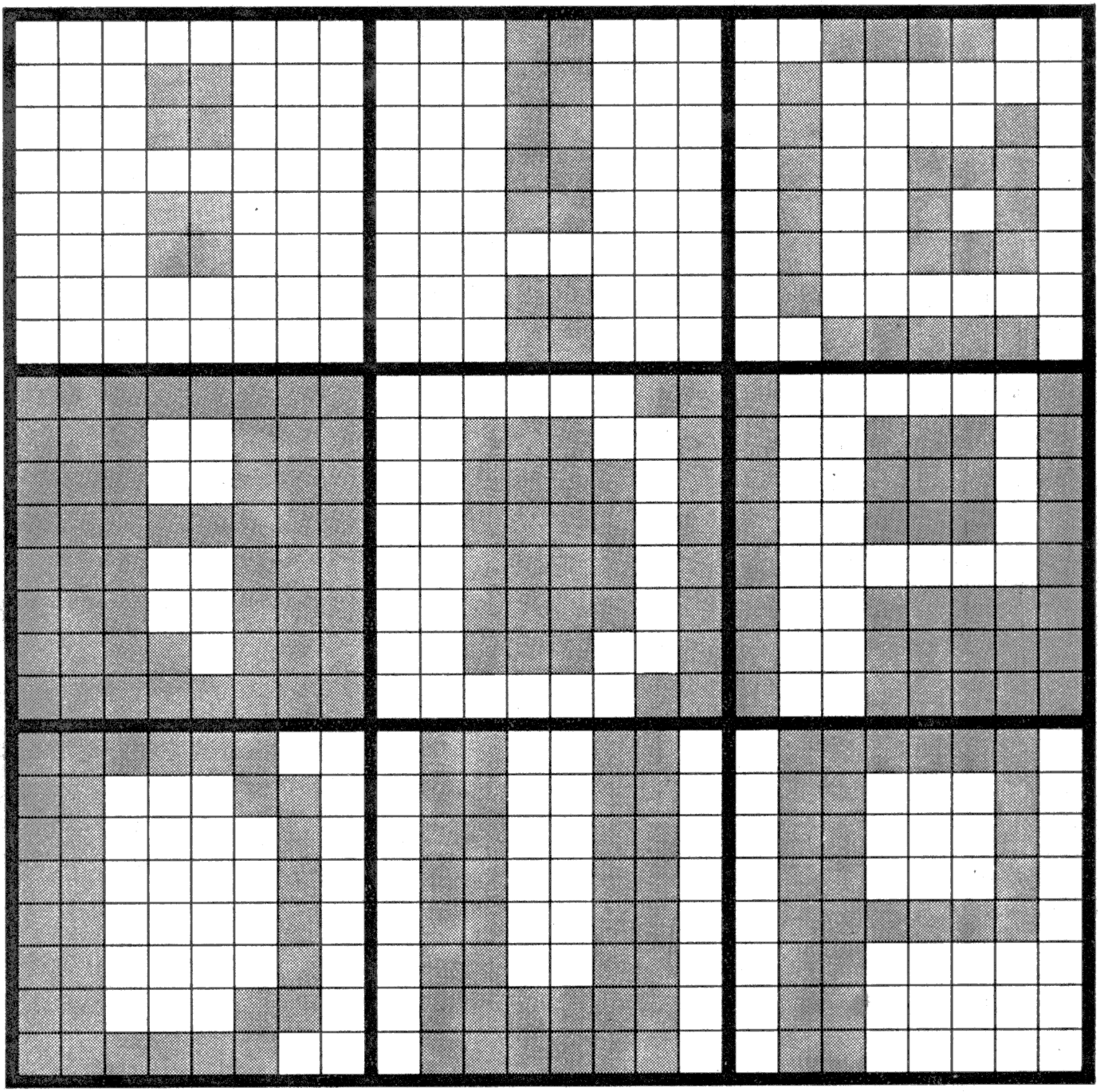
User manual Multi-Forth 83



BBC Micro

Skywave SOFTWARE

Table of Contents

1.1 Introduction to Multi-FORTH 83 on the BBC

This Manual is intended as a guide to the use of Multi-FORTH 83, and assumes that the user will read it in conjunction with a book on FORTH. We recommend the book "The Complete FORTH" by Alan Winfield, which is available from most booksellers or in case of difficulty from us for £6.95 + £1.00 postage and packing.

Multi-FORTH 83 matches the FORTH 83 Standard and also contains a number of non-standard words to accomplish Multi-tasking and the interface to the BBC machine.

Multi-FORTH 83 is Multi-tasking. This gives the programmer the ability to write real-time routines as is described in Section 19

The language of FORTH is a structured programming language which allows the user to manage and manipulate all of the memory addressable by the microprocessor. FORTH is also a language in which the user can link down to machine code routines and in this respect, FORTH is only a step above assembly level programming.

Multi-FORTH 83 contains a full Assembler in a separate Assembler Vocabulary for the user to define words in machine code if so desired. FORTH is however, a high-level user friendly language in that it allows the user to create his own command set. The entire program set written in FORTH is a customised set of instructions and in this way approaches other high-level languages.

In addition to the standard attributes of the FORTH language, Multi-FORTH 83 adds extra flexibility with its multi-tasking. Multi-tasking allows the user to multi-plex programs so as to appear that they are executing simultaneously, much the same as a TV picture is built up out of a rapidly moving dot so as to give the illusion of a continuous picture. This is a feature previously available only on much more expensive systems. With the proper hardware, such as plenty of I/O, a multi-tasking system can be used as a real-time controller and data collection system. This means that the computer can operate at a speed sufficient enough to control the environment as events occur.

A multi-tasking system could be used to enter data from a real-time environment. An example would be sampling the breathing cycle of a patient in a hospital in order to determine his or her respiratory rate. A multi-tasking system also gives the user the flexibility of allowing a program to run in the background (it is possible to run one or more programs in the background while editing another in the foreground).

