EXTERNAL LEVELS

This is a PDP12 function which returns the status of a specific 'External Level' or 'sense line'. The argument must be a value between 0-11, and the result is '-1' if the level is in one state and '+1' if it is in the other. FXL() is similar to the FSS() function.

@Example

```
FOR I=0,11;T %1,FXL(I)          Type status of all 12 sense lines
```

UNTIL...NEXT Bug

There is a subtle interaction between the UNTIL and NEXT commands in LAB-FOCAL (V5D), which makes the following restriction necessary:

> UNTIL...NEXT loops will not work correctly unless both of the commands are terminated by the same character.

This means that if the NEXT command is followed by a semicolon, the UNTIL command must also be followed by a semicolon (this is the normal situation and thus requires no special effort). However, if the NEXT command is followed by a SPACE (because, for instance, it has a line-number argument), then the UNTIL command must also be followed by a space. The examples below show the result of different combinations of spaces and semicolons:

(1) Good example (semicolon, semicolon): ↓ ↓

```
1.1 For J=1,3; Type I=J; Until(I,eq,4); Type I=I+1; Next; Type !

1.000   2.000   3.000   4.000
2.000   3.000   4.000
3.000   4.000
```

Comment: this is the 'normal' method of programming UNTIL..NEXT loops.

(2) Bad example (space, semicolon): ↓ ↓

```
1.1 For J=1,3; Type I=J; Until(I,eq,4) ; Type I=I+1; Next; Type !

1.000   2.000   3.000   4.000   2.000   3.000   4.000   3.000   4.000
```

Comment: The space following the UNTIL command causes the NEXT command to be ignored (hence no CRLFs). Few programmers would use such syntax.

(3) Bad example (semicolon, space): ↓ ↓

```
1.1 For J=1,3; Type I=J; Until(I,eq,4); Type I=I+1; Next .2
1.2 Type !

1.000   2.000   3.000   4.000 ?03.30 @ 01.10 UNKNOWN COMMAND
```

Comment: This error occurs when a NEXT command is followed by a space, while the UNTIL command is followed by a semicolon. This is the 'normal' style of writing such commands: it looks OK, but it doesn't work!

(4) Good example (space, space): ↓ ↓

```
1.1 For J=1,3; Type I=J; Until(I,eq,4) ; Type I=I+1; Next .2
1.2 Type !

1.000   2.000   3.000   4.000
2.000   3.000   4.000
3.000   4.000
```

Comment: putting a space at the end of the UNTIL command fixes the problem. It is not feasible to supply a patch for the interpreter, hence this small imperfection must remain as one of LDF's special features!!