```
UWFUWFU                UWFUWFU
WFUWFUW                WFUWFUW
FUWFUWF                FUWFUWF
UWFUWFU                UWFUWFU
WFUWFUW                WFUWFUW
FUWFUWF                FUWFUWF
UWFUWFU                UWFUWFU
WFUWFUW                WFUWFUW
FUWFUWF                FUWFUWF
UWFUWFU                UWFUWFU
WFUWFUW                WFUWFUW
FUWFUWF                FUWFUWF
UWFUWFU                UWFUWFU
WFUWFUW                WFUWFUW
FUWFUWFW              WFUWFUWF
UWFUWFUWFUWFUWFUWFUWFUWFU
 FUWFUWFUWFUWFUWFUWFUWFU
  WFUWFUWFUWFUWFUWFUWFU
      UWFUWFUWFUWFU
```

```
UWFUWFU              UWFUWFU
WFUWFUW              WFUWFUW
FUWFUWF              FUWFUWF
UWFUWFU              UWFUWFU
WFUWFUW              WFUWFUW
FUWFUWF              FUWFUWF
UWFUWFU       U      UWFUWFU
WFUWFUW     UWF      WFUWFUW
FUWFUWF    UWFUF     FUWFUWF
UWFUWFU   WWFUWFU    UWFUWFU
WFUWFUW UWFUWFUWF  WFUWFUW
FUWFUWFUWFUWFUWFUWFUWFUWF
UWFUWFUWFUWFUWFUWFUWFUWFU
WFUWFUWFUWFU FUWFUWFUWFUW
FUWFUWFUWFU   WFUWFUWFUWF
UWFUWFUWFU     UWFUWFUWFU
WFUWFUWFU        FUWFUWFUW
FUWFUWFU          WFUWFUWF
UWFUWFU             UWFUWFU
```

```
UWFUWFUWFUWFUWFUWFUWFUWF
WFUWFUWFUWFUWFUWFUWFUWFU
FUWFUWFUWFUWFUWFUWFUWFUW
UWFUWFUWFUWFUWFUWFUWFUWF
WFUWFUW
FUWFUWF
UWFUWFU
WFUWFUW
FUWFUWFUWFUWFUWFUW
UWFUWFUWFUWFUWFUWF
WFUWFUWFUWFUWFUWFU
FUWFUWFUWFUWFUWFUW
UWFUWFU
WFUWFUW
FUWFUWF
UWFUWFU
WFUWFUW
FUWFUWF
UWFUWFU
```

INDEX TO MAJOR TOPICS IN THIS MANUAL

INTRODUCTION to U/W-FOCAL for the PDP/8

UWF is a powerful, interactive, language for the PDP/8* which combines ease of use with an extensive set of features, allowing one to perform complex calculations or control laboratory experiments with a minimum of effort. UWF is an advanced version of FOCAL*, a stack processing language which is somewhat similar to BASIC and FORTRAN, but much easier to use! Programmers who are familiar with either of these languages should be able to write programs for UWF almost immediately, while those who are just learning will find that UWF's simplified commands and built-in editor make their progress very rapid.

HARDWARE REQUIREMENTS

The minimum system configuration for running UWF is an 8K machine capable of executing the PDP8* instruction set, and some sort of terminal. UWF will automatically take advantage of any additional memory present as well as permitting the use of high-speed I/O devices such as punched paper tape or an audio tape recorder for program and data storage. There is also a much more elaborate version available for systems licensed to use the OS/8* operating system which provides complete device-independent file support.

*FOCAL, PDP8 and OS/8 are trademarks of Digital Equipment Corp., Maynard, Mass.

LOADING UWF

To load the binary tape containing UWF into your machine proceed as follows:

1) Make sure that the BIN loader is resident in Field 0.

2) Set the Switch Register to 7777 and hit 'ADDR LOAD', then reset Switch 0 if input is from the High Speed reader (leaving 3777), otherwise go to the next step.

3) Place the tape in the reader so that the read-head is positioned over the leader section and hit the 'START' (or 'CLEAR' and 'CONTINUE') switches.

The run light will go off when the tape has finished loading; check to be sure the display is zero, indicating a successful load. If not, repeat steps 1-3 above to see if you get the same checksum error each time. If you do, the tape you are using has probably been damaged and you should try again with another copy.

STARTING THE PROGRAM

In order to start UWF for the first time, do the following:

1) Set the Switch Register to 0100 (UWF's Starting Address)
2) Press the 'ADDR LOAD' and 'EXTD. ADDR LOAD' switches
3) Set switches 0-2 and 6-11 for any options desired (v.i.)
4) Now set the 'RUN/HALT' switch to RUN, and hit 'CONTINUE'

UWF should respond immediately with a congratulatory message and indicate the amount of memory available in your system. For installations with more than 8K, here is how the additional memory space is used:

12K	Expanded symbol storage	or	FCOM
16K	One additional program area	"	"
...			
32K	Five more program areas, or four plus		FCOM

If you wish to limit the amount of memory available for any reason, you should set switches 9-11 in the switch register before proceeding with Step 4:

Switches 9-11 (octal): 0=All, 1=28K, 2=24K, 3=20K, 4=16K, 5=12K and 6 or 7=8K

There are a number of other 'custom' features which can be installed automatically when the program is first started up. These options are controlled by the setting of bits 0-2 and 6-8 in the switch register. The first group (0-2) selects various terminal options while the second group (6-8) controls additional features. Switches 3-5 are ignored and may be set to any value. Once UWF has been started the first time step 3 is unnecessary and the switches may remain set to '100'.

Switch 0: Use 'CRT-style' rubouts instead of 'backslashes'
Switch 1: Output four 'nulls' after every Carriage Return
Switch 2: Print error messages on a new line

Switch 6: Add three extra 'secret variables' (&,:,\)
Switch 7: Add the 'KONTROL' command and the 'FDIN' function
Switch 8: Add the 'FCOM' and 'FBUF' functions (requires 12K)

Example: a switch setting of '4134' limits the program to 16K, adds FCOM, FBUF and the digital I/O routines, and installs 'scope rubouts'. The '100' bit is ignored.

Some of these patches can also be installed (or removed) after the program has been started - see Appendix II for further details. Note that adding the 'FCOM' function reduces the effective memory size by 1 field, hence users with 16K who add this option will normally lose the first additional program area. Since it might be more desirable in this particular case to have FCOM replace the extra variable storage, there is a 'magic location' which can be changed (-before- you start things up!) to effect this arrangement. (16K configurations only - see Appendix II for details.)

Note that UWF runs with the interrupt system -ON- which allows program execution to overlap certain I/O operations. The result is faster run times, a 'live' keyboard and the possibility of adding 'background' tasks which can be controlled by the program or 'high-level' interrupts in which an external event causes the execution of a specific group of statements within the program.

With the interrupt system enabled, however, it is possible that an 'unknown' device will create a continuous interrupt and thus prevent UWF from running. If the 'RUN' light goes on but there is no output as soon as you hit the 'CONTINUE' switch halt the machine, hit the 'RESET' or 'CLEAR' switch a few times and then restart at location 100. If UWF still appears to be stuck in an endless loop you will probably have to add an appropriate 'clear flag' instruction to the interrupt routine. See Appendix II for the proper location.

UWF's CONTROL KEYS

UWF recognizes the following control keys:

1) CTRL/F is the master program break key - it will restart UWF at anytime, assuming that the program is still running.

2) CTRL/C is the monitor break key. It will eventually trap to a resident 'ODT' package which is not yet implemented.

3) CTRL/S (XOFF) stops output to the terminal. This provides users with video terminals time to inspect their results.

4) CTRL/Q (XON) resumes output to the terminal. Some terminals issue XON/XOFF codes automatically when the screen fills up.

5) The RETURN key is used to terminate all command lines. UWF will not recognize a command until the RETURN key is typed.

6) The RUBOUT or DELETE keys are used to cancel the previous character. On hard-copy terminals a '\' is echoed for each character deleted. On video terminals they just disappear!

7) The LINEFEED key is used to retype a command line -before- typing RETURN in order to verify that all corrections were made properly. This is mostly for hard-copy terminals.

Remember: UWF can be interrupted at any time simply by typing CTRL/F. This is accomplished by holding down the CTRL key and then typing the letter 'F'. UWF will respond by typing a question mark (?) followed by the line number where the program was interrupted and then print an asterisk to indicate that it is ready for further instructions:

```
?@ 05.13                    UWF was interrupted at line 5.13
```

DIRECT and INDIRECT COMMANDS

UWF prints an asterisk (*) whenever it is in command mode waiting for new instructions. You can then type in either 'direct commands' which are executed immediately, or 'indirect commands' which are saved for execution at a later time. To use UWF as a fancy 'calculator' simply give it a 'direct command' and hit the RETURN key. For example, if you enter the command:

```
*TYPE PI
```

UWF will print the value '3.141592654E+00', correct to 10 significant figures. The Direct Command feature is the essence of interactive programming since it permits one to work through a long calculation a step at a time, or to try out several different ways of doing something. You can experiment with any of UWF's commands by simply typing them in as you read through this manual.

Indirect Commands always begin with a line number which indicates the order in which they are to be executed. They may be entered in -any- order, however, and can be examined or changed at any time. Changes to indirect commands are facilitated by the use of UWF's built-in editor which allows lines to be modified, moved to a different part of the program, or deleted. Since indirect commands can be selectively executed by direct commands it is possible to build a very powerful set of 'macros' which can then be called with just a few keystrokes.

Line numbers in UWF have a 'dollar and cents' format: 'XX.YY' where 'XX' may range from 00-31 (the 'group' number) and 'YY' may have any value from 00-99 (the 'step' number). Group 0 and Step 0 both have special meanings in some commands, so the first line of the program is usually labeled '1.1'. Notice that leading and trailing zeros are not necessary, but one must always include a space after the line number to separate it from the commands on the rest of the line. Here are some sample indirect commands:

```
*3.61 TYPE !
*12.7 COMMENT
*1.99 QUIT
```

A standard programming practice is to index sequential commands by an increment of either '.05' or '.1'. Thus line '1.2' would be used for the statement following line '1.1' rather than line '1.11'. This leaves room to insert up to 9 additional lines in case changes to the program are necessary. Of course lines can always be moved to make room, but it is a nuisance to have to do this and such changes might require alteration of other parts of the program as well.

GROUP and RELATIVE LINE NUMBERS

Several UWF commands are capable of operating on all statements with the same group number. To reference an entire group of lines one simply specifies the group number without designating any particular program step: 'WRITE 1', for example, will list all the program steps in 'Group 1'. Since the number '1' and the number '1.00' are indistinguishable to UWF, it is not possible to write just line 1.00 without writing the rest of the lines in the same group as well. For this reason the first line in a group is generally reserved for comments in order to avoid any complications with group operations.

UWF can also designate a 'sub-group' operation consisting of all the lines following a specified line in the same group. Such operations are indicated by a 'negative line number': 'WRITE -1.5', for instance, will list all of the lines in Group 1 starting from line 1.5 (if it exists).

Line numbers in the range '.01-.99' are termed 'relative line numbers', i.e. they refer to lines in the -current group-, rather than to lines in 'Group 0'. The use of such line numbers is encouraged because it makes the program more compact and also allows subroutines to be moved easily from one part of the program to another without having to worry about internal references. Lines with numbers less than 1.00 -can- be saved as part of the indirect program, but they can only be executed when the program is started from the beginning since there is no way to branch to them at a later time.

Finally, references to line '0' also have a special meaning. A few commands interpret such references to mean 'the entire program', while most others regard 'line 0' as a reference to 'the next command'. Line 0 itself is the permanent comment line (the 'Header Line') at the beginning of the program.

PUNCTUATION

It is a common practice to put several commands on the same line in order to reduce the amount of paper required for listing the program as well as to consolidate related operations. A 'semicolon' (;) is used to separate such commands:

```
*SET A=5; TYPE A
```

Commands which operate on more than one expression use a 'comma' to separate the values. Thus the command 'TYPE A,B' is equivalent to 'TYPE A; TYPE B'. Spaces may be included to improve the readability of a program, but one must remember that 'space' is a 'terminator' (equivalent to a comma) so the command 'TYPE A B' is interpreted as 'TYPE A,B', not as 'TYPE AB'.

NUMBERS AND VARIABLES

UWF can handle up to 10-digit numbers with a magnitude range of 10^615. Numbers may be written as signed or unsigned quantities and may include a decimal fraction as well as a 'power-of-ten' exponent indicated by the letter 'E'. All numbers are stored internally in a 'floating-point' format with 35 bits of mantissa and 11 bits of exponent. This is equivalent to more than 10-digit accuracy. UWF will respond with an error message if a user attempts to enter a number with too many digits. The following all represent the value 'sixty':

```
60      60.00      6E1      600.0E-1
```

UWF also allows letters to be treated as numbers so that questions may be answered with a 'YES' or 'NO' response rather than with a numeric reply. When decoded in this manner, the letters 'A-Z' have the values '1-26', except that the letter 'E' always means 'power-of-ten'. Thus the answer 'NO' would have the numerical value '155' and the number 'sixty' could also be written as '0DT' or '0FEA'. Note that the leading '0' is only required when incorporating such 'numbers' into a program. It is not required as part of a user response.

VARIABLE NAMES

Variables are used to store input values or to save intermediate results. Variables are thus like the storage registers on a calculator except that the programmer may make up his own names to designate the values stored. UWF allows variable names of any length, but only the first two characters are retained internally. Thus the names JOHN and JOE would both refer to the variable 'JO'. The first character of a variable name must obviously not be a number, nor can it be the letter 'F' since that letter is used to designate functions. However UWF does allow symbols such as '$' and '"' to be used as part of a variable name so you can have quantities such as A$, A', and A". Variables are always stored as numeric quantities; UWF does not currently have 'string' variables.