FORTH is somewhat harder to learn than BASIC, however the flexibility and speed gained with FORTH makes it a desirable programming language for most, if not all, programming tasks.

Multi-FORTH 83 on the BBC is a paged-ROM which resides at 8000 Hex. Because of the way in which the BBC´s operating system (the MOS, Machine Operating System) handles the paged-ROM area, it is possible to allocate a priority to the ROM so that if desired, the machine can come up in Multi-FORTH 83 rather than BASIC when first turned on. See Section 4, Installation and System Description.

It is also possible to go to a BASIC Cold Start at any time by executing *BASIC and you can even use all the DFS Commands by entering something like *CAT to look at the disc directory.

While in Multi-FORTH 83, the "Break" key can be used to perform a Cold or a Warm Start as desired, and the "Escape" key is interrupt driven so as to allow an exit from an infinite loop or other problem.

1.2 FIG UK

FIG UK is the United Kingdom branch of the FORTH Interest Group. This group originated in the USA in the late seventies because a group of FORTH enthusiasts wanted to make the language available on home computers at a time when FORTH was only running on mini-computers such as the PDP-8. The charter of the group was to translate a common model of FORTH into assembly language listings for each computer. It was agreed that the group's work would be distributed in the public domain by FIG.

This work has been carried on in England by the FIG UK who meet regularly at the Polytechnic of the Southbank in London. A newsletter is published bi-monthly and you can be assured of a friendly welcome at their meetings.

Further details of the Forth Interest Group (UK) can be obtained from :-

c/o Honorary Secretary
15 St Albans Mansion
Kensington Court Place
LONDON W8 5QH

2.1 A Brief Introduction to FORTH

If you have done a lot of programming in BASIC you will find that FORTH is completely different in many aspects and the most striking is the fact that no line numbers are used and consequently there is no GOTO command. This takes a BASIC programmer a while to get used to as he has become accustomed to GOTO´ing all over the place in order to give his programs some structure. FORTH is a structured language and the user has to be a little bit more methodical and learn to think out his program before "playing" with the keyboard.

Another feature of FORTH is the ability to work in any number base and to change the number base from within programs. This can be quite useful when writing "Real World" programs in which most of the work involved is setting and resetting bits on various ports. The programmer can then work in Binary and change back to Decimal or Hex at will, and can, if desired, create other number bases.

FORTH is very fast. It is normally 10 times faster than BASIC, and when the ultimate speed is required, FORTH words can be defined with the Assembler and they will then run at full machine speed, the only overhead being the threading of the words back into the Vocabulary.

Because FORTH is an Interpretive Compiler, source text is only used once and some means is required to store the text for editing and documentation purposes. The method used in FORTH is a virtual memory system where text is used in Blocks of 1k arranged as 16 lines of 64 characters and stored on the disc. This appears to the user as a slow form of RAM and the system interfaces to the DFS in a transparent manner, without any user intervention. The Blocks or Screens can be used to store all manner of information other than just pure ASCII Source Text.

2.2 Reverse Polish Notation

This is something that newcomers to FORTH find very puzzling because they think that Reverse Polish Notation is obsolete and that it is useless. In fact it is exactly the opposite, and when used properly, has a great many advantages.

RPN is the method used to implement a Stack Structure and is identical to the micro-processor stack mechanism. An interesting thought to consider is why all the microprocessor manufacturers implement a stack structure in their micro-processors.

The reason is speed.

A Stack is a fast way of communicating information from one routine to another, and a way of storing information for a short while. You could look at a stack as a FIFO (First-in, First-out) buffer.

Master RPN and the Stack and you have mastered FORTH. A great deal of FORTH is to do with the stack and stack manipulation. A large part of good programming in FORTH is to do with using the stack wisely. To help you with this, we have provided you with an automatic stack display utility which will show you what is on the stack so that you can observe the stack effect of FORTH words.

Now turn to Section 22.1 for details of the Stack Display Utility as this will be required for the rest of this part of the manual.

2.3 Defining New Words

If you have been using BASIC for some time you will know that the language consists of whatever commands you have been provided with originally. There is no easy way of adding your own words. You have to make do with writing your program with the commands provided.

FORTH is the opposite of this. You can create whatever new commands you wish, and you can use existing FORTH words or you can create totally new words of your own. Most programming in FORTH is the former, but an Assembler is provided to achieve the latter, and it is possible to mix assembler statements with high-level FORTH if you wish.

New words are created by using the defining words supplied in FORTH, and there are even defining words supplied for defining new defining words !! Think about that for a while.

So FORTH is completely flexible and the act of programming in FORTH is the act of creating new words. In fact you are actually creating an application language when you write a program and the words so created can be very descriptive of their actual functions. (READ_D-TO-A might be a typical example.)

New words created are put into FORTH´s "Dictionary" which already contains all the pre-defined words supplied on the system. Executing the word DLIST will reveal the Dictionary on the screen.

2.4 Loading Definitions from the Disc Blocks

As was briefly mentioned earlier, Source Text is normally stored on the disc in the form of Screens or Blocks of 1k. In order to store the text on the Disc Screens, the FORTH Editor is used. See: Section 6

The text being placed onto the Disc Screens would normally be the user's program and the user would make sure that his Source Text is safely on the Disc before attempting to compile the text by loading it into the system. If the Source Text is, say, on Screen 5, it is only necessary to execute the command "5 LOAD" for the Source Text to be brought into the system and compiled as if it had all been typed from the Keyboard.

Screens can be linked together by using a "Load Block" or by chaining each screen together. This has the effect of producing a large area of space for applications requiring more than 1 screen.

When screens are LOADed, the Source Text is compiled into a list of addresses in the FORTH Dictionary under the name assigned to the word and the Source Text is no longer required by the system.

The Source Text will invariably be required again by the user for editing or printing out and so is safely stored on the disc for future use. The system, however, does not require that source again, as it has converted it to an internal form which it can execute time and time again if required. From this point on the source text is redundant unless the user requires to make some change and he FORGETs his current definition and reLOADs his edited Source Text.

For those with no disc system, the cassette tape system has to be used instead but this works exactly the same as above, except that the tape has to be positioned correctly in order to find the block files required. See: Section 5.6

2.5 Vocabularies

When you execute a word on your system, be it DLIST or whatever, FORTH searches its Dictionary for the word. If it cannot be matched in the Dictionary FORTH tries to convert it to a number and if it converts to a number in the current base, it is pushed onto the stack. If it cannot be converted to a number an error condition is notified by displaying the message "Undefined".

When the FORTH Dictionary becomes large, searches take more time as the search has to compare every word in the Dictionary. Also as the Dictionary becomes larger the user starts to have difficulty in making up unique names for each word.

One solution to the above problems is to have the ability to create "Sub-Dictionaries" or Vocabularies, which can be selectively searched and hence the Assembler Vocabulary does not have to be searched when an Editor word is required and so on.

This is done by having a defining word called VOCABULARY and some system variables called CURRENT and CONTEXT and the word DEFINITIONS.

3.0 The FORTH-83 Standard

Material relating to the FORTH-83 Standard is the Copyright Property of the FORTH Standards Team and is used with their permission.

"In the interests of transportability of application software written in FORTH, standardisation efforts began in the middle-1970´s by the European FORTH User´s Group (EFUG). This effort resulted in the FORTH-77 Standard. As the language continued to evolve, an interim FORTH-78 Standard was published by the FORTH Standards Team. Following FORTH Standards Team meetings in 1979 the FORTH-79 Standard was published in 1980.

The FORTH Standards Team is comprised of individuals who have a great variety of experience and technical expertise with FORTH. The FORTH Standards Team consists of both users and implementers. Comments, proposals, and correspondence should be mailed to:

FORTH Standards Team
P.O.Box 4545
Mountain View
CA 94040
United States of America.

FORTH´s extensibility allows the language to be expanded and adapted to special needs and different hardware systems. A programmer or vendor may choose to strictly adhere to the standard, but the choice to deviate is acknowledged as beneficial and sometimes necessary. If the standard does not explicitly specify a requirement or restriction, a system or application may utilise any choice without sacrificing compliance to the standard provided that the system or application remains transportable and obeys the other requirements of the standard."

4.1 Installing the EPROM & Initial Power-up

The BBC Micro is able to take up to four paged ROMs. These ROMs fit into the positions denoted by the following I.C's :

IC52	--	Normally the BASIC ROM.
IC88	--	Normally the DFS ROM.
IC100	--	Normally vacant.
IC101	--	Normally vacant.

Multi-FORTH 83 can be fitted into any of these sockets, but would usually go into IC100 or IC101 position, in which case it would have a lower priority than BASIC and the Machine would come up in BASIC at power-on. If the User wished his BBC Micro to become a "FORTH Machine" he has to merely replace the BASIC ROM at IC52 with the Multi-FORTH 83 ROM and place the BASIC ROM elsewhere, thereby giving the FORTH ROM the higher priority and the Machine will come up in Multi-FORTH 83 at Power up and not BASIC.

The correct orientation of the Multi-FORTH 83 ROM will be obvious from examination of the BASIC ROM or the MOS ROM at IC51. We would suggest to the User that if he has some difficulty understanding or getting past this section then he is unlikely to possess the skills required to fit the ROM successfully.

Assuming that the User has inserted the ROM in the correct way (yes, the same way round as all the other chips in the BBC Micro), we are now ready to power up the machine. Assuming that BASIC is the highest priority ROM in the system, it will be necessary to type :

```
*FORTH-83      <return>
```

and the system response should be :

```
Multi-FORTH 83

C/W? _
```

The User should the press C (in the case of a cold-start) or W for a warm-start. So in our case the answer is of course C.

A new line will be executed and the User should now type VLIST <return> to obtain the customary dictionary listing on the screen just to prove the system is working properly. Should there

be any difficulty with these instructions it is possible that the User has not inserted the ROM in the correct place or perhaps he has bent one of the leads of the IC over.

The Multi-FORTH 83 ROM or EPROM is tested by us at least four times at various stages of manufacture. The only faulty ROMs are are those that the customer has fitted incorrectly and/or blown up and/or damaged in some way.

We speak from long and hard experience !!

4.2 Warm and Cold Restarts

A warm start can be performed by hitting the "ESCAPE" key or by executing the word QUIT or ABORT when a word start is required from within a program. The word ABORT" could also be used. A cold start can be performed by hitting the "Break" key.

When the machine is asking for a cold or warm start (C/W?) it is possible to enter OS Commands using "*" so that devices or parameters can be set up before entering Multi-FORTH 83.

A cold start from within a program can be accomplished by executing a jump to 8000 Hex, which is the start of the EPROM. The easiest way of doing this is by defining a code word :

```
HEX    CODE COLD  8000 JMP,  END-CODE
```

4.3 System Response and Errors

Multi-FORTH 83 prompts the User with an "OK" after each successful operation in execution mode (i.e. from the keyboard). The operation may be as simple as putting a number onto the stack, or as complicated as an entire block of source text. As long as no error is found, the system reports with an "OK".

If an error should occur during an operation, an error statement will be displayed on the screen followed by an error code. The error codes are listed on the inside back over of this manual and in Appendix 21.1a.

The FORTH word REPORT calls MESSAGE to print out the error message in the format :

Msg..n

If MESSAGE were to be vectored to another user created word, the format of the error messages could be changed to whatever the user requires.

The system perfoms a warm start after an error message is printed.

5.0 Multi-FORTH 83 on the BBC

The BBC Micro is quite a powerful machine by virtue of the way in which the Operating System (M.O.S) has been written and the hardware facilities provided.

Multi-FORTH 83 has had to take into account all the features of the BBC and this section in the manual deals with most of them. The User is recommended is read this section in conjunction with the BBC User Guide, as most of the commands are indentical the main difference merely being the order of the parameters due to FORTH using Reverse Polish Notation (RPN).

It has not been possible to implement all the features of the PLOT command due to space limitations. The user probably would not be aware that some PLOT functions in BASIC are done purely by software using BASIC´s floating point arithmetic and the others are done by driving the hardware in the screen handler. We have implemented the hardware features in FORTH for a limited PLOT command. (Although limited, it still works very well !!).

5.1 OS Commands

OS commands always start with a "*" and Multi-FORTH 83 recognises this character and always acts in a special way when this character is encountered. See: EXPECT

The effect of this is that the user can use all the * commands that he has become familiar with in BASIC, including all the DFS commands such as *CAT and *FORM, *INFO, etc.

Because the Operating System treats the character "*" in a special way, the whole of the line containing "*" is passed to the OS for interpretation. This can cause problems if other FORTH words appear on the same line, which they would if * commands are being used in colon definitions or from disc blocks.

In order to overcome this problem the "End-of-Line" character contained in the user variable EOL is used. This has a default value of ASCII 5E hex which is the character "^". This is used as EOL until such time as the EOL character is changed. See: Section 6

An example :

```
: TEST  *CAT  ;       will not work, whereas

: TEST  *CAT ^ ;      works fine.
```

In a similar way a number of commands which would normally need to be followed by a "return" and would consequently be on separate lines can be put on the same line by using "^".

Using the editor :

```
4 P xxxx ^ 5 P yyyy ^ 6 P zzzz      <return>
```

will input 3 lines.

Another example :

```
*CAT  .( Hello )     will not work, but

*CAT ^  .( Hello )   will work ok.
```

It must be obvious to the user that it would be totally wastful of EPROM space to put FORTH words to carry out OS functions into Multi-FORTH 83 as the user can use all the * commands available on the machine.

An understanding of the use of the EOL character will be required before the user can incorporate "*" commands into colon or other definitions.

5.2 VDU Commands

There is no word called "VDU" in Multi-FORTH 83 as its function is covered by the word EMIT. If you look at the BBC User Guide Page 373, you will see that VDU is functionally equivalent to the BASIC CHR$ command, and the direct equivalent of that in FORTH is EMIT.

Because Multi-FORTH 83 is multi-tasking, it would not be desirable if a task was swapped out half-way through sending control characters to the VDU. Therefore when using EMIT to output a string of VDU characters, always use KEEP and GIVE to stop any interruption of the characters. See: Section 19

An example :

The equivalent of VDU 14 to turn "auto-paging mode" on would be:

```
14 EMIT
```

Try VLIST to prove that this is so.

Its easy, isn´t it ?!!

5.3 SOUND and ENVELOPE

The user should refer to the BBC User Guide, Page 180, as these two words are functionally identical to the BASIC commands of the same name. The only difference is the order of the parameters.

An example : (from Page 184)

```
2 1 2 -2 2 10 20 10 1 0 0 -1 100 100 ENVELOPE

1 2 100 100 SOUND
```

5.4 COLOUR

The user should refer to the BBC User Guide, Page 222, as this word is functionally identical to the BASIC command of the same name. The only difference is the order of the parameters.

An example : (from page 222)

```
5 MODE          <return>
1 COLOUR        <return>
2 COLOUR        <return>
3 COLOUR        <return>
```

5.5 PLOT Command

Try this :

```
: BOX  3 PICK OVER 4 PLOT  2DUP 5 PLOT  SWAP 2 PICK 5 PLOT
       2 PICK ROT  5 PLOT       5 PLOT  ;

0 MODE    <return>

1064 16 1272 944 BOX     <return>

or   536 624 1064 1016 BOX    <return>
```

5.6 Time

Multi-FORTH 83 uses the BBC´s interrupt clock for the time function. The BBC´s interrupts are also used to drive the tasker mechanism and in order to do that they have to be redirected into Multi-FORTH 83.

This can cause a problem if the user wants to go into another language, such as BASIC, because the BBC´s interrupts will have been redirected and will need restoring back to their original destination. Two FORTH words have been provided to do this, one called ON, the other OFF.

When the system starts up and presents the C/W? prompt, the timer interrupt has not been revectored into the FORTH Rom and so this would be the ideal time to start up another language. If it is not done at this time the word OFF would be needed before calling a new language from inside Multi-FORTH, otherwise the lack of clock interrupts may cause the language to ´crash´.

For example :

to enter BASIC, type OFF *BASIC <return>

or alternatively press "Break" and type *BASIC at the C/W? prompt.

If you want to include your own code in the timer interrupt then you should use OFF first, then revector location 204 hex and then use ON.

ON should never be used when the system is already ON or the system will lock-up.

The BBC system clock can be accessed by using the following sequence of words, returning a double number on the stack representing the number of ´ticks´ since power-up.

```
0PAGE 1 OSWORD DROP DROP      (this reads timer to 0PAGE)
0PAGE 2@  SWAP                (this fetches it from 0PAGE)
```

D. would display it to the screen.

5.7 Filing System Commands

a/ Disc

Multi-FORTH 83 would normally be used with a disc system for program development, as a cassette system is very slow and restricting. The Tape and ROM system is supported, however, and is described in the next section.

Multi-FORTH 83 uses the discs in the same way as other files on a disc. Space is reserved in the disc's directory for files which Multi-FORTH treats as virtual memory block files, although to the DFS they appear the same as any other type of file.

Very few commands are supplied to use the disc as the user still has access to the DFS in the same as any other language, such as BASIC, by making use of all the normal commands which are prefixed by "*".

The user would then format his disc in the same way as he would from BASIC, and assuming that a formatted disc is being used, it is only necessary to create an initial block file of 0k and then invoke the "USE" command and then the editor.

This would be the sequence :

```
OPENOUT  TESTBLK   CLOSE      <return>
```

which creates an entry in the directory, opens the file called TESTBLK, and assigns that file to file handle F0 (the default). The file is now ready to USE. However, if listing is attempted at this stage strange and nasty things could happen as the file will extend and the random contents of the extending file will be output through the operating system to the screen.

The results of this will be unpredictable, so it is wiser to wipe the contents of the disc first using the editor command WIPE.

```
USE  TESTBLK          <return>

0    Blocks Available.

EDITOR     <return>

SCR ?                 (Check that block 0 is the current
                      screen. Should return 0, if not
                      type 0 SCR !).

WIPE       (Now start clearing some screens, because
           they will contain junk otherwise).

N WIPE     (Now the next block)
```

(continued)

```
.....     (And so on as many times as you want initial
          clean blocks).

FLUSH     (Always finish a session with this word).
```

We can now start listing the blocks to the screen safely as they now contain blanks as a result of being WIPEd. The user should refer to Section 6 The Visual Editor, for details of how to put Source Code onto the disc blocks for later compilation.

More than one block file can be open at any one time by using F1 F2 and F3 to associate files with file-handles in this manner :

```
F1 USE file1

Similarly    F2 USE file2        and so on with F3
```

It is assumed that 'file1' and 'file2' have already been created as described on the previous page. The user can then switch between files simply by executing F1 or F2 in a similar way to DR0 and DR1 in good old FIG-FORTH. F0, of course, is the default and is automatically assigned when the first file is opened with USE or the other words.

The disc buffer 'remembers' which file a block belongs to so that :

```
F1  1 BUFFER  F2  1 LIST      <return>
```

will list block 1 from file 2. However, a block does not remember which filing system it came from so when using the TAPE or ROM filing system 'USE' should not be used and therefore no file handle would be associated with the file.

Because of this, two checks have to be made when reading or writing a block.

1/ If the filing system is the TAPE or ROM, and the file handle does not equal zero, error "FS?" occurs.

2/ If the filing system is the DISC, and the file handle equals zero, error "No Block File" occurs.

Other Disc words:

```
OPENIN    OPENUP    OPEN      CLOSE     FS    EXT
```

b/ Tape

The tape filing system is not as easy to use as the disc system because of the limitations of the media being used, but with practice it becomes very straightforward. A typical session might be like this :

```
*TAPE                (Invoke the tape system)
1 BUFFER             (Sets aside a buffer to block 1)
1 SCR !              (Sets SCR to block 1)
EDITOR  WIPE         (Clears the buffer to spaces)
1 EDIT               (Invokes the editor)
```

The user can now place source text into the block and the process above is repeated for all the other blocks required.

Once the source text is written to the block, it can be saved by using FLUSH.

```
FLUSH
Position Tape at Block 1
RECORD then RETURN
                              (Start up recorder)
Blk1        01 nnnn   etc.
```

This process is repeated for every block of text you wish to save. Reading it back in again is just as easy :

```
n LIST          (For whatever block you wish to load)
Position Tape at Block n
Searching
Loading         (When specified block is found)
Blkn        01    etc.
```

It is possible to use tape and disc files together by using F0 F1 F2 and F3 :

(continued)

(continued)

```
Define disc file as F1

     F1  USE file1

To use disc file

     *DISC ^ F1      1 LIST

To use tape file

     *TAPE ^ F2      1 LIST
```

5.8 The User Defined Function Keys

The user defined keys are set up exactly as you would from BASIC the extra bonus being that you can fill them with Multi-FORTH 83 Commands !!

An example :

```
*KEY 0 VLIST |M     <return>
```

This will program key 0 to execute a VLIST command. All the other keys can be programmed in a similar way with whatever commands you desire.

Keys can be programmed from the disc by using the EOL character in the manner described in Section 5.1

An example :

```
SCR# 1
  0
  1  *KEY 0 VLIST |M ^
  2  *KEY 1 DLIST |M ^
  3  *KEY 2 7 MODE |M ^
  4
  5       etc.
```

See: BBC User Guide, Page 141 "Programming the User Defined Keys"

The user could also program keys from within another program if required.

6.1 The Visual Editor

The purpose of the Editor is to permit the user to create source text in the virtual memory system so that this source text can be stored on the disc or tape and brought in as required. The user can always type programs directly at the keyboard, but as FORTH is a compiler, this source text is lost forever. Another advantage of storing source text in virtual memory form is that it can be recalled and changed if required and it can be dumped onto a printer quite easily.

Multi-FORTH 83 uses blocks, each of which contains 1024 characters (= 1k bytes). Each block has a logical block number assigned to it for easy reference; numbers run from 0 through to the capacity of your disc system. For display purposes, FORTH divides the 1024 characters into 16 lines of 64 characters with the lines numbered from 0 to 15.

The easiest way of using the Editor is to use the word EDIT. This of course assumes that the user has opened the file by typing USE <filename>. The user then executes n EDIT <return> where n is the desired block number. EDIT invokes the Editor, puts the machine into EDITOR Vocabulary and switches to Mode 3, fetches the desired block from the disc or tape and then performs a LIST. The current screen is then stored in SCR ready for manipulation later by B and N if desired.

Once selected, the current block may be displayed again by the following command : L

To display the next block, use N to change SCR and then L to display it :

N L

In a similar manner, use the combination of B (for Back) and L to display the block that precedes the current one.

Lines of text can be stored in PAD to enable the user to move lines about. PAD will only store one line of text at any one time so be careful ! Study the section on the Editor commands.

On the BBC the editing of lines is done in an identical way to the editing of BASIC program lines by using the four cursor keys and the ´Copy´ and ´Delete´ keys. The only proviso is that the line on the lower part of the screen must start with a valid P command. I.e. it should start with, say 10 P and then the cursor key can be used to move the editing cursor up the screen and the ´copy´ key can be used as desired. There must be a space on either side of the ´P´. In this example, when the return key

is pressed the line of text will be placed into line 10 of the current block. This is how blocks would be edited and lines of text manipulated.

Practice will make perfect !!

Always be sure to type FLUSH at the end of an editing session to ensure that your work is saved to the disc or tape system.

When the BBC´s editing keys are used, it has the unfortunate effect of disabling the system interrupts and stopping the multi-tasker from working. This is an unfortunate limitation of the BBC´s Operating System and not a limitation of Multi-FORTH 83.

The user should therefore avoid using the BBC´s cursor and copy keys while tasks are running as they will stop otherwise and you may not like that !!

6.2 Editor Commands

B addr ---

This word is normally used in the form : B L <return>
It decrements the contents of SCR by one thereby allowing the user to move backwards through a block file.

COPY n1 n2 ---

This word copies block n1 to block n2.

D n ---

Delete line n but hold it in PAD. Line 15 becomes blank as lines n+1 to 15 move up one line.

E n ---

Erase line n with blanks.

H n ---

Hold line n at PAD (used more by the system than by the user).

I n ---

Insert the text from PAD at line n, moving the old line n and following lines down. Line 15 is lost.

L ---

List the current screen.

LMOVE addr n ---

This is an internal word to move lines about.

N addr ---

This is normally used in the form : N L <return>
It increments the contents of SCR by one, thereby allowing the user to move forwards in the block file.

P n ---

The next 64 characters from the input stream will be moved to line n of the current block. See: Previous section.

RENUM n ---

This word RENUMbers the last disc block accessed. I.e. to copy block 1 of file 0 to block 5 of file 1 :

```
F0 1 BLOCK DROP   F1 5 RENUM    FLUSH
```

S n ---

Spread at line n. n and subsequent lines move down one line. Line n becomes blank. Line 15 is lost.

T n ---

Line n is output to the screen.

WIPE ---

The current block is filled with blanks. Use with caution !

The "End-of-Line" Character

This character at first appears to present a problem to the editor in that the Operating System regards it as a termination of input and therefore it never reaches the editor.

Therefore if the user wishes to place an EOL character "^", on a disc block, which he will want to do if he is programming the user keys from a disc block, it is first necessary to temporarily substitute another character for the contents of the user variable EOL.

This variable has a default value of 5E hex (ASCII ^) and can be replaced by any character which is not being used in that session.

For example :

```
ASCII #  EOL !
```

Now you can insert ^ using the editor. When you have put in all the ^'s that you wish to use replace the default value :

```
ASCII ^  EOL !
```

7.1 Stack Structure

FORTH is different from most other computer languages in that it uses a stack. A stack is a data structure which stores things in the order in which they were entered. Items can be removed with the last item first. Here is an example:

Enter three numbers :

```
0 1 2  <return>    The stack looks like this:

2     top
1
0     bottom
```

To display the top item on the stack, simply type :

```
.     <return>
```

This will remove 2 from the stack and display it. Typing :

```
.     <return>      a second time will display the 1.
```

All the mathematical operations are also performed on the stack. To examine this, type the following:

```
0 2 3      <return>      The stack looks like :

3
2
0
```

Now type :

```
*     <return>      This command takes the top two items off
```

of the stack, multiplies them, and puts the result back on the stack. The result is :

```
6
0
```

If another multiplication were performed, then :

```
*     <return>      Would leave a :

0                        ( 6 * 0 = 0 )
```

Most of the commands in Multi-FORTH use a stack. Multi-FORTH has a separate 16-bit Return stack and a 16-bit number stack. You will be using the number (or parameter) stack most of the time. The Return stack is by FORTH to store internal parameters relating the the compiler and loops, and the user will not be involved with the Return Stack very often.

7.2 Dictionary and its Use

After reading the last section you should have a feel for how the parameter stack works. Multi-FORTH words are stored in another place in memory, the dictionary. The dictionary grows upwards from 2518H. Every FORTH word is stored in memory with a header. The header contains the number of characters in the word plus the characters of the word itself. The number of characters plus the characters themselves are used by the outer-interpreter when a search of the dictionary is made for a word.

The programmer can create new words in the dictionary using various compiling words. These words are described in detail in Section 16

Here is an example of how a new word would be defined using the COLON and SEMI-COLON compiling words. (Known as a Colon Definition). To create a word which takes the average of two numbers on the stack and displays the result, type :

```
: AVG  + 2 / . ;    <return>
```

The above program computes averages by adding the two values on the stack and then pushing 2 on to the stack and performing a divide. The dot (.) then takes the value off the stack and displays it.

The word AVG can now be used to take the average of two numbers. Find the average of 86 and 46 by typing :

```
86 46 AVG      <return>
66 OK
```

The answer of 66 is displayed to the screen.

VARIABLEs and CONSTANTS can also be created in the dictionary. This is covered in Section 12

7.3 Command Format

Many of the descriptions of Multi-FORTH words will be of the following form :

```
top --> stack                          stack <-- top
        before        COMMAND          after
      execution                      execution
```

Each word is described by an example. The state of the stack is shown before and after the word is executed. The words are first described in a generic format and then an example of each one is given.

What the symbols mean :

```
    n = 16 bit number (n1, n2, n3 etc.)
    d = 32 bit number (d1, d2, d3 etc.) Sometimes nlow and nhigh
              are used to describe how double numbers appear on
               the stack or in the dictionary.
    u = unsigned 16 bit number.
 addr = represents an address in memory.
    b = byte.
    c = character.
    f = boolean flag (0= false, 1= true).
```

When the results are shown using the Stack Display Utility, the following format is used :

```
        Stack Base > nn nn nn
```

The right-most nn is the top of the stack, the number of nn items represent the number of items on the stack. See: Section 22.1

8.0 Mathematical Commands

FORTH uses integer arithmetic. For some this may be inconvenient at first. However, one of the commodities a computer has is speed. Often it is desired that operations be performed as quickly as possible, perhaps because the calculation is done many times per second, and integer arithmetic is much faster than floating point arithmetic. If you need more accuracy in your values, the values can be scaled by a factor of 100 or 1000. Scaling by 100 would allow you to include pennies in calculations based in the pound and pence system.

Most 16 bit arithmetic is signed arithmetic in Multi-FORTH. However, most 32 bit, and all 64 bit arithmetic is unsigned. This may seem to present a problem if you are not keeping track of the approximate magnitude of your calculations.

Here is the difference between signed and unsigned arithmetic. Below is listed a chart showing the difference, with the Binary and Hex formats of the numbers shown. This can be extended to 32 bit and 64 bit numbers.

UNSIGNED	BINARY	HEX	SIGNED
65535	1111111111111111	FFFF	-1
65534	1111111111111110	FFFE	-2
..	....	..	..
..	....	..	..
32768	1000000000000000	8000	-32768
32767	0111111111111111	7FFF	32767
..	....	..	..
..	....	..	..
0	0000000000000000	0000	0

Table 8.1

Unsigned, Signed, Binary, and Hex Numbers

\+ (Addition) Adds the top two stack items n1 and n2 leaving the sum on the stack.

n1 + n
n2

```
1 1 + . <return>
2 OK
```

- (Subtraction) Subtracts the top item, n1 from the second item, n2 and leaves the result on the stack.

```
n1 - n                          2 1 - . <return>
n2                              1 OK
```

* (Multiplication) Multiplies the top two items, n1 and n2, and leaves the product on top of the stack.

```
n1 * n                          2 4 * . <return>
n2                              8 OK
```

/ (Division) Divides the second item, n2 by the top item, n1, and leaves the result on top.

```
n1 / n                          6 2 / . <return>
n2                              3 OK
```

2* (Multiply by 2) This multiplies the top stack item by two.

```
n1 2* n                         3 2* . <return>
                                6 OK
```

2/ (Divide by 2) This divides the top item by two.

```
n1 2/ n                         4 2/ . <return>
                                2 OK
```

ABS (Absolute Value) Leaves the absolute value of the top item on top of the stack.

```
n1 ABS n                        -12 ABS . <return>
                                12 OK
```

MAX (Maximum) Finds the larger of the two top stack items and leaves it on top of the stack.

```
n1 MAX n                        9 4 MAX . <return>
n2                              9 OK
```

MIN (Minimum) Finds the smaller of the top two stack items and leaves it on top of the stack.

```
n1 MIN n                        9 4 MIN . <return>
n2                              4 OK
```

MINUS (Unary minus) Changes the sign of the top stack item.

```
nl MINUS n                    31 MINUS . <return>
                              -31 OK
```

+- (Swap Sign) Applies the sign of the top item, nl, to the second item, n2 and leaves the second item at the top of the stack.

```
nl +- n2 (signed)             2 -3 +- . <return>
n2                            -2 OK
```

MOD Performs the division, n2/nl, and leaves the 16 bit remainder on the stack.

```
nl MOD nr                     15 4 MOD . <return>
n2                            3 OK
```

/MOD Performs the division, n2/nl, and leaves the remainder nl on top, and the quotient, n2, as the second item.

```
nl /MOD nl                    5 2 /MOD . <return>
n2      n2                    1 OK     . <return>
                              2 OK
```

*/MOD Multiplies the second, n2, and third, n3, items and divides by the first, nl, leaving the remainder on the top of the stack with the quotient below it. (Signed arithmetic).

```
nl */MOD nr                   3 3 2 */MOD . <return>
n2       nq                   1 OK        . <return>
n3                            4 OK
```

M* This multiplies two 16 bit numbers, nl and n2, and leaves a 32 bit result, d.

```
nl M* d                       20000 20000 M* D. <return>
n2                            400000000 OK
```

M/ This divides a 32 bit number, d, by a 16 bit number, n, leaving a 16 bit remainder, nr, on top of the stack and a 16 bit quotient, nq, as the second item.

```
n M/ nr                       400000001. 20000 M/ . <return>
d    nq                         1 OK    . <return>
                                20000 OK
```

*/ Multiplies the second item, n2, and the third item, n3, and then divides by the first, n1, leaving the result on the stack.

```
n1 */ n                    4 6 3 */ . <return>
n2                         8 OK
n3
```

D+ (32 bit add) This is a double precision add which adds the top 32 bit numbers found on the stack.

```
d1 D+ d1                   400000. 40000. D+ D. <return>
d2                         440000 OK
```

D- (32 bit subtract) Performs a double precision subtraction of the top 32 bit item from the second 32 bit item.

```
d1 D- d1
d2
```

DABS (32 bit ABS) This operation takes the absolute value of the 32 bit number on the stack.

```
d1 DABS d2
```

9.1 Logical Operators

AND

Performs a bitwise AND of the two 16 bit items on the stack.

```
13   00001101       (8 bit example)
 7   00000111
AND
 5   00000101       result
```

OR

Performs a bitwise OR of the two 16 bit items on the stack.

```
5    00000101       (8 bit example)
9    00001001
OR
13   00001101       result  (13= 0D hex)
```

XOR

Performs a bitwise exclusive-or, XOR, of the two 16 bit items on the stack.

```
5    00000101       (8 bit example)
7    00000111
XOR
2    00000010
```

NOT

Performs a bitwise not, NOT, of the two 16-bit items on the stack.

```
13   00001101       (8 bit example)
NOT
242  11110010       result
```

9.2 Comparison Operators

`<` If the second stack item is less than the first, the operation leaves a -1, (true) otherwise it will leave a 0.

```
ul < f                  0 2 < . (NEW LINE)
u2                      -1 OK      true
```

`>` If the second stack item is greater than the first, the operation leaves a 1, otherwise it will leave a 0.

```
ul > f                  0 2 > . (NEW LINE)
u2                      0 OK       false
```

`0=` Tests whether the top item is 0. If it is, then the operation leaves a 1, otherwise it will leave a 0.

```
ul 0= f                 13 0= . (NEW LINE)
                        0 OK       false
```

`0>` Tests whether the stack item is positive. If it is, then the operation leaves a 1, otherwise it leaves a 0.

```
ul 0> f
```

`0<` Tests whether the stack item is negative. If it is, then the operation leaves a 1, otherwise it leaves a 0.

```
ul 0< f
```

`=` Tests whether the top two stack items are equal. If they are, the operation leaves a 1, otherwise it leaves a 0.

```
ul = f                  37 DUP = . (NEW LINE)
u2                      -1 OK           true
```

`D<` Checks if the second 32 bit item is larger than the first 32 bit item. If the operation is true, then a 1 is left, otherwise a 0 is left. This is an unsigned comparison.

```
dl D< f
d2
```

D> Checks if the second 32 bit item is smaller than the first 32 bit item. If the operation is true, then a 1 is left, otherwise a 0 is left. This is an unsigned comparison.

```
d1 D> f
d2
```

D= Checks if the two 32 bit items on the stack are equal. If they are, a 1 is left, otherwise a 0 is left.

```
d1 D= f
d2
```

DMAX Leaves the larger of the two 32 bit items on the stack. This is an unsigned operator, so -2. will be greater than 2.

```
d1 DMAX d
d2
```

DMIN Leaves the smaller of the two 32 bit items on the stack. This is an unsigned operator.

```
d1 DMIN d
d2
```

D0= Checks whether the 32 bit item on the stack is equal to 0. If it is, a 1 is left, otherwise a 0 is left.

```
d D0= f
```

U< This is an unsigned less than comparison of the two 16 bit items on the stack. If u2 is less than u1, then a 1 is left, otherwise a 0 is left.

```
u1 U< f
u2
```

10.1 Treatment of numbers

FORTH treats numbers as 16 bit values unless told otherwise by the insertion of a "." somewhere within the number, in which case it is treated as a 32 bit value. This is after FORTH has searched the dictionary, because FORTH always searches the dictionary first when anything is input.

If the item is not found in the dictionary, it is passed to the "Convert" routine to be changed into the binary form representing the number in the current number base. If a conversion cannot be carried out an error is flagged. When the number is converted it is "pushed" onto the stack. The convert routine also checks for the "." character, and if necessary treats the number as a double number (32 bit) and "pushes" two 16 bit values onto the stack. The user variable DPL also plays a part in this operation and its role is best explained by some examples :

```
56   <return>          places 16 bits onto the stack.
Stack Base > 56

DPL  ?                 what does DPL hold ?
-1

56.  <return>          let´s try a double number.
Stack Base > 56 0

DPL  ?
0                      This is interesting !
```

Let´s try something else ....

```
56.00     <return>
Stack Base > 56 0

DPL  ?
2
```

So the user variable DPL is telling us the number of places to the left of the decimal point. This is very useful for creating fixed point arithmetic routines.

10.2 Number Bases

One of the most useful features of FORTH is the easy way in which numbers of various bases are handled. FORTH is capable of working in any number base. This is not so difficult to achieve, however, as the microprocessor can only work in Binary. This means that a conversion process must be done to work in Decimal or Hex. Once you have that conversion process, it is not difficult to extend it to cover all number bases. FORTH refers to a variable called BASE during numerical conversion, and changing base is as simple as changing the contents of the variable BASE.

The default base on initial power-up is Decimal (Base 10, 0 to 9)

DECIMAL	Sets the current base to decimal.
HEX	Sets the current base to hexadecimal (Base 16)
BASE	A variable used to contain the current base of the system.

```
nl BASE ! <Return>          Makes the current base nl.

BASE @ . <Return>           Places current base onto the screen

3 BASE ! <Return>           Changes the base to 3

DECIMAL 532 3 BASE ! . <Return>
201201 OK                   (201201 is 532 in base 3)
```

After DECIMAL and HEX, the most useful number base is BINARY and a definition of BINARY would be :

```
: BINARY  2 BASE ! ;
```

If desired, the current number base can be changed from within executing programs and even while compiling.

10.3 Stack Manipulation

As mentioned before, the use of the stack is fundamental to FORTH. We will assume that the user has the stack display utility working. See Section 22.1 for details of the Stack Display Utility.

A good deal of FORTH is concerned with direct manipulation of data on the stack. The stack normally holds 16-bit items and other magnitudes are represented by groups of 16-bit items.

DROP Drops the top stack item.

```
34 56 67 <return>  Ok
Stack Base > 34 56 67

DROP  <return>  Ok
Stack Base > 34 56
```

SWAP Exchange the top two stack items

```
Stack Base > 34 56

SWAP   <return>  Ok
Stack Base > 56 34
```

DUP Copy top stack item

```
Stack Base > 56 34

DUP    <return>  Ok
Stack Base > 56 34 34
```

ROT Rotate the top three stack items, bringing the deepest to the top.

```
Stack Base > 56 34 34

ROT    <return>  Ok
Stack Base > 34 34 56
```

-ROT Rotate the top three stack items, putting the top to the deepest.

```
Stack Base > 34 34 56

-ROT   <return>  Ok
Stack Base > 56 34 34
```

OVER Copy the second stack item and place on top of the stack.

```
Stack Base > 56 34

OVER   <return>  Ok
Stack Base > 56 34 56
```

PICK Duplicate the n´th stack item on the stack. N must be greater than zero.

0 PICK is equivalent