THE SYMBOL TABLE

The names of the variables which have been used by the program are saved (along with their values) in a region of memory called the 'Symbol Table'. This area is completely independent of the area used to store the program so changes to the text buffer do not affect any of the values stored in the symbol table. This is extremely convenient since it allows a program to be interrupted, modified, and then restarted somewhere in the middle, knowing that any intermediate results obtained will still be available. Of course the programmer may examine any such values simply by TYPEing them out, or he may change a few values with a direct command before restarting the program. Variables are always assumed to have the value 'zero' until another value has been assigned. The 'TYPE $' command can be used to list all the values in the Symbol Table in the order they were defined by the program. The ZERO command is used to clear the table, or to selectively set some of the variables to zero. Variables with the value '0' may be replaced by other, non-zero variables when the symbol table fills up. This is transparent to the programmer since 'undefined' variables are always zero anyway.

PROTECTED VARIABLES

The symbols {!,",#,$,%} and optionally {&,:,\}, along with the value of 'PI', are 'protected' variables which cannot be replaced or removed by a ZERO command. This makes them useful for saving results which are needed by a second program. Since they cannot be input or output directly and do not appear in a symbol table dump, they are also sometimes called 'secret' variables. Note that UWF automatically sets 'PI' equal to '3.141592654' so users should not use 'PI---' as a variable name or this value will be lost. The variable '!' ('bang') is used as the dimension constant for double subscripting (v.i.) and many of the remaining 'secret variables' serve as dummy arguments for Program Defined Functions (see page 66). To TYPE the values of these variables you must prefix a '+' sign or enclose them in parentheses: TYPE +! or TYPE (!) will output the value of the first one.

SUBSCRIPTED VARIABLES

Variables may be further identified by attaching a subscript enclosed in parentheses immediately after the name, e.g. 'A(1)'. Such subscripts may consist of arithmetic expressions involving other subscripted variables so quite intricate relations can be developed. Unlike many other high-level languages, UWF does not require any 'dimension' statements for processing subscripted variables, nor are the subscripts limited to only positive integers (they are limited to 12 bits, however). A variable such as 'APPLE(-PIE)' is thus perfectly acceptable although UWF will view this more prosaicly as simply 'AP(-3)'. Non-subscripted variables are the same as those with a subscript of zero, i.e. 'A' = 'A(0)'.

To handle double subscripting, UWF -does- require a small amount of additional information. Before using a double subscripted variable the programmer must store the maximum value of the first subscript in the protected variable '!'. This value may be greater than the actual maximum without incurring any storage penalty, but if it is too small more than one array element will be stored in the same location. Since this single 'dimension constant' is used for all arrays it should be chosen for the largest array in cases where the program uses several different sizes.

To illustrate: suppose that operations on a 5x5 array were necessary. Then '!' ('bang') should be set to 5. If 3x3 arrays were also needed simultaneously (which is not very likely) their elements would all be unique and only 9 storage locations would be used, not 25. Non-square arrays are handled just as easily: a 5x20 array would still only require that '!' be set to 5 since that is the maximum value of the -first- subscript. This method of storing two-dimensional arrays proves very convenient for a wide range of linear algebra problems. The value of '!' is generally used as a loop limit so that the routines can be used with any size array.

ARITHMETIC OPERATORS

UWF recognizes 6 basic arithmetic, and 1 special 'character value' operator:

1) + Addition
2) - Subtraction
3) * Multiplication
4) / Division
5) ^ Signed Integer Powers
6) = Replacement
7) ' Value of next character

These 7 operators may be combined with explicit numbers and function or variable names to create 'Arithmetic Expressions' such as:

```
X= -B/2*A + Y= FSQT(B^2-4*A*C)
```

Such expressions can be used -anywhere- that explicit numbers appear in this writeup. In particular, they may be used to compute line numbers. All expressions are evaluated to '10-digit' accuracy, independent of the format used for output. Intermediate results are generally rounded off rather than being truncated. Most commands use, or operate on, arithmetic expressions. If such expressions are -omitted- a value of 'zero' is always assumed. This occurs frequently when evaluating line numbers, hence you should recall the comments about line '00.00' mentioned on page 6.

PRIORITY of ARITHMETIC OPERATIONS

Arithmetic operations are performed in the following sequence:

First priority - integer powers (^)
Second priority - multiplication (*)
Third priority - division (/)
Fourth priority - subtraction and negation (-)
Fifth priotiry - addition (+)
Last priority - replacement (=)

When UWF evaluates an expression which includes several operations, the order above is followed. For example, UWF evalutes the expression:

X=5^2/5*2-Z=5/2 leaving 'X' equal to 'zero' and 'Z' equal to 2.5

Notice that multiplication has a higher priority than division. This is different from the convention in many other languages where these operations have equal priority. In most cases this difference is of no consequence. The effect of embedding replacement operators is to cause portions of the expression to be evaluated in a somewhat different order than would be the case if they were not included. In the example above, for instance, the quantity '5^2' is divided by the quantity '5*2' and then the quantity 'Z' which is equal to '5/2' is subtracted from the result. However, if one were to add a 'Y=' operator after the first '/' then the quantity '5^2' would be divided by 'Y' which would be equal to '5*2-Z'.

ENCLOSURES

The order of evaluation can also be changed by the use of enclosures. Three different kinds are allowed by UWF: Parentheses '()', Square Brackets '[]', and Angle Brackets '<>'. Subscripts and function arguments are common examples of expressions contained in enclosures. UWF treats all sets identically (they must be matched, of course!), except in some of the monitor commands in the OS/8 version. If the expression contains nested enclosures, UWF will evaluate it starting with the innermost set and working outward. For example:

```
[A=5*<B=2+3>-5]^2          is evaluated as 'four hundred' with 'A'=20 and 'B'=5
```

ABBREVIATIONS

UWF doesn't care whether you tell it to 'TYPE' or 'TAKEOFF'! The reason is that only the -first- letter of the command is recognized, just as only the first -two- letters of a variable name have significance. So while we have been carefully spelling out all the commands in the examples so far, we could just as well have abbreviated them to their first letters.

This feature of the language is both good and bad. On the one hand it greatly reduces the amount of typing required and at the same time increases the number of program steps possible. But on the other hand, a program containing hundreds of single letter commands looks more like a sheet of hieroglyphics than anything else. This makes it quite difficult for the beginner to understand the program logic until he himself has become more familiar with the meaning of all the symbols. For maximum clarity the examples in this writeup will generally be spelled out, but you should realize that the commands 'T PI' and 'TYPE PI' will produce -exactly- the same result.

We will now turn to a detailed examination of all the commands available to the UWF programmer, beginning with the editing commands since they are required for further program development.

COMMAND MODE EDITING

When UWF is in command mode you can use the RUBOUT or DELETE key to correct any typing errors. Each time that you hit this key UWF will delete the precedeing character and echo a '\' on the terminal. If you have a video terminal, and you set switch 0 up when you started UWF for the first time (or made the appropriate patch yourself), hitting DELETE will actually remove the character from the screen. This is obviously much nicer since 'what you see is what you've got'. On the other hand, users with a hard-copy terminal can always just hit the 'LINEFEED' key to have the current input line retyped so that they can see just how it 'really' looks to UWF. There is no limit to the length of input lines, however if your terminal does not handle 'wrap-around' automatically, the practical limit is the width of paper.

In addition to RUBOUT, the BACKARROW (or UNDERLINE key as it is identified on newer terminals) may be used to delete all characters to the left, including the line number of an indirect command. You may then start over again. It is not necessary to hit RETURN although you may wish to do so to get back to the left margin again. Note that LINE FEED will not echo a blank link and RUBOUT will stop echoing a '\' when it reaches the beginning of the line.

The use of 'BACKARROW' as a 'line-kill' character necessarily means that this character (and RUBOUT, of course) cannot be part of the program, but all remaining ASCII characters, both upper and lower case, can be used. Control codes can also be used, but they should be scrupulously avoided since they are non-printing and are therefore impossible to find when they are embedded in a program. In fact, if you ever have a mysterious error in what appears to be a perfectly good command, just try retying it in its entirety to eliminate any 'ghosts'.

Once you hit the RETURN key, UWF will digest whatever you have typed, so subsequent changes require the use of the editing commands. The text buffer can hold approximately 7000 (decimal) characters - typically 3-4 pages of printout. To list any or all of this material you use the WRITE command; to eliminate some of it you use ERASE and to make changes without having to retype the unchanged part, you use the MODIFY command. This command can also be used to MOVE parts of the program to a different location.

W R I T E

The WRITE command, without any modifier, will list all of the indirect commands currently saved in the text buffer. Lines are typed out in numerical sequence, no matter in what order they were entered, and are separated into the groups you have specified. For this reason it is very convenient to use a different group number for each major part of the program even if such a section only has a few program steps. Using the first line (line XX.00) for a COMMENT to describe the purpose of that section is also highly recommended.

The WRITE command can also be qualified by a string of numbers to limit the listing to selected portions of the program. 'WRITE 1', for example, will print out just the commands belonging to Group 1, while 'WRITE 2.2' will list only that single line. A command such as 'WRITE 1,2,3.3,4.9,5' will list 3 groups and 2 single lines, in the order specified. Of course you should try to plan your program so that it executes smoothly 'from top to bottom', but if you do need to add a major section at the end, the WRITE command can be used to at least make a listing showing the logical program flow. Another convenient feature of the WRITE command is the ability to list a specific line and all lines following it within the same group. This is done by specifying a -negative- line number. Thus 'WRITE -1.5' will list line 1.5 (if it exists) plus the remainder of Group 1. The WRITE command will not produce an error if the line or group you specified is missing - it will simply not list it: What you see is what you've got!

E R A S E

The ERASE command is used to delete parts of the program. 'ERASE' without a qualifier deletes the entire program, while 'ERASE 2' will delete just Group 2. Other possibilities are 'ERASE 9.1' which will only remove that single line, and 'ERASE -4.5' which will eliminate the second half of Group 4. Since 'ERASE 0' erases everything, you must use an 'ERASE -.01' command to erase all of Group 0. There is no way to erase lines such as '2.00' without erasing the entire group at the same time; this is one restriction on the use of such lines. Unlike the WRITE command, only a single qualifier may be used with ERASE, and UWF will return to command mode immediately after executing the commmand. Typing in a new line with the same number as an old one will effectively erase the previous version. Entry of just a line number by itself will result in a 'blank line' which may be used to separate sub-sections of a program. Note that this treatment of blank lines differs from that used by BASIC. Blank lines will be ignored during program execution.

M O D I F Y / M O V E

To change a program line or to move it to a different part of the program, you must use the MODIFY or MOVE commands. MODIFY without a qualifier can be used to examine the header line, but it cannot be used to change this line. MODIFY with a single line number permits changes to the line specified while a MODIFY (or MOVE) with -two- line numbers allows similar changes, but saves the modified line with the new number. The old line, in this case, remains just as it was.

MODIFY only operates on single lines (at the moment), so a command such as 'MODIFY 1' will allow changes to line 1.00, not to all of Group 1. Similarly, 'MOVE 2,3' will move line 2.00 to line 3.00; it will not move all the lines in Group 2. Since UWF does not have a 're-number' command, moving lines and then erasing the old copy is the only way to add additional lines when you forget to leave enough room between sequential line numbers.

After you have entered a MODIFY (or MOVE) command, UWF will optionally print out the line number and then pause until you enter a search character. As soon as you have done so, the line specified will be typed out through the first occurrence of this character. If you want to insert material at this point, just type it in; if you want to delete a few characters, simply use the RUBOUT or DELETE key. Other editing options may be invoked by typing one of the following control keys. Note: mistakes made while trying to modify a line often lead to embedded control codes, so if you do get confused, just type CTRL/F and try again.

CTRL/F	Aborts the command - the line is unchanged
CTRL/G (BELL)	Rings bell and waits for a new search char.
CTRL/J (LF)	Copies the rest of the line without changes
CTRL/L (FF)	Looks for the next occurrence of search char.
CTRL/M (CR)	Terminates the line at this point
BKAROW, UNDRLN	Deletes all chars to the left, except lineno.
RUBOUT, DELETE	Deletes previous character, as in command mode

The last two operations are similar to those available during command mode except that BACKARROW or UNDERLINE does not delete the line number. To remove the first command on a line containing several commands, just enter a semicolon (;) as the search character, wait for the first command to be typed out, hit BACKARROW or UNDERLINE and then hit the LINE FEED key.

CTRL/G and CTRL/L may be used to skip quickly to the part of the line requiring changes. If the change(s) you wish to make involve frequently used characters (such as an '='), you can initially select a different symbol which occurs less frequently and then use BELL to change to the character you really wish to find. Or you can simply keep hitting the FORM FEED key to advance through the line. In case your terminal happens to respond to a FF, you will be pleased to know that UWF does not echo this character!

If you just want to move a line from one location to another, type a LF as the initial search character. If you are adding new commands in the middle of a line, be sure to use the LF key - not the RETURN key - to finish copying the rest of the line. Otherwise you will lose the commands at the end of the line and you will have to MODIFY the line a second time in order to re-enter them! If you have a hard-copy terminal you may wish to WRITE out the line after you have modified it to check for additional errors. With a video terminal, on the other hand, the corrected line will be displayed just as it is.

If you have many lines to move (say all the lines in Group 5), and you have a slow terminal, you can disable the printout during the Move in order to speed things up. To do this, simply disable the keyboard echo by using the '0 I' command (this is discussed on page 46). A disadvantage to this method is that not even the MOVE commands will be printed so you have to operate 'in the dark', but this is still the best way to make such a major program change. To restore the keyboard echo just hit CTRL/F.

On video terminals the number of the line being modified is printed out at the beginning so that the changes will be properly positioned on the screen. With a hard-copy terminal, however, the line number is not normally printed in order to leave as much room as possible for rubouts and insertions. Appendix II indicates the location to change if you wish to add the line number printout in this case.

EXPANDED TEXT STORAGE

If your machine has more than 12K of memory, UWF will automatically use Fields 3-7 for additional text buffers. This allows such systems to keep several different programs in memory at the same time which is obviously a very convenient thing to do. The LOOK command is then used to select the desired 'area' for editing, program execution, etc. Programs in different areas are essentially independent and may use the same line numbers, but the symbol table and the 'stack' are shared by all areas.

The LOOK command has the form: 'LOOK Area', where 'Area' has the value '0' for the main text buffer and '1', '2', '3', etc. (up to 5) for the additional fields. LOOK always returns to command mode and is normally only used as a direct command. 'L 1' will switch to Area 1 while 'L 0' (or just 'L') will return to Area 0. For calls between program areas, see the LINK command described later on page 34.

INPUT / OUTPUT COMMANDS

UWF's I/O commands are called 'ASK' and 'TYPE', respectively. The TYPE command has appreared previously in a few of the examples; basically it converts the value of an arithmetic expression to a string of ASCII characters which are then sent to the terminal, or to whatever output device has been selected as a result of an appropriate 'OPEN' command (see page 44). Similarly, the ASK command is used to input numeric values, either from the keyboard, or from another input device. Both of these commands recognize 6 special operators for controlling the format of I/O operations. These operators are, in fact, just the symbols previously identified as 'protected variables' and it is because of their special significance in ASK / TYPE commands that they cannot be input or output directly. These operators, and their meanings, are as follows:

1)	!	Generate a new line by printing a CR/LF
2)	"	Enclose character strings for labeling
3)	#	Generate a RETURN without a LINE FEED
4)	$	Print the contents of the Symbol Table
5)	%	Change the output format
6)	:	Tabulate to a given column or ignore input

You will notice that these are mostly 'output' operations. Nevertheless, they perform the same function during an ASK command that they do in a TYPE command. The '#' operator does not work on all I/O devices and is therefore seldom used. It was originally intended for overprinting on the same line, but may be easily patched (see Appendix II) to generate a FORM FEED, should that be desirable. The remaining operators will now be discussed in greater detail.

THE NEW LINE ! (BANG) OPERATOR

The '!' operator is used to advance to a new line. UWF never performs this function automatically, so output on a single line may actually be the result of several ASK or TYPE commands. 'Bang' operators can be 'piled together' to produce multiple blank lines: 'TYPE !!!!!', for example, would advance 5 lines. Note that to produce a single blank line may require either 1 or 2 '!'s, depending upon whether anything has been written on the first line.

THE QUOTE " OPERATOR

UWF uses the '"' operator to enclose strings which are output just as they appear in the program. Thus the command: TYPE "HELLO THERE, HOW ARE YOU TODAY?" would simply print the message enclosed by the quote marks. The 'ASK' command uses such output for prompting: ASK "HOW OLD ARE YOU? ",AGE will print the question and then wait for a response. In some cases the TRACE operator (?) is also useful for printing labels during an ASK or TYPE command - see page 52.

THE SYMBOL TABLE DUMP $ OPERATOR

The Symbol Table Dump operator ($) has already been mentioned briefly on page 8. It prints all the symbols defined by the user's program in the order in which they were encountered. It does not print the values of the 'secret variables'. To conserve paper and to permit as many symbols as possible to be listed on a video terminal, the listing normally has three values per line. This format can be changed simply by specifying a different number after the '$'. Thus 'TYPE $5' will change the default value to 5, which is convenient on terminals which can print up to 132 characters per line. The total number of symbols possible depends upon the amount of memory available. In an 8K machine there will only be room for about 120 variables while in a 12K machine one can have approximately 675. For internal reasons, a Symbol Table Dump always terminates execution of the command line it is on, hence commands following it on the same line will not be executed.

THE FORMAT % OPERATOR

The format operator (%) allows UWF to print numeric results in any of three standard formats: integer, mixed decimal, or 'floating-point' (scientific notation). A format remains in effect until another one is selected. Initially UWF is set to print all results in full-precision scientific notation so that all digits of a result will be output. However for many calculations a 'decimal' or 'integer' style of output is more desirable. Such formats are selected by the value of an arithmetic expression following the '%' operator which has the form:

%ND.DP

where 'ND' is the Number of Digits to be printed (the result will be rounded off to this precision), and 'DP' is the requested number of Decimal Places. 'DP' should be smaller than 'ND' unless 'ND' is zero; if 'DP' is zero the result will be an 'integer' format and no decimal point will be printed. Thus the command 'TYPE %2,PI' will produce the result ' 3'.

Notice that the form of the format specification is similar to that used for line numbers. This may help to explain why it is necessary to use '%5.03', rather than '%5.3', when you wish to have 5 digits printed with up to 3 decimal places. The number of decimal places actually printed may not be exactly what you have requested. If UWF finds that the number being output is too big to fit the format you specified it will reduce the number of decimal places. For example, if you try the command:

TYPE %5.04, 123.456

you will actually get the value ' 123.46' printed since it is not possible to show 4 decimal places with only 5 digits. Note however that UWF -did- print the 5 most significant digits in a format approximately like the one requested. Programmers accustomed to dealing with large powerful computers which print only a string of '*****'s under similar circumstances should find UWF's approach quite sensible.

What happens if the number is so large that even the most significant part overflows the field specified? In that case UWF automatically switches to floating-point format for that value so you will be able to see an unexpected result without having to re-run the entire program! You can try this out simply by typing the value '123456' without changing the format from the previous setting. UWF will print: ' 1.2346E+05'.

To purposefully select a floating-point format you should specify one with 'ND' equal to 0. Thus the format '%.05' will print 5-digit numbers in proper scientific notation (1 digit before the decimal point). The default format when UWF is first loaded is '%.1' which prints all 10 digits. To return to this format you can simply specify '%', since the value '0' is treated the same as '%.1'. Note that using an arithmetic expression for the format specification, rather than just a fixed number, permits the format to be changed dynamically while the program is running: '%VF' would select a format using the value of the variable 'VF'.

Finally, note that UWF will never print more than 10 significant digits - the limit of its internal accuracy. If the quantity 'ND' is larger than this, spaces will be used to fill out the number. If the quantity 'DP' is larger, zeros will be added. In any case, if the number is negative UWF will print a minus sign just ahead of the first digit. A plus sign is never printed (except as part of the exponent), but a space is reserved for it anyway. An additional space is also printed at the beginning in order to separate the number from any previous output. This space may be omitted (or changed to an '=' sign) by making the patch shown in Appendix II.

To summarize the various output format settings:

%N	'N' digit integer format
%N.D	'N' digits with up to 'D' decimal places
%.D	'D' digits in scientific (F.P.) notation
%	the same as '%.1' - full precision F.P.

THE TAB : OPERATOR

The tab (:) operator provides a convenient way to create column output. The expression following the colon is used to set the column, i.e. ':10' specifies column 10. The tab routines do not attempt to go backward if the column specified is to the left of the current print position - the command is simply ignored in this case. 'Tabs' are recommended in place of a string of spaces so that changes in the output format will not affect subsequent columns.

There are two special cases: tabbing to column 0 and tabbing to a negative column. Neither is possible since columns are numbered from 1 to 2047, but both are useful operations. Expressions which have the value zero can be evaluated by the tab operator within a TYPE command -without- producing any output. This is convenient occasionally, especially for calling the FOUT function (see page 61). Tabbing to a negative column has been given a quite different interpretation, however.

Since the current version of UWF can only input numeric values with the ASK command, there is a need for a method to skip over label fields when re-reading output produced by another program. This facility is provided by 'tabbing' to a negative column number which causes no output, but instead reads and ignores the specified number of characters. Thus the command 'TYPE :-1' will -read- 1 character from the input device. This may well appear confusing, since we have an 'output' command waiting for input, so the 'ASK' command may be used instead: 'ASK :-1' performs the same function. This feature provides a simple way to get the program to wait for operator intervention. For example, the command 'TYPE "TURN ON THE PUNCH":-1' would print the message and then wait for any keyboard character to be typed. An 'ASK :-2000' command will let a visitor type almost anything s-he likes into the computer without danger of destroying a valuable program.

Having discussed all the ASK/TYPE operators, there is really very little more to explain about the commands themselves. TYPE can evaluate a whole series of arithmetic expressions which are generally separated by commas or spaces or one of the above operators, while ASK can input values for a whole list of variables, again separated by commas or spaces. Here are a few examples:

```
TYPE !!:10"TODAY IS"%2,15 %4"  OCTOBER"1978!
ASK "TYPE A KEY WHEN YOU ARE READY TO GO":-1
TYPE !"THE ROOTS ARE:" %5.02, R1 :20 R2 !!
ASK !"WHAT IS THE INITIAL VALUE OF X? " IX
```

Notice that the TAB and NEW LINE operators can be included in an ASK command to help format the input. Thus 'ASK X :10 Y !' would keep the input responses in two nicely aligned columns. It is quite convenient to be able to output the necessary prompting information with the ASK command; other languages frequently require separate commands (such as 'PRINT' followed by 'INPUT') for these operations. The trace operator described on page 52 is also useful in ASK and TYPE commands when one is interested in a 'minimal effort' I/O structure.

One other feature of a TYPE command should be noted: it is possible to save the value of a quantity being 'TYPEed' just by including a replacement operator in the expression. Thus 'TYPE X=5' will output the value '5' and also save it as the value of the variable 'X'.

Numeric input for the ASK command can take any of the forms listed on page 7, specifically: signed integers, alphabetic responses, decimal values or numbers containing a power-of-ten exponent. Because such numbers are processed as they are being input it is not possible to use the RUBOUT key to delete an erroneous character. Rather, one must effectively hit the 'clear key' (as on a calculator) and then re-enter the entire number. The 'clear' function is indicated by typing a 'BACKARROW' or 'UNDERLINE' just as it is during command input. If you do attempt to use RUBOUT, no '\' will be echoed which serves as a reminder that this key is ignored during an ASK command.

INPUT TERMINATORS

UWF allows a variety of characters to serve as input terminators. In addition to the RETURN key, one may use a SPACE (spaces in front of a number are ignored, but may be used to format the input as desired - spaces following the number always act as a terminator), a COMMA, SEMICOLON, or other punctuation marks such as a QUESTION MARK or COLON. A 'period' is, of course, recognized as a decimal point, but a second period also works as a terminator. Any of the arithmetic operators also serve as terminators; in particular, the '/' and '-' characters are often convenient. This allows responses such as '1/2' or '1-5' for the values of -two- different variables.

In fact, any character -except- 0-9, A-Z, RUBOUT and LINE- or FORM-FEED can be used to terminate the response to an ASK command. More to the point, however, is the fact that the program can test to see which terminator was used. This allows a very simple input loop to read an indefinite number of items until a specific terminator (a '?', for instance) is found. See the discussion of the FTRM function on page 63.

The ALTMODE or ESCAPE key is a special case: typing either of these keys leaves the previous value of the variable unchanged. This allows quick responses to repeated requests for the same value. The program, of course, can pre-set the value of the variable so that an ALTMODE response will merely confirm the expected value.

ARITHMETIC PROCESSING COMMANDS

There are four commands in this group: SET, XECUTE, YNCREMENT and ZERO.

S E T

The most frequently used command in the UWF language is the SET command. This command evaluates arithmetic expressions without producing any output (except when the trace feature is enabled - see page 52). Such expressions typically have the form:

SET Variable Name = Arithmetic Expression

But more general expressions, particularly those containing sub-expressions, are perfectly acceptable. Thus a command such as 'SET A=B=C=5' could be used to set all three variables to the same value while 'SET I=J+K=1' would initialize the value of 'K' (to 1) as well as set 'I' to 'J+1'. Expressions used with the SET command do not need to contain replacement operators: the command 'SET FIN()' could be used, for instance, to input a single character. The value of the function would not be saved, however; this is sometimes useful when calling 'I/O' functions for their 'side-effects'.

Note that the word 'SET' (or its abbreviation 'S') is not optional as it is in some other languages. The flexible syntax employed by UWF makes it mandatory that every command begin with a command letter. One SET command, however, will process as many expression as can fit on a single line. The expressions should be separated by commas or spaces, for instance:

'SET A=1,B=2,C=A+B' which is equivalent to 'SET C=(A=1)+B=2'

Another point to remember is that the same variable may appear on both sides of an

'=' sign. Thus 'SET X=X+5' has the effect of redefining the value of 'X' to be 5 more than the initial value. This can get to be tricky if the same variable appears several times in a single expression on both sides of replacement operators. The rule here is that in each instance the variable will have its current value until the entire expression to the -right- has been evaluated; then it will be replaced with the new value. To give a fairly simple, yet intriguing, example:

SET A=B+A-B=A which is equivalent to SET C=B+A,B=A,A=C-B

This will interchange the values of 'A' and 'B'. Another expression which does the same thing is: SET A=B+0*B=A. Notice that the processing of this expression involves two different values of 'B': The first time 'B' is encountered it is on the right side of an '=', so its current value is used; the second time it is on the left side, so the rest of the expression is evaluated, the substitution made, and then processing of the first part is resumed. Thus 'A' retains its original value until the very end (when it is replaced by the initial value of 'B', which was saved on the stack).

Z E R O

The special case of 'SET Var=0' is conveniently handled by the ZERO command. A single ZERO command may be used to set several variables to zero, making it very convenient for initializing sums and 'flags': 'ZERO A,#,C' will set those three variables to zero. As a special case, if no variables are specified, the ZERO command clears the entire symbol table. This effectively sets -all- the variables to zero since this is the default value for 'undefined' quantities.

One other use of the ZERO command should be mentioned. When the Symbol Table fills up, UWF tries to replace any variables which have the value '0' with new variables. This procedure succeeds as long as there is at least 1 variable with this value, since that one will simply be renamed, and no matter what the name, it will always be zero. As a result of this scheme, programmers may regain symbol table space by ZEROing unneeded variables when they are finished with them.

Y N C R E M E N T

Another special case of the SET command - 'SET Var = Var + 1' is handled by the YNCREMENT command. This command allows a list of variables to be either increment or decremented by the value '1'. The command 'Y K', for example, is equivalent to 'SET K=K+1' while 'Y -J' is the same as 'SET J=J-1'. Of course commands such as 'Y N,O-P' are permitted; this one increments the variables 'N' and 'O' and decrements 'P'. Either commas, spaces or minus signs may be used to separate the variable names.

X E C U T E

The XECUTE command has been included for compatibility with earlier versions of UWF. Its purpose was to evaluate arithmetic expressions without setting useless 'dummy' variables. This is now accomplished by the SET command itself simply by omitting the replacment operator. Thus 'SET FOUT(7)' may be used to ring the bell on the terminal. Internally 'SET' and 'XECUTE' are identical; it is recommended that SET be used in new programs.

BRANCH and CONTROL COMMANDS

This class of commands is used to test arithmetic results, set up loops and otherwise control the sequence of command execution. There are 11 commands in this catagory - UWF has a very rich control structure built around two fundamentally different types of transfers: the 'GOTO' branch and the 'DO' call. Both interrupt the normal sequence of command execution, but the GOTO is an unconditional branch while a DO call eventually returns to the next command following the call. The DO command is similar to the 'GOSUB' in BASIC, but is considerably more flexible.

G O T O

This command has the form 'GOTO line number'. It causes an immediate transfer to the line specified. The 'GO' command is the usual way of starting the indirect program at the lowest numbered line; it may be used to start the program at any other line as well: 'G 2.1' will start at line '2.1'. An explicit line number may be replaced by an arithmetic expression to create what FORTRAN calls an 'Assigned Goto': 'SET X=5.1 . . . GOTO X'.

D O

The DO command is effectively a subroutine call. A DO command without a modifier (or equivalently, a 'DO 0' command) calls the entire stored program. This may be used as a Direct Command in cases where you wish to follow such action with additional commands, e.g. 'DO;TYPE FTIM()' might be used to check the running time of a benchmark program.

DO also accepts a list of line and group numbers such as 'DO -.7,8,9.1', which would call the subroutine starting at line XX.70 in the current group, then Group 8 and finally line 9.1. 'DO' is completely recursive: a DO may thus 'do' itself! Note that the commands called by a DO are not designated anywhere as subroutines - they may be, and usually are, just ordinary commands somewhere in the main program. This is one of the major differences between DO calls in UWF and GOSUBs in BASIC. Suppose, for example, that the program had a line such as:

1.3 ZERO A,B,C; SET D=5, E=6

which occurred in Group 1 as part of an initialization sequence. If the same set of commands were needed later in Group 12, one would only need to write 'DO 1.3'. This facility for re-using common parts of the program is akin to writing 'macros' and is generally considered to be a good programming practice. The one feature missing from the DO command is the ability to explicitly pass arguments to the 'sub-routine'; this must be handled by the use of 'common' variables. As you will see later on (page 66), Program Defined Function calls provide this capability in a somewhat limited form.

A DO call may be terminated in one of four ways:

1) There are no more lines to execute in the range specified
2) A RETURN command is encountered
3) A loop containing a DO is terminated by a NEXT or BREAK
4) A GOTO transfers to a line outside the range of the DO

The first condition is the most common, especially for single line calls. The second condition is explained below, while the third is explored in the discussion of the NEXT command. That leaves only the fourth possibility: GOTO branches can be used to terminate a DO call simply by transfering to a line outside of the range; however the line transfered to will be executed first, which can lead to slightly unexpected results. For instance, if the line branched to happens to immediately precede the group, no exit will occur because UWF will find itself back in the proper group again when it finishes the line. Another somewhat similar case occurs when calling a 'sub-group': GOTO transfers anywhere in the same group will be honored without causing a return. Thus if you wish to force a return from a DO call, do it with the RETURN command (v.i.), not with a GOTO.

R E T U R N

The RETURN command provides a way to selectively exit from a DO call in cases where the entire subroutine is not required. Since a 'DO' call always specifies the implied range of the subroutine (a single line or an entire group), a RETURN command is normally not required. There are cases, however, especially when calling a 'sub-group', in which a RETURN is necessary to force an early exit. If there is no subroutine call to return from, RETURN will go back to command mode instead, i.e. it behaves just like a QUIT command. This is a useful feature, since programs which end with a RETURN can be run normally, but can also be called as subroutines via the LINK command (see page 34).

RETURN can also designate a line number, for example: RETURN 5.3. In this case the normal return to the calling point is aborted (except for PDF calls, see page 68) and the program continues from the line specified. This is a very important feature since it effectively transforms a DO call into a GOTO branch. It is all the more useful since it can be 'turned on and off' simply by making the return point an arithmetic expression which, when zero, indicates a normal return, but otherwise causes a branch to the line specified. This gives UWF a 'multiple return' feature which is found in only a few high-level languages.

I F

The form of the IF command is:

IF (Arithmetic Expression) negative, zero, positive

where 'negative', 'zero' and 'positive' are line number expressions not containing commas. Depending upon the sign of the value being tested, the program will perform a 'GOTO' branch to one of the three possibilities. The expression being tested must be enclosed in parentheses and must be separated from the command word by a space.

Not all of the branch options need to be specified, and relative line numbers are especially useful for those which are. Here are some examples of IF commands:

IF (D=B^2-4*A*C) .2,.3,.4	Tests for all 3 possibilities
IF (A-5) 5.1, 5.1	Branches if A is less than 5
IF (-X) .9 or IF (X),,.9	Branches if X is greater than 0
IF [I-J] , .2	Branches only if I equals J
IF <W> .4,,.4	Branches only if W is non-zero

These examples illustrate the flexible nature of the IF command. In commands with only 1 or 2 branch options, if the branch is -not- taken, the next sequential command will be executed - whether this command is on the same line or on the next line (unless the IF is in a FOR loop, v.i.). Here, then, is a case where 'line 0' is interpreted as the 'next command'. Also note (example 1 above) that the expression being tested may contain replacement operators so that the value may be saved for use elsewhere in the program.

O N

The ON command is identical in form to the IF command: ON (exp) N,Z,P. The difference is that DO calls are used in place of GOTO transfers, so upon completion of the subroutine, the program will continue with the next command following the ON test.* This is often a very convenient thing to do since it allows additional processing for specific cases. As with the IF command, not all 3 calls need to be specified, so one can test just for equality (zero), or for some other condition. Notice that an entire group can be called by the ON command.

* The automatic return can be aborted if desired - see the RETURN command for further details.

J U M P

The JUMP command has two distinct forms which have been designed to serve the needs of interactive programs:

```
JUMP  line number              -or-
JUMP  (expression) S1, S2, S3, S4, S5, . . .
```

The first form is a conditional GOTO in which the branch is taken -unless- there is a character waiting in the input buffer. This form is used to test the keyboard for input without interrupting the program if there isn't any. This feature is essential in interactive programs which allow program flow to be controlled dynamically from operator response. For example:

```
1.1 YNCR I; JUMP .1; TYPE I
```

will hang in a loop incrementing the variable 'I' until a key is struck, then type the number of cycles. The character used to interrupt the program can be read with the FIN function (see page 61) and so used to further control program flow. If the example above simply called FIN to read the character directly, the program would hang in the input wait loop and nothing further could be accomplished until the operator struck a key.

The second form of the JUMP command provides a computed subroutine (DO) call which is essentially similar in form to the ON command except that the actual -value- of the arithmetic expression being tested is used (rather than just the -sign- bit) to determine which subroutine to call. The call list is indexed from 1 to N, and any number of subroutines may be specified. Values of the expression which do not match up with a specified call are ignored. In the example shown above, Subroutine No. 4 will be called if the expression has the value 4.5, whereas if the expression has the value -1, 0, or 12.3, no subroutine at all will be called. As with the IF and ON commands, line numbers may be omitted (or set to zero) to avoid a call for certain values of the expression.

Typically the expression is simply the ASCII value of a keyboard character which is used to select an appropriate subroutine. For example:

```
JUMP (FIN()-'@) A,B,C,,E
```

will call subroutine 'A' if the letter 'A' is typed, etc. Notice that typing the letter 'D' is treated as a 'NOP' by this particular command. As with the ON command, the program normally continues with the next sequential command following the subroutine call unless a RETURN command is employed to transfer elsewhere.

L I N K

The LINK command allows systems with more than 12K to call subroutines stored in different text 'areas'*, thus 'linking' such areas together as part of a 'main' program. The command has the form:

```
LINK Area, Subroutine Pointer
```

where 'Area' may have the values '0' or '1' in a 16K system, and up to '5' if sufficient memory is available. The 'Subroutine Pointer' is a line or group (or sub-group) number as described for the DO, ON and JUMP commands. A value of '0' specifies that the entire area is to be used as a subroutine. Examples:

```
L,4             Calls group 4 in Area 0
L 1,-8.5        Calls sub-group starting at line 8.5 in Area 1
L,.3            Calls line XX.30 in the same group in Area 0
L 2,;T "DONE"   Executes all of Area 2, then types 'DONE'
```

Notice that the comma is required punctuation even when the second parameter is zero, as in the last example.** To avoid returning to the calling area at the end of the subroutine, use a RETURN command with a non-zero line number, such as 'R .9' to abort the normal return sequence. By using a computed line number in such a command the calling program can control the return. A 'QUIT' command can also be used to cancel all returns - see below.

The variables created or used by a program in one area are shared by all areas, so be careful to avoid conflicts. Also, since each LINK saves its return on the 'stack', watch out for calls which never return, but simply chain from one area to another. This will eventually lead to a 'stack overflow' which can be cured by using a 'QUIT X' command to cancel all pending returns.

The LINK command functions properly for calls from within the same area, but the DO command is clearly preferable since, for one thing, it can handle multiple calls which the LINK command cannot. LINK can be used in direct commands; it is somewhat similar to the 'LIBRARY GOSUB' command in the OS/8 version.

*For more information on storing programs in different areas, see the discussion on page 18.

**LINK and LOOK differ only in the presence or absence of a second parameter. If only the area is specified UWF returns to command mode (LOOK), otherwise it executes a subroutine call (LINK).

Q U I T

The QUIT command stops program execution and resets the 'stack' pointers so that all pending operations (such as subroutine returns) are destroyed. CTRL/F as well as any execution error performs an effective QUIT, thereby returning to command mode. There are rare occasions, however, when it is desirable to be able to 'quit' and then simulate a keyboard restart so that the program will continue running without actually returning to command mode. This is accomplished by specifying a non-zero line number as a 'restart' point. Thus 'QUIT 1.1' will stop execution, clear the stacks, and then restart at line 1.1. To restart at the lowest numbered line of the program, use a 'Q .001' command. 'QUIT 0' or just 'Q' will stop the program and return to command mode.

It is also possible to use QUIT to specify a restart point for any error condition. This is accomplished by specifying a -negative- line number, i.e. something like 'QUIT -9.1'. This command will not stop the program when it is executed; it will merely remember the line number and then continue with the next command. If an error subsequently occurs, however, the program will be automatically restarted at line 9.1 instead of returning to command mode.

This provides UWF with a somewhat limited error recovery procedure, but one which can be used to take care of certain 'unexpected' conditions which might develop while the program was running unattended. Note that it is up to the user to determine which error caused the restart. One way that this could be accomplished is to select different restart points for differenct sections of the program where specific errors might be expected. This feature should be considered somewhat 'experimental' in the sense that it may not be included in later releases of UWF if other features appear to be more important.

The error trap is automatically reset everytime UWF returns to command mode in order to prevent conditions set by one program from causing an unexpected restart at a later time.

LOOP COMMANDS

UWF has 3 commands for constructing program loops. The FOR command sets up the loop; the NEXT command serves as an optional terminator, and the BREAK command provides a way to exit from a loop before it runs to completion. UWF's loops are slightly different from those in other languages, but the differences, once recognized, are easy to accomodate. Basically, UWF uses 'horizontal' loops which consist of only the statements following the FOR command on the same line. Most other algebraic languages use 'vertical' loops which consist of a number of contiguous program steps with some way to designate the end of the loop.

UWF's approach is convenient for short loops since the commands to be repeated are simply coded on the same line with the FOR command and no 'end-of-loop' designation is required. Loops which require several lines of code are handled just as easily by putting a 'DO' command in the main loop to call all the statements which cannot be placed on the same line. A NEXT command at the end of those statements then serves to designate both the end of the loop as well as the continuation point of the program. Symbolically, UWF's loops may thus have either of these two forms:

```
        FOR * * *; loop commands
```

or

```
        FOR * * *; DO -.YY
XX.YY   first loop command
        second loop command
        . . .
        last loop command; NEXT
```

The latter form is practically identical to that used by BASIC or FORTRAN, with the mere addition of the 'DO' call on the first line.

F O R

This command initializes a loop, assigning a 'loop variable' to count the number of iterations. The form is:

FOR Var = Initial Value, Increment, Final Value ;

or, more generally,

FOR expression 1, expression 2, expression 3 ;

where the first variable to the left of a replacement operator in expression 1 will be used as the loop counter. The semicolon after expression 3 is required punctuation. An increment of +1 is assumed if only the initial and final values are given. Notice that the increment, if specified, is the -second- expression! This is different from the convention used by BASIC and FORTRAN. There are no restrictions on any of the expressions: they may be positive, negative or non-integer. Thus one can increment either 'forward' or 'backward', using any step size. The execution of the FOR command is such, however, that one pass will always occur even if the 'initial value' is greater than the 'final value'. In any case, the exit value of the loop variable will be one increment more than the value it had in the final iteration.

Here are some examples:

```
1)    FOR I=J=1,10;
2)    FOR L=1,N; FOR M=-L,L;
3)    FOR X(I)= 10, -1, 1;
4)    FOR A=0, PI/180, 2*PI;
5)    FOR Q= P/2, Q, 5*Q;
```

Notice that loops may contain other loops (Ex. 2). Such 'nesting' is permitted to a depth of 15 or so, but in practice, loops are rarely nested more than 5 deep. Another point, illustrated in example 5, is that the initial value of the loop

variable can be used by the expression for the increment and the final values; also notice that subscripted variables are permissible as loop indices (Ex. 3).

In example 1 it may appear that both 'I' and 'J' will be used as control variables. This is not the case: only the first variable (in this case 'I') will be incremented. Other variables (such as 'J') may be given values by replacement operators in any of the three expressions, but these values will not change during the loop (unless commands within the loop change them). It is often quite convenient to use the FOR command to initialize several variables used in the loop along with the value of the loop index.

The novice programmer who wishes to try writing a simple loop might begin with the following direct command:

```
FOR I=1,10;TYPE I,I^2,!
```

which will print out the first 10 squares in some (unspecified) format. The more experienced programmer will quickly appreciate UWF's loop structure; for one thing, no rules regarding branches into the middle of a loop are necessary since there is no way to branch to the middle of a line!

N E X T

The normal termination for a loop is at the end of the line containing the FOR command. If the loop contains a GOTO branch, however, the end of the line branched to becomes the terminator. It is convenient at times, especially in direct commands, to terminate the loop in the middle of a line so that other commands which logically follow the loop can be placed on the same line. The NEXT command serves this purpose, as shown in the following example:

```
FOR * * *;  loop commands;  NEXT;  other commands
```

which excludes 'other commands' from the loop.

This construction also works in 'vertical' loops:

```
        FOR * * *; DO -.#
#       commands
        commands
        NEXT
```

The commands executed by the 'DO' will be terminated upon encountering the NEXT command. But more importantly, when the loop is finished, UWF will continue with the first command following the NEXT - thus skipping over the commands in the body of the loop. If this 'NEXT' command were to be omitted or replaced by a 'RETURN', the program would simply 'fall through' to the first statement in the loop (the one indicated by '#' in the example above).

Notice that the NEXT command contains no references to the loop variable. This is a little different from the way most versions of BASIC implement this command, but the effect is quite similar and since only the first letter of the command word is decoded, variations such as 'NI' or 'NEXT-J' may prove helpful to some programmers. Nested loops, of course, may require 'nested NEXTs': 'N;N'. Here is an example which types out all the elements of a 5x5 array a row at a time with a CR/LF printed at the end of each row:

```
FOR I=1,5; FOR J=1,5; TYPE A(I,J); NEXT; TYPE !
```

NEXT has one other feature: it may be used with a line number to specify a continuation point other than the next sequential command. Thus: 'FOR * * *; commands; NEXT .8' will branch to line XX.80 when the loop runs to completion.

Note 1: A NEXT command which is executed outside of a FOR loop is ignored unless it specifies a line number, in which case the branch will always be taken. A 'NEXT' command may thus be placed at the beginning of any line and used as a target for a 'do nothing' branch from within a loop without affecting the normal execution of that line.

Note 2: Loops which contain conditional branches (i.e. 'IF' commands) should be careful that all paths end with an appropriate 'NEXT' if it is desired to skip

over the statements in the loop under all conditions. Whichever 'NEXT' is executed on the final iteration will determine the program flow.

B R E A K

Once a loop has been initiated it must normally run to completion. Branching to a line outside of the loop is not effective: that line will simply be treated as a continuation of the main loop (see comments about GOTO's in a loop in the preceding section). One way to force an exit from a loop would be to set the loop variable to a value greater than the final value. This is obviously not very 'elegant', to say the least, so the BREAK command has been provided to solve this difficulty. A BREAK causes an immediate exit from the loop (preserving the current value of the loop index), and the program then continues with the next sequential command following the BREAK. As you might expect, BREAK may also specify a line number so you can branch to a different part of the program at the same time that you leave the loop. A 'BREAK' without a line number is ignored (just like the NEXT command) if it is encountered outside of a loop, so lines containing BREAKs can be used by other parts of the program. Each BREAK exits from only a single loop, so to exit from nested loops it would be necessary to use multiple BREAK commands: 'B;B 15.1' will exit from 2 loops and then transfer to line 15.1.

MISCELLANEOUS COMMANDS

We will now finish the alphabet: C,H,K,P,U,V are the remaining command letters. 'U' and 'V' are not implemented in this version and may be used for whatever purpose the user desires. The '@' command is also available for expansion purposes.

C O M M E N T

Any command beginning with a 'C' causes the rest of the line to be ignored. Such lines may thus be used for comments describing the operation of the program. In particular, line XX.00 (the first line in a group) should generally be reserved for comments. Branching to a Comment line from within a loop will terminate that cycle of the loop. In this way a 'COMMENT' is equivalent to the Fortran 'CONTINUE' statement. A 'NEXT' command performs the same function and in addition may be used to designate the continuation of the program.

H E S I T A T E

The HESITATE command delays the program for a time specified by the command. The argument (which must be an integer) is nominally in milliseconds, so 'H 1000' will generate approximately a 1 second delay. However, the exact delay is directly dependent upon the cycle time of the machine, so some calibration is necessary. Here is an example using the 'H' command:

```
FOR T=1,9; TYPE "*"; HESITATE 250*T
```

P U N C H

The PUNCH command allows the programmer to save a copy of the text buffer and the symbol table in binary format, ready to reload with the standard Binary loader. This command requires either a High Speed punch or an audio (cassette) recorder. Tapes created with the PUNCH command can only be read with the Binary loader since they are not punched as ASCII characters. The advantage to punching tapes in this format is that they tend to be somewhat shorter than ASCII tapes and they also contain a checksum so there is less probability of an error going undetected. The disadvantage however, is that they are absolute memory dumps and so are not necessarily transportable between different versions of UWF. They also cannot be loaded by UWF itself from a remote location, but require access to the front panel of the computer in order to activate the Binary loader as well as to restart UWF once the tape is loaded.

To use this command (assuming that you have a cassette recorder, but the same procedure applies to a HS papertape punch), advance the tape to an unused area, turn on the recorder and then type 'P' followed by a RETURN. Approximately 5 seconds of leader code will be punched, followed by the contents of the text buffer and then the symbol table. To restore a program (and any symbols in use at the time it was dumped), position the tape at the start of the leader, and while this section is being read, start the BIN loader from the front panel. If you start the loader before reaching the leader section, the computer will halt with a checksum error (AC not zero); hit the CONTINUE switch quickly, and if you are still in the leader, all will be well. After reading in the program tape you must manually restart UWF at location 100 (see page 2).

The PUNCH command always returns to command mode (like MODIFY and ERASE), so it cannot be followed by other commands, and should not be included in the program itself. In systems with more than 12K the PUNCH command will dump the contents of the -current- program area, so to save the program in Area 3, for example, use a 'L 3' command to get to it and then type 'P' to punch it out. A program can only be reloaded into the area that it came from; so if you wish to move a program to a different area you must WRITE it out (rather than PUNCHing it), and then read it in again as explained on page 45.

THE 'OPEN' COMMANDS

In addition to the PUNCH command described above, UWF has a series of 'OPEN' commands which allow ASK and TYPE (or other I/O operations) to use something other than the terminal. These commands consists of -two- words (or two single-letter abbreviations) separated by a space. You may recall that the letter 'O' has already been used for the 'ON' command and wonder how it could also be used for 'OPEN'. 'OPEN' and 'ON' can be distinguished, however, since ON must always be followed by an arithmetic expression. Here is a short summary of the 'OPEN' commands currently available. The mnemonics, which were chosen in part to be compatible with the OS/8 version, are somewhat less than perfect!

OPEN INPUT	O I	Selects the Terminal as the Input device
OPEN OUTPUT	O O	Selects the Terminal as the Output device
OPEN READER	O R	Selects the High Speed Reader for Input
OPEN PUNCH	O P	Selects the High Speed Punch for Output
OUTPUT TRAILER	O T	Punches leader/trailer code (ASCII 200)
OPEN ----,ECHO	O -,E	Connects the Input device to the Output

Only the first two commands (O I and O O) and the ECHO option are useful unless you have a high speed reader/punch (or an audio tape recorder). The list of 'OPEN' commands could also be expanded to include things like 'O L' (for selecting a Lineprinter) or 'O S' to send output to a 'scope display, etc. Such expansion is, however, entirely up to the user.

I/O DEVICE SELECTION

The Input and Output devices are always reset to the terminal when you hit CTRL/F. To select a different device, use the appropriate OPEN command. For example, to read in a program from the High Speed Reader, simply type in an 'O R' command and henceforth, until this assignment is changed, all input to UWF will come from the reader rather than from the keyboard. In particular, even direct

commands will be taken from the reader, so you can set up a program tape to run the machine while you are gone. Also, if the tape contains a listing of a program it will be read into the text buffer just as though you were typing it in yourself. This is an alternative method for saving programs which has the advantage that they are available as ASCII tapes which can be edited or processed by other programs. A 'time-out' trap in the reader routine normally senses when the end of the tape has been reached and then restores the terminal as the input device. A 'backarrow' or 'underline' is printed on the terminal to indicate that it is the active input (and output) device once more. If you need to manually restore the terminal to its usual status, just hit CTRL/F.

Similarly, to select the High Speed Punch (or Cassette Recorder) for use as the output device, just use an '0 P' command. To dump the text buffer on tape, for example, enter the commands:

```
0 P,T; W; 0 T,0                    (do not hit RETURN)
```

and then start the punch or recorder. Hit RETURN and then wait for the asterisk (*) to reappear on the terminal.

To re-read such a tape at a later time, position it in the reader somewhere in the leader section, use the ERASE command to clear the program area, and then type '0 R' followed by the RETURN key. If input is from a paper tape reader, the reader will now begin to read the tape. If input is from an audio recorder you should actually start the tape moving (in the leader section) before hitting the RETURN key, otherwise the first few characters are likely to be 'garbage' as the tape comes up to speed and UWF may well conclude that you have run out of the tape before you have even begun!

It is also possible to use the reader/punch for data storage purposes. This works best with paper tape since the audio recorder lacks a 'stop-on-character' capability, making it difficult for UWF to keep up with the data once the tape has started moving. By way of an example, the following command will read in 50 numbers from the high-speed reader:

```
0 R; FOR I=1,50; ASK DATA(I); NEXT; 0 I,E
```

Notice that an '0 I,E' command is used at the end of the loop to restore input to the keyboard. If this command were omitted the H.S. reader would continue to be used for input, probably causing an error to occur since it is unlikely that the next data value on the tape would correspond to anything expected from the keyboard. The ',E' part of this command is explained more fully in the next section.

THE ECHO OPTION

The ',E' option may be added to either an '0 I' or '0 R' command to specify that the input characters are to be 'echoed' to the output device. Generally this option is -always- used with '0 I' and -never- used with '0 R'. The echo option may at first appear slightly confusing since UWF normally runs with the keyboard echo -on- and thus one comes to expect that whatever is typed will be printed on the terminal. This makes the terminal appear much like a simple typewriter and tends to obscure the fact that if UWF were not sending back each character it received, -nothing- would be printed! The 'ECHO' option must be specified when selecting the input device, or -NO ECHO- will be assumed. Thus an '0 I' command will select the keyboard for input (it may already -be- selected) and effectively turn the echo off. An '0 I,E' command is necessary to restore the echo under program control. Of course any program error, or typing CTRL/F, will also restore the echo.

The ability to disable the input echo is convenient at times since it allows a program to read one thing and possibly print something else. An example of this mode of operation occurs during command input: when you type the RUBOUT key you do not get this character printed, but rather a 'backslash' (\), or on a video terminal, a three character sequence: 'backspace, space, backspace', which effectively removes the character from the screen. UWF programs can also be written to use the keyboard for program control, and in such cases it is often desirable to have 'silent' input. You can try this out quickly by using a direct '0 I' command to disable the echo. Now type in '0', 'space', 'I', 'comma', 'E' and hit RETURN and the echo will return again.

Another time when you will want to disable the echo is when reading in a program tape on the 'low-speed' reader. If you turn off the echo in this case you can avoid getting an unwanted listing while you relax to the rhythm of a quiet little 'burp,burp,burp' instead of a 'clackety clack clack'. Just hit CTRL/F at the end of the tape to turn on the echo again.

Similarly, when reading a data tape from the high-speed reader it is generally undesirable to have it all printed on the terminal. Thus the '0 R' command automatically disables the echo; but if you wanted to see what some of the data looked like, you could use an '0 R,E' command. To make a copy of a program or data tape you would first switch output to the punch and then turn on the echo to 'print' each character received on the output tape, e.g.

```
0 P;0 R,E;S FIND()
```

The 'FIND' function (described on page 62) keeps reading from the input device, looking for the character code specified. In this case a 'null' was used which will never be found, so the effect of this command is to continue reading until the end of the tape is reached at which point the terminal will automatically be restored as the I/O device, with the echo enabled. If only portions of a tape were to be copied you could use the FIND function to search for an appropriate character and then switch I/O back to the terminal yourself. You can use the ECHO option to skip sections of the tape by disabling the echo until you 'find' the right character and then turning it back on to copy some more.

THE LEADER / TRAILER OPTION

The 'T' option punches leader/trailer code (ASCII 200). This is convenient (but not essential) for separating output on paper tape, and somewhat more important when using an audio recorder since there is no visual indication of breaks in the data. Blank tape may also be used as 'leader' and both are ignored each time the reader is selected as the input device. However, after the first valid input character has been read these same codes are interpreted as the 'end-of-tape' and cause both input and output to be restored to the terminal. A 'backarrow' or 'underline' is also printed to indicate that the EOT was detected. This character serves the dual purpose of also removing any 'garbage' characters which might have been read after the last valid input.

The 'T' option can be used alone ('0 T') or in conjunction with another 'OPEN' command. The number of L/T codes punched is determined by an optional arithmetic expression following the letter 'T' (and separated by a space from it), with the previous specification being used as the default. The initial value is 512, which is about right for use with an audio recorder, but somewhat ridiculous for paper tape (over 4 feet of leader!). A value of 70 or so is more appropriate in this case. You can always just repeat the 'T' option to get a slightly longer leader if you want to: '0 T 100,T' will punch out 200 L/T codes but leave the default set at 100. Notice how this option was used in the example on page 45 for writing out all of the program buffer. The length specified by the 'T' option is also used by the 'PUNCH' command (see page 43).

K O N T R O L

This is an optional command which may be used to program the DR8-EA parallel I/O module. The 'K' command is used to set and clear individual bits in the output register while the FDIN function (described on page 65) is used to read the corresponding bits in the input register. These options are added by the initialization routine if Switch 7 is -UP- (see page 3).

The KONTROL command uses -positive- numbers to turn bits on, and -negative- numbers to turn them off. Each bit is directly controllable, independent of the setting of any of the others. Thus a 'K 1' command, for example, will turn on bit '1' without changing the state of any of the other 11 bits, while a 'K -1' command will turn it off again. In order for this scheme to work successfully the bits must be numbered from '1-12' rather than from '0-11' which is the usual convention. This is because '-0' is not distinguishable from '+0'. In fact, '0' is interpreted to mean 'clear all bits', so a 'K 0' command (or just 'K' since '0' is the default for all arithmetic expressions) can be used to quickly initialize this register.

More than one bit position can be set at a time, e.g. a command such as:

K 1,-2,3 will set bit 1, clear bit 2, and finally set bit 3

In this form, each operation occurs sequentially with perhaps 10 milliseconds or so between operations. This allows a command such as 'K 1,-1' to be used to generate a short pulse on line 1. If it is necessary for several signals to occur simultaneously, those operations can be enclosed in parentheses:

K 1,(2,3,4),-1 will set bit 1, then bits 2,3,4, then clear bit 1

Since for some purposes it is more convenient to be able to specify various bit combinations with a single arithmetic expression rather than setting and clearing each bit individually, a third mode of operation is also available. In this mode, the last 4 bits (bits '9-12') are set to the value of an expression preceded by an '=' sign. The remaining 8 bits are not changed. Thus a 'K,=5' command would

first clear all bits (the comma indicates a missing argument which is the same as '0'), then set bits '10' and '12' while clearing bits '9' and '11' (which were already clear in this case).

To summarize the 3 different forms of the KONTROL command:

K N,-N	Turns a single bit on or off N=0 turns -all- bits off
K (L,M,-N)	Performs all operations in parentheses simultaneously, instead of sequentially
K =N	Sets the 4 least-significant bits to the binary value of N; this form may not be used inside parentheses.

ERROR MESSAGES

UWF traps any 'illegal' operation such as division by zero or an unknown command and prints a peculiar little message to tell you what the problem was and where in the program it occured. If you type in the command: 'SET 2=3' for example, UWF will reply with:

'?07.44'

which is its way of telling you that you have something besides a variable on the left side of an '=' sign. To decode the error message you should look at the back cover of this manual (or the summary card) which lists all of the error diagnostics and their probable cause.

If this same error had occured while the program was running (i.e. not from a direct command), the error message would also indicate the line in the program containing the erroneous statement:

?07.44 @ 15.13

indiates 'operator missing or illegal use of an equal sign' in line 15.13.

The program 'QUITS' whenever an error occurs, thus all pending operations are cancelled and in general it is impossible to resume -precisely- at the point of interruption, but it is often possible to make the necessary changes, perhaps update a few variables with direct commands, and then restart from a point close to where the error occured.

This version also has an 'auto-restart' feature which allows the program to continue after an error instead of returning to command mode. This feature is selected by an option in the 'QUIT' command and is described in greater detail on page 35.

THE TRACE FEATURE

To further assist in finding the source of an error, UWF has a facility for printing out each program step as it tries to execute it. Thus when an error occurs you can see exactly where the problem is. The 'trace' feature is turned on by the occurrence of a '?' in the program (not one which is preceded by a single quote or enclosed in double quotes, however) and turned off again by another '?'. Thus only the portion of the program between pairs of question marks will be output while the program is running. The '?' may be given in a direct command, so to trace the entire program, just use a 'GO?' command to start it. Similarly a 'DO 5.2?' command will selectively trace that one line.

As a further aid to finding logical errors (as opposed to simple programing mistakes), when the trace is on, UWF will print out the result of all expressions appearing in SET commands. Thus you can see the values of all the variables as well as the program steps which created those values. A video terminal is obviously preferable for program traces since rather voluminous output can be generated in quite a short time.

A somewhat secondary use of the TRACE feature is for simplified input/output prompting. Whenever variables have names closely resembling their usage, it is a bit of a waste to have commands such as:

```
ASK "AGE? "AGE          or          TYPE "COST="COST
```

when, with only a small sacrifice in the punctuation, the following will do just as well:

```
ASK ?AGE ?              or          TYPE ?COST?
```

UWF will print out just the characters enclosed by the '?'s. For this reason it is preferable to use 'spaces' as separators rather than 'commas', i.e.

```
'ASK ?A B(I) C(J,K) ?'
```

will print out each variable name followed by a space and then wait for its value

to be input. One small disadvantage to this 'trick' is that when such statements are -actually- being traced, the text enclosed by '?' marks will -not- be printed due to the 'toggling' nature of the trace switch.

There is one other small anomoly associated with the trace feature: A command such as 'SET !=5,$=10' will not set those two 'secret variables' when it is traced, but will instead first perform a CR/LF and then dump the symbol table! This is because during a program trace all SET commands are treated internally as though they were 'TYPEs' and hence the secret variables take on their special roles as operators. There is a simple solution to this problem, however, and that is to simply prefix a '+' sign, or otherwise embed such variables in the midst of an arithmetic expression so that they are no longer recognized as ASK/TYPE operators. Thus the command 'SET +!=5,+$=10' would be traced properly.

COMMAND SUMMARY

The following table provides a quick review of UWF's entire command repertoire.

	FORM	EXAMPLE
@	not implemented in this version	
ASK	list of Variables, "prompts", formatting options	A X,Y(I),Z(J,K)
BREAK	line number	B or B 11.45
COMMENT	your programs whenever possible	C FOR COMMENTS
DO	list of lines, groups, or sub-groups	D .7, -9.5, 10
ERASE	line, group, sub-group, or 'all'	E 5 or E 9.1
FOR	Var = start, increment, finish	F I=1,5;F J=I,-1,0
GOTO	line number	G 11.8 or G .3
HESTATE	time delay desired	H 1000
IF	(Arithmetic expression) negative, zero, positive	I (K=I-J), .5
JUMP	line number	J .3;C WAIT LOOP
JUMP	(Arithmetic expression) one, two, three, four, ...	J (N) 1, .2, -3.4

KONTROL	bit positions	K 1,(-1,2,3),=X
LOOK	program area	L 1
LINK	program area, subroutine pointer	L 2,4.1 or L,10
MODIFY	line number	M 5.1
MOVE	old line number, new line number	M 3.3,6.3
NEXT	line number	F I=1,10;N;T PI
ON	(Arithmetic expression) negative, zero, positive	O (A-5) -9.2, 9
PUNCH	punches program and variables in binary format	P
QUIT	line number	Q or Q 5.1
RETURN	line number	R or R .2
SET	list of arithmetic expressions	S A=5, B=C=A/2
TYPE	arithmetic expressions, "labels", formatting	T !?A ?:10"B="B
U	available for user expansion	
V	available for user expansion	
WRITE	list of lines, groups, sub-groups, or 'all'	W or W -1.5,2,3.1
XECUTE	list of arithmetic expressions (same as SET)	X FSIN(#)/FCOS(#)
YNCR	list of variables	Y I-J,K L
ZERO	list of variables or 'all'	Z,#,A,B(I),C(J,K)

OPEN INPUT, ECHO	normal terminal input	O I,E
OPEN READER	selects high-speed reader	O R
OPEN PUNCH	selects high-speed punch	O P
OPEN OUTPUT	selects terminal for output	O O
OUTPUT TRAILER	punches leader/trailer code	O T or O T 70

INTERNAL FUNCTIONS

In spite of the fact that only about 3.3K words have been used to implement UWF, there are nearly 20 built-in functions and a facility for adding a limitless number of Programed Defined Functions.

The 'internal' functions provide the user with full-accuracy ('10-digit') approximations for commonly used relations such as log, exponential, sine, cosine, square root, etc. Also included are simple numerical functions such as absolute value, integer, sign and fractional parts, maximum/minimum, etc. And finally, there are a few functions for character processing and special I/O operations such as reading the Switch Register and loading the MQ. All function names in UWF begin with the letter 'F'; thus variables names may not begin with this letter.

TRANCENDENTAL FUNCTIONS

This class of functions, so named because the relations they represent can only be expressed as infinite series, includes the natural log and exponential functions and the three most common trignometric functions. The series approximations used by UWF have been optimized by a constrained least-squares procedure to reduce the error over the principal argument range to at worst a few parts in 10 billion.

The trancendental functions can be removed if you wish to increase the number of variables available in the 8K version. Removing them creates space for another 55 variables - a total of 175 instead of only 120. Program Defined Functions can be incorporated in their place at the expense of greater execution time and slightly poorer accuracy. See page 71 and Appendix II.

F L O G

FLOG(X) returns the natural logarithm of the absolute value of the argument. An error occurs if X is zero since the theorectical result is infinite. No error occurs if 'X' is negative, although the Log function is, in fact, only defined for positive arguments. This implementation facilitates the use of FLOG for extracting roots and raising values to non-integer powers. The Common (base-10) logarithm is easily obtained from the FLOG function just by dividing by FLOG(10). Example:

```
TYPE %,"NATURAL LN(PI)="FLOG(PI) :45"COMMON LOG(PI)="FLOG(PI)/FLOG(10)!

NATURAL LN(PI)=  1.144729886E+00       COMMON LOG(PI)=  4.971498727E-01
```

F E X P

FEXP(X) returns the value of e^X where 'e'= 2.718281828... The value of 'e' is always available as FEXP(1). This function is often used to extract roots and compute non-integer powers. For example, 'X^3.5' is found from the expression: FEXP(3.5*FLOG(X)). Similarly, the cube root of 27 may be found from the expression: FEXP(FLOG(27)/3). The absolute value of the argument must be less than approximately 1400 in order to avoid numeric overflow.

F S I N - F C O S

FSIN(A) and FCOS(A) return the value of the sine and cosine of the angle 'A' when 'A' is measured in -radians-. A 'radian' is a unit of angular measure preferred for scientific and engineering work because it eliminates factors of PI in many formulae. One radian is 1/2PI of a full circle, or approximately 60 degrees.

To convert angles from degrees to radians you simply multiply by PI/180. The value of 'PI' is a protected variable which is always available. Here is a short table of the values of FSIN and FCOS over the first quadrant as produced by the command shown. Notice how the radian value was saved for use in the second function call:

```
FOR A=0,10,90; TYPE %2,A %15.1, FSIN(R=A*PI/180), FCOS(R)!
```

```
 0      0.0000000000    1.0000000000
10      0.1736481776    0.9848077530
20      0.3420201433    0.9396926207
30      0.5000000001    0.8660254037
40      0.6427876096    0.7660444431
50      0.7660444431    0.6427876096
60      0.8660254037    0.5000000001
70      0.9396926207    0.3420201433
80      0.9848077530    0.1736481778
90      1.0000000000    0.0000000000
```

F T A N - F A T N

The Tangent function is not provided as an internal function since it is just the ratio of FSIN/FCOS and is thus easy enough to compute. The user may implement his own FTAN function, however, as described in the discussion of Program Defined Functions on page 67.

The inverse ('arc-') tangent function is available, however. FATN accepts values of any magnitude and returns the -angle- (in radians) which would give that tangent. The range of answers is from -PI/2 (-90 degrees) to +PI/2 (+90 degrees). To convert from radians to degrees, just multiply by 180/PI. For example, to check that the angle whose tangent is -1 is, in fact, -45 degrees:

```
TYPE 180*FATN(-1)/PI          -45.00000000
```

All other trig functions can be derived from these primary functions. For example:

Cotangent	FCOS(A)/FSIN(A)
Arcsine	FATN(A/FSQT(1-A*A))
Arccosine	FATN(FSQT(1-A*A)/A)
Hyperbolic sine	(FEXP(A)-FEXP(-A))/2
Hyperbolic cosine	(FEXP(A)+FEXP(-A))/2

Consult any advanced Algebra book for other such identities.

- -

F S Q T

- -

The FSQT function computes the square root of the argument using an iterative approximation which guarantees that no more than the last bit will be in error. Example: TYPE FSQT(2), FSQT(2)^2!

1.414213562 2.000000000

- -

F A B S

- -

FABS returns the absolute value of the argument: TYPE FABS(-1), FABS(1)

1.00000000 1.000000000

- -

F S G N

- -

FSGN returns -1, 0 or +1 depending upon whether the argument was: negative, zero or positive.

Example: TYPE FSGN(PI), FSGN(PI-PI), FSGN(-PI)

1.000000000 0.000000000 -1.000000000

- -

F I T R

- -

FITR returns the InTegeR part of the argument. Thus 'FITR(PI)' is '3' and 'FITR(-5.5)' is '-5'. Note that some languages have an 'entier' function which is the 'integer less than or equal to the argument'. For positive numbers this produces the same result as UWF's FITR function, but for negative values it gives the next lowest number. If you are converting a program which was originally written in another language, be sure to watch for this subtlety! It should be noted that many functions and commands in UWF convert values to an integer form internally without requiring the programer to do so. Subscripts, for example, are always used in integer form, meaning that 'A(1.5)' is legal, but is no different from 'A(1)'. In general, a value which is used as an index or is stored in a hardware register is always converted to an integer before use.

- -

F R A C

- -

FRAC returns the fractional part of a number - the part which FITR discards! This may be used to do 'modulo-N' arithmetic or to check for a remainder. The user is cautioned, however, that the value returned by FRAC may have only limited accuracy and hence checks for 'exact' values computed from expressions containing the FRAC function should generally be avoided. To illustrate, the fractional value of '.002' is .002, but the fractional value of 1.002 is off in the 8th place while that of 1000000.002 is only correct to 3 digits. This is simply the result of taking the difference between two large numbers.

F M I N - F M A X

These functions compare two arguments, returning the algebraically smallest or largest value. Thus 'FMIN(+1,-2) would return '-2' while FMAX would return '+1'. These functions have several uses. A simple example in connection with the FLOG function allows one to avoid the 'log-of-zero' error with a call such as 'FLOG(FMAX(1E-10,X)). Similarly, the FMIN function can be used to avoid typing non-existent values when dumping an array in a multi-column format. In this example, 'C' is the number of columns and 'N' the number of data values in the array:

```
FOR I=1,C,N; FOR J=I,FMIN(N,C+I-1); TYPE Q(J); NEXT; TYPE !
```

As a final example, an entire array can be scanned for its extrema simply by comparing each element with the previous best estimates:

```
SET MIN=MAX=A(1); FOR I=2,N; SET MIN=FMIN(A(I),MIN), MAX=FMAX(A(I),MAX)
```

A disadvantage of this method for locating the extremes is that no information is available as to which element is biggest or smallest, only the values are returned.

F R A N

The FRAN function returns a different pseudo-random number each time it is called. The numbers are limited to the range 0-1 and have an approximately 'flat' distribution. Other distributions, for instance Gaussian or Lorentzian functions, can be created as Program Defined Functions by using FRAN in an appropriate expression. The function is initialized by the input wait loop so the values you observe will appear to be truly random. The pair-wise and higher-order groupings do have a small correlation coefficient, but even so, a reasonable value of PI can be obtained using FRAN to generate a 2-dimensional scatter pattern. The principle use of FRAN appears to be for games.

CHARACTER and I/O FUNCTIONS

The remaining internal functions handle character manipulation and other special-purpose I/O operations. The character functions include FIN, FOUT, FIND and FTRM, while FSR, FMQ and FDIN are 'I/O-type' functions. FBUF and FCOM provide access to extended memory for storing large data arrays.

F I N

The FIN function reads a single character from the Input Device and returns the numerical value of that character. A list of character values may be found in Appendix I and the value of any character can be obtained within the program simply by preceding it with a single quote mark. Thus the expression ('A) will have the value of the letter 'A' (193) while ('A-'Z) will be the difference of the codes for 'A' and 'Z'. Character strings can be read with the FIN function and later output with FOUT; this is a bit slow, but does provide UWF with a limited string-handling facility.

F O U T

The FOUT function generates a single character from the value of an arithmetic expression. It will thus output what FIN has input: 'FOUT(193)' will generate the letter 'A'. More commonly, however, FOUT is used to output special control characters which would otherwise be invisible if they were simply included in a 'TYPE "..."' command. For instance, 'FOUT(7)' is used to ring the 'bell', while 'FOUT(140)' outputs a 'form-feed' character. 'FOUT(27)' generates an ESCAPE code which is used by many terminals to initiate special functions such as reverse video, cursor movement, etc.

FOUT expects arguments in the range 0-255; values beyond this range will be output, but should be avoided. Most terminals respond in the same way to values in the range 0-127 and 128-255. UWF's input routines, however, always return values in the higher range (128-255) in keeping with the standard established for the PCM-12.

The value returned by FOUT is always -zero-, not the value of the character code! This was done to simplify calling the function as part of a command. For instance, you can output a formfeed ahead of a program listing by using a 'WRITE FOUT(12)' command instead of just 'WRITE'. Similarly, since 'tabbing' to column zero is ignored, you can include FOUT's in ASK or TYPE commands just by putting them in 'tab expressions'. To print a 'double quote' mark, for instance, you could use the following:

TYPE "THIS IS A ":FOUT('")" MARK!" which will produce THIS IS A " MARK!

F I N D

FIND searches for a character equal to its argument, reading and echoing all characters it encounters until it finds a match. The echo is controlled by the setting of the input echo switch, as described earlier on page 46. The character which matches is -not- echoed, however, but is returned as the value of the function. To output this character too, you may use a call such as 'S FOUT(FIND('A))' where 'A' is the search character. To read in a comment line, just search for a Carriage Return: SET FIND(141). To read the same line in from a paper tape, however, you should search for the Linefeed following the CR: SET FIND(138). This is due to different conventions for the 'end-of-line' character. FIND also checks continually for a CTRL/Z. This is recognized as an 'End-of-File' mark and causes FIND to return with the value 'zero' instead of with the value of the search character.

F T R M

As discussed earlier on page 25, the ASK command treats any input other than '0-9' and 'A-Z' as a terminator, which means that data values may be conveniently 'flagged' by the use of a special terminating character. The purpose of the FTRM function is to then pass this information back to the program so that special action may be taken if necessary. For instance, a program might need to be able to work with either metric or English measurements, using an appropriate terminator to differentiate between them. Similarly one can devise a 'pocket calculator' program which accepts numbers terminated by one of the arithmetic operators and then performs the indicated function. One of the more common uses for this feature is to permit an indefinite number of data values to be read in, sensing a special terminator for the last value. A loop like the one in the example below (which checks for a '?') is all that is required:

```
4.1 ZERO N;TYPE "ENTER QUIZ GRADES, TERMINATE THE LAST ONE WITH A '?'"!
4.2 ASK G(N=N+1); IF (FTRM()-'?) .2,,.2; TYPE %2"THERE WERE"N "GRADES"!
```

F B U F - F C O M

These functions allow UWF to use extra memory for data storage and are thus of interest only for systems with more that 12K. They may be added by setting Switch 8 -UP- when UWF is started for the first time (see page 3). FBUF is designed to handle 12-bit (signed) integer data while FCOM may be used for storing either 24-bit integers or 48-bit floating-point values. Both functions are called in the same manner: the first argument specifies the relative location in the storage area and the second argument (if any) is the value to be stored at that location. The function always returns the value at the location specified. Thus:

FCOM(I)	returns the 'Ith' value in the 'FCOM' area
FBUF(I,V)	stores the value of 'V' in the 'Ith' location

The range of the index is typically 0-4095 for FBUF and 0-1023 for FCOM. FCOM has another mode however, in which data is stored as two-word integers (rather than four-word floating point values) thereby doubling the amount of storage available but limiting the range of the data to +/- 2^23. To use FCOM in this mode, specify a -negative- index (legal range is -1 to -2048). Here is a loop which stores the square root of all numbers from 0-1023:

```
FOR I=0,1023; SET FCOM(I,FSQT(I))
```

Although FBUF and FCOM share the same field, FBUF starts from the 'bottom up' while FCOM stores from the 'top down', so both functions may be used simultaneously. Furthermore both functions are fully recursive, so calls such as 'FCOM(I,FCOM(J)) may be used to move data from one location to another.

F S R

The FSR function reads the value of the Switch Register. This may be used to control program options. The value is treated as a signed number so the range is from -2048 (4000 octal) to +2047 (3777 octal).

F M Q

The FMQ function displays the integer part of the argument in the MQ register. This is quite handy for 'spying' on the progress of a long calculation simply by displaying the value of a loop index. Since FMQ returns the integer part of the argument, it can be included in a subscript expression, such as 'A(FMQ(I))' which is functionally the same as 'A(I)' but also displays the index in the MQ.

F D I N

This is an optional function for reading the input register of a DR8-EA parallel I/O module. It may be added (along with the 'KONTROL' command) by setting Switch 7 -UP- the first time UWF is started. The interface may be wired to respond to either levels or pusles, the difference being that it will 'remember' a pulse, but 'forget' when a level changes. Each bit is separately addressable, and each may be wired for pulse or level sensing. For use with the FDIN ('Digital INput') function, the bits are considered to be numbered from 1-12 (rather than from 0-11), just as they are for the 'KONTROL' command (page 49).

The value of 'FDIN(0)' (or just 'FDIN()' since 'zero' is always the default value of an argument) is simply the weighted sum of all input bits which have been 'set'. Bit '1' has the value 2048, bit '2' 'weighs' 1024, etc. The maximum value is thus '4095' if all the bits are turned on. Any bits which are read by the FDIN function will be reset if they are resettable, i.e. if they are wired for 'pulse' input. This ensures that only one occurrence of an event will be detected by the program.

FDIN can be made to respond to only a single bit, or to a collection of bits, by including various arguments as the programmer desires. For instance, 'FDIN(1)' will only sense the state of bit '1'. If bit 1 is on, FDIN will have the value 2048, while if it is off, the value '0' will be returned, regardless of the setting of any other bits. Furthermore only bit 1 will be reset. The value of 'FDIN(-1)' on the other hand, will be the status of all bits -except- bit 1, i.e. bits 2-12. Any bits which are read will be reset as described above.

More complicated masks can be constructed by specifying multiple bits. Thus 'FDIN(1,3)' will only look at bits '1' and '3', while 'FDIN(-2,-5)' will look at -all but- bits 2 and 5, etc.

PROGRAM DEFINED FUNCTIONS

UWF allows the user to define his own set of special functions within the program. Such 'Program Defined Functions' ('PDFs') may consist of any set of UWF commands, ranging from a single program step to as much as an entire group. A PDF is very similar to an ordinary subroutine ('DO') call, but with 3 important differences:

1) a PDF may pass arguments to the subroutine
2) a PDF returns a numeric value - the value of the function
3) a PDF may occur in any command, not just DO, CN, LINK, etc.

The last difference is especially important since it allows subroutine calls in some circumstances when they might not otherwise be possible.

The form of a PDF call is:

F(line number, argument list)

where the letter 'F' identifies this as a function call and the line (or group) number identifies the function. This number can be replaced by a suitably chosen variable so that one may use a 'named' function call rather than a 'numeric' one (v.i.). The argument list is not required, but may contain several arguments. Typically, only 1 or 2 are used although this is not a fundamental restriction. The arguments may consist of other PDF calls which do not themselves have arguments, or any other internal functions, with or without arguments. The use of nested PDF calls containing an argument list is restricted since the arguments are not stored recursively. Here are few examples of Program Defined Functions:

F(2,A*B)	Calls Group 2, passing 'A*B' as the argument
F(.9,X,Y)	Calls line XX.90 in the current group
F(-9.5)	Calls sub-group at line 9.5 with no arguments

Coding a PDF is no different from writing an ordinary subroutine, but the mechanism for passing argument values and returning the function result needs to be explained. The value of each arithmetic expression appearing in the argument list is saved in a specific 'protected variable'. The first argument is saved in the variable '#', the second one in the variable '$', and the third in the variable '%'. Additional arguments are possible, and if necessary more protected variables should be defined when initializing UWF (see page 3). The ordinary variables created by the program may also be used as 'common' variables (those appearing in both the 'main' program and the definition of the function) for passing information to the subroutine.

PDF calls are not required to always have the same number of arguments, so infrequently used parameters can be placed after frequently used ones. These will not be changed unless they are modified by the subroutine itself. In the first example, the value of 'A-times-B' is placed in the variable '#'. In the second example, 'X' is placed in '#', and 'Y' goes into '$'. If this function were called subsequently with only a single argument, the value placed in '$' would not be disturbed. No arguments are used in the third example, but any variables defined by the program may be used by the subroutine. This is the only reasonable way to handle arrays.

The subroutine must then be written to use the appropriate protected variable whenever it needs the value of the corresponding argument. A routine to compute the length of a vector, for instance, might use an expression such as 'FSQT(#*#+$*$)'.

The value returned by the function is just the result of the last arithmetic expression processed by the subroutine. This expression may be evaluated by any suitable command, but typically the SET command is employed. To begin with a very simple example, here is how you could code the tangent function:

```
9.9 SET FSIN(#)/FCOS(#); COMMENT: THIS IS THE TANGENT FUNCTION
```

You could also include a replacement operator to save the result in a variable, or you could use the TYPE command to output the result of the expression, or whatever. Since it is the -last- result which is returned as the value of the function, however, if other calculations are necessary for checking the result or

performing ancilliary calculations, the value desired must be saved and then 'SET' again just before returning.

There are a number of UWF commands which do not disturb a PDF result and so may be used without caution in the definition of the function. These are COMMENT, RETURN, YNCREMENT and ZERO. On the other hand, branching commands always evaluate a line number (which may be zero), and so cannot be used to terminate a PDF without destroying the (expected) function result. It should also be pointed out that the line number option in a RETURN command (see page 31) will be ignored by a PDF call. This is necessary to ensure that the program returns to complete the function call.

While most PDF calls just use an explicit line or group number to identify the function, it is possible to be somewhat more elegant! By using a variable with a nicely selected name you can specify the 'F(TAN,X)' function rather than the 'F(9.9)' function. To do this, just set the variable 'TAN' to the value 9.9. This has the additional advantage that you can easily move the subroutine to a different part of the program without having to change all the function calls.

EXAMPLES OF PROGRAM DEFINED FUNCTIONS

Here are a few interesting PDF's which illustrate some of the things you can do. A symbolic name has been used in most cases; it must be set to the value of the line or group actually used to code the function.

1) F(PWR,X,Y) - raises 'X' to the 'Y' power when 'Y' is non-integer:

```
SET FEXP($*FLOG(#))
```

Sample call: TYPE F(PWR,27,1/3) 3.000000000

2) F(!,N) - computes the Nth factorial (maximum value of N is about 300)

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

Sample call: TYPE F(!,5) 120.0000000

3) F(SUM) - computes the sum of the subscripted array 'G(I)'

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

Sample call: SET AVE=F(SUM)/N

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

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

This function is useful for computing series approximations

5) F(OCTAL,VALUE) - converts a number from decimal to octal

```
FOR I=N=0,4;SET N=N+(#-8*#=FITR(#/8))*10^I
```

```
Sample call:  TYPE F(OCTAL,1000)                1750
```

This is the most interesting of the functions shown so far, if for no other reason than that it uses all the arithmetic operators in a single SET command as well as some fancy redefinitions within the loop. The technique employed is quite general for changing from one number base to another, so simply by interchanging the '8's' and '10's' in the definition you can construct a function to give you the decimal equivalent of an octal number:

```
TYPE F(DECIMAL,1000)                512
```

To be still more elegant you can rewrite the function to use the value of '$' in place of the number '8' shown above and thus have a general-purpose routine for converting to any number base (less than or equal to 10). A fun thing to do once you have made this change, is to try it out with a direct command such as:

```
FOR J=2,10; TYPE F(BASE, 99, J)!
```

which will then type out the value of 'ninty-nine' in all number bases from 2-10. The loop limit represents the maximum number of digits required to represent the number, so if you try this with large numbers and small number bases you will probably need to increase the limit to something more than '4'.

6) PDF replacements for the trancendental functions:

These functions may be used in place of the internal functions in the event that you wish to delete some of them to increase the number of variables available on an 8K machine.

```
F(EXP)=        25.1 IF (#*#-.01).2; SET #=F(EXP,#/2)^2
EXP=25.1       25.2 SET #=1+#+#*#/2+#^3/6+#^4/24+#^5/120

F(LOG)=        26.1 IF (#*#-2.04*#+1).2; SET #=2*F(LOG,FSQT(#))
LOG=26.1       26.2 SET #=(#-1)/(#+1), #=2*(#+#^3/3+#^5/5+#^7/7)

F(ATN)=        27.1 IF (#*#-.01).2; SET #=2*F(ATN,#/(1+FSQT(1+#*#)))
ATN=27.1       27.2 SET #=#-#^3/3+#^5/5-#^7/7

F(SIN)=        28.1 IF(#*#-.01).2; SET #=F(SIN,#/3), #=3*#-4*#^3
SIN=28.1       28.2 SET #=#-#^3/6+#^5/120
F(COS)=        28.3 SET F(SIN, PI/2-#)

F(TAN)=        29.1 IF (#*#-.01).2;S #=F(TAN,#/2), #=2*#/(1-#*#+1E-99)
TAN=29.1       29.2 SET #=#+#^3/3+#^5/7.5+#^7/315

F(ASIN)=       30.1 IF (#*#-.01).2;S #=2*F(.1,#/(FSQT(1+#)+FSQT(1-#)))
ASIN=30.1      30.2 SET #=#+#^3/6+.075*#^5+#^7/22.4
F(ACOS)=       30.3 SET PI/2-F(ASIN)

F(HSIN)=       31.1 IF (#*#-.01).2; SET #=F(HSIN,#/3), #=3*#+4*#^3
HSIN=31.1      31.2 SET #=#+#^3/6+#^5/120
F(HCOS)=       31.3 SET FSQT(F(HSIN)*#+1)
```

The method used in these functions is to recursively reduce the argument to a value typically less than .1, evaluate a series approximation which is reasonably accurate for an argument of this magnitude, and then 'bootstrap' back using an identity such as 'e^2X=(e^X)^2'. Thus the approximation for F(EXP) is evaluated after reducing the argument to the proper range and then the result is squared enough times to return to the original value. This clever method was devised by A.K. Head.

7) In many cases a PDF call is preferable to a simple 'DO' because it can pass a parameter or two to the subroutine at the same time and can also return a 'status' value. As an example of such a use, consider a subroutine for finding the roots of a quadratic equation. There are three possible cases: the roots are equal, the roots are real, but unequal, or the roots are complex numbers. If the values produced by the subroutine are stored in 'R1' and 'R2', then after calling the routine one must still decide how to interpret the results. If the subroutine were to return the value of the 'discriminant' this could be accomplished as follows:

```
ON (F(QR)) complex, equal, unequal
```

where 'QR' is the group number of the Quadratic Root subroutine, and 'complex', 'equal', 'unequal' are line or group numbers associated with the 'ON' command which serves both to call the subroutine and to test the result at the end. Other such examples will undoubtedly occur to the reader.

FUNCTION SUMMARY

Here is a list of all the functions implemented in the standard version of UWF. Since up to 36 internal functions are possible, it should be clear that this list is not exhaustive.

FABS	Returns the absolute value of the argument
FATN	Returns the angle in radians whose tangent is given
FBUF	Optional: stores or retrieves 12-bit signed integers
FCOM	Optional: accesses additional memory for data storage
FCOS	Returns the cosine of an angle measured in radians
FDIN	Optional: returns value of digital input register
FEXP	Returns value of 'e^X' where |X| is less than 141&
FIN	Reads and returns the value of a single character
FIND	Searches for a given character code
FITR	Returns integer value of the argument
FLOG	Returns the natural logarithm of the argument
FMAX	Returns the maximum value of two arguments
FMIN	Returns the minimum value of two arguments
FMQ	Displays the argument in the MQ, returns same
FOUT	Outputs a single character value
FRAC	Returns the fractional part of the argument
FRAN	Returns a random number in the range 0-1
FSGN	Returns the sign value of the argument: -1,0,+1
FSIN	Returns the sine of an angle measured in radians
FSQT	Returns the square root of a positive number
FSR	Returns the signed value of the switch register
FTRM	Returns the value of the last ASK terminator
F	Program Defined Functions

DECIMAL VALUES FOR ALL CHARACTER CODES

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

FOUT(141) will output a RETURN/LINEFEED while FOUT(13) will just do a RETURN. Codes 225 through 255 are lower case letters, some of which serve other functions on keyboards without lower case. Many keyboards use 'SHIFT/K' for '[', 'SHIFT/L' for '\' and 'SHIFT/M' for ']' and corresponding combinations for the control codes following 'CTRL/Z'. These symbols are often not printed on the keytops. Codes 0-127 are the same as codes 128-255 except for the parity bit. UWF always forces the parity bit during input.

ERROR CODES FOR UWF (V4E) OCTOBER 1978

?	Keyboard interrupt (CTRL/F) or restart from location 10200
?01.50	Group number greater than 31
?01.93	Non-existant line number in a MODIFY or MOVE command
?03.10	Non-existant line called by GOTO, IF, NEXT, BREAK or QUIT
?03.30	Illegal command
?03.47	Non-existent line or group: DO, ON, JUMP, LINK or PDF call
?04.35	Missing or illegal terminator in a FOR command
?06.03	Illegal use of a function or number: ASK, YNCR or ZERO
?06.41	Too many variables (ZERO unnecessary ones to recover space)
?07.44	Operator missing or illegal use of an equal sign
?07.67	Variable name begins with 'F' or improper function call
?07.76	Double operators or an unknown function
?08.10	Parentheses don't match
?10.50	Program too large (sorry, you'll have to erase some of it)
?18.32	FCOM index out of range
?19.72	Logarithm of zero
?21.57	Square root of a negative number
?22.65	More than 10 digits in a number
?25.02	Stack overflow: reduce nested subroutines and expressions
?27.90	Zero divisor
?31.<7	Non-existant program area called by LOOK or LINK
← or _	End of input sensed, I/O switched back to the terminal

A P P E N D I X I I

Here is a list of patches for adding a number of special features to UWF. They are shown in the format: FLLLL/ CCCC PPPP; QQQQ where 'FLLLL' is the Field + Memory location, 'CCCC' is the original contents, and 'PPPP' is the patch. In cases where several successive locations are to be changed, a semicolon is shown, followed by the next patch 'QQQQ'. Note that the 'FCOM' patch shown below is for 16K versions only and must be added -before- UWF is started the first time.

FIELD 0

00045/ 4463 4442 Replace extra variable storage with FCOM (16K only - see page 3)
00061/ 7610 6213 Print a CR/LF before printing an error message

FIELD 1

10402/ 4547 0000 Eliminate the line number printout in MODIFY

11216/ 7000 4533 Make the ASK command print a ':' each time
11241/ 1377 7040 Use the '#' operator to output a Form Feed

12471/ 1000 1177; 4533 Change 'rubout' for video terminals
13070/ 7106 7107 Increase the delay after a Carriage Return

13134/ 7000 6xxx Clear an unwanted interrupt (next 3 locations too)

15665/ 1103 1213 Make TYPE print an '=' ahead of each value
15666/ 4534 7200 Remove the initial space (or '=') printed by TYPE

14503/ 62X1 62Y1 Change the data field used by FCOM ('X,Y' may be 2-7)
14545/ 62X1 62Y1 Ditto for the FBUF function ('X' is set at startup)

10033/ 4566 5200 Remove the FLOG, FEXP and FATN functions to increase the
12371/ 5020 1754; 1754; 1754 size of the symbol table in the 8K version.

10033/ 5200 5303 Remove FSIN and FCOS to increase the symbol table size a
12367/ 5205 1754; 1754 little bit more (8K only).

NOTES

ERROR CODES FOR UWF (V4E) OCTOBER 1978

?	Keyboard interrupt (CTRL/F) or restart from location 10200
?01.50	Group number greater than 31
?01.93	Non-existant line number in a MODIFY or MOVE command
?03.10	Non-existant line called by GOTO, IF, NEXT, BREAK or QUIT
?03.30	Illegal command
?03.47	Non-existent line or group: DO, ON, JUMP, LINK or PDF call
?04.35	Missing or illegal terminator in a FOR command
?06.03	Illegal use of a function or number: ASK, YNCR, or ZERO
?06.41	Too many variables (ZERO unnecessary ones to recover space)
?07.44	Operator missing or illegal use of an equal sign
?07.67	Variable name begins with 'F' or improper function call
?07.76	Double operators or an unknown function
?08.10	Parentheses don't match
?10.50	Program too large (sorry, you'll have to erase some of it)
?18.32	FCOM index out of range
?19.72	Logarithm of zero
?21.57	Square root of a negative number
?22.65	More than 10 digits in a number
?25.02	Stack overflow: reduce nested subroutines and expressions
?27.90	Zero divisor
?31.<7	Non-existant program area called by LOOK or LINK
← or _	End of input sensed, I/O switched back to the terminal

F P A L

FPAL allows the user to code short 'machine language' functions directly into his program. This provides 'keyboard control' of special devices which are not supported by any of the normal functions or commands, and also permits operations requiring only 12-bit arithmetic to proceed at full machine speed. Routines as long as 32(10) instructions can (in theory) be incorporated, but in practice, FPAL routines are seldom longer than about 5-10 instructions - just enough to execute a short sequence of IOTs to pulse a control line, for instance.

The form of the function call is: FPAL(AC,inst,inst,inst...), where 'AC' is an arithmetic expression, the value of which will be placed in the AC prior to calling the routine, and the remaining arguments are construed as a list of -octal- numbers which represent the desired machine instructions. These are stored in Field 1 such that the first instruction is at 'page+1', the second at 'page+2', etc. After the last instruction has been tucked away, FPAL loads the AC with the integer part of the first argument, clears the Link, and calls the routine. The final value of the AC is then returned to the program as the value of the function. Note that the user does not have to worry about any of the 'calling' instructions - he only has to write the essential machine code.

Here are a few examples which may help clarify how the FPAL function works and illustrate some of the things it can do:

Ex. 1: UWF has an 'FMQ' function for loading a number into the MQ register (where it is preserved by all internal arithmetic operations), but no corresponding function for finding out what is already there. The following FPAL function will not only do this, but will also increment the value in the MQ at the same time:

```
TYPE MQ=FPAL(,7501,7001,7521)
```

Note that the first argument has been omitted in this example, since no information is being passed -to- the function. The first instruction (7501=MQA) reads the MQ, the next (7001=IAC) increments this value and the third (7521=SWP) interchanges the

new and old values, saving the new value for a subsequent call, and returning the old value to the program. Machines based on the 6100 microprocessor may not be able to display the MQ while UWF is running. Using this function however, the value of the hardware register can be saved in the variable 'MQ', and output by the 'TYPE' command as well. So being able to actually 'see' this register is not a necessity.

Ex. 2: Several variations of this routine come to mind almost immediately. For instance, we could use the hardware 'SWP' instruction to interchange two values:

```
SET MQ=FPAL(AC,7521)
```

or we could take advantage of the 'MQA' instruction to perform an 'inclusive OR' between a value in the MQ and one in the AC: SET FMQ(A),AB=FPAL(B,7501).

Ex. 3: As a final example, suppose that we have constructed an A/D converter interface which uses the same instruction set as the AD8-EA. In order to test it out we can use the following FPAL routine to implement the 'FADC' function:

```
SET CH(N)=FPAL(N,6531,6532,6534,5203,6533)
```

The channel number ('N') will be placed in the AC at the beginning and can be used to control the multiplexer via a '6531' (=ADLM) instruction. The converter is then started (6532=ADST) and we begin testing the 'done' flag (6534=ADSD) to see when it is finished. This involves a 'JMP .-1' instruction which means that the location of the 'ADSD' instruction (relative to a page boundary) must be known. Since FPAL routines always start at 'page+1', (location 'page+0' can be used as a 'temporary'), a jump to the -third- instruction becomes '5203'. When the conversion is finally done the result is read into the AC (6533=ADRD), and returned to the program.

It goes almost without saying, that such easy access to machine-level code is both powerful - and - dangerous! No error checking can be performed, so a single 'typo' can lead to instant disaster! Always be sure, therefore, to save a copy of a valuable program -before- you try out any sort of 'wild' FPAL function, and be especially careful with ISZs, DCAs, JMPs and JMSs since they can modify memory or send the program off 'into the wild blue yonder'.

Similarly, give special consideration to any IOT which might cause a hardware interrupt since UWF runs with the interrupt system enabled! Most interfaces have an 'interrupt disable' instruction, but if it is necessary to use an 'IOF' in order to protect UWF from a spurious interrupt, be sure to clear the flag and then issue an 'ION' before exiting from the function - otherwise it may be necessary to restart the interpreter in order to activate the interrupt system again (see page 2).

ADVANCED CONSIDERATIONS

While it is clearly possible to use FPAL to implement patches to UWF itself, this practice is -strongly- discouraged (and no help with such folly will be offered) since this makes programs 'version dependent'. On the other hand, there -are- a few 'tricks' which could prove useful at various times:

1) The value of the first parameter is actually converted into a 24-bit integer, of which only the lower 12-bits are loaded into the AC at the beginning of the routine. This means that the values '4095' and '-1' will both load '7777'(8) into the AC. The high-order part of the number can be accessed with a 'TAD 45' (1045) instruction, while the low-order half can always be recalled with a 'TAD 46' (1046) if it is needed later on in the function.

2) The value of the AC is normally returned as a signed number; if it is more desirable to have an 'unsigned' result you can simply code an 'ISZ .+1' instruction as the last step of the routine. Thus: 'TYPE FPAL(4095)' will return '-1', whereas 'TYPE FPAL(4095,2202)' will return '4095'. The '2202' instruction is 'ISZ .+1' when located at 'page+1'.

Notice that numbers appearing in the -first- argument of an FPAL call are treated as 'decimal' values and can be replaced by variables and/or other functions. The remaining arguments, however, are processed as character strings and so cannot be replaced by arithmetic expressions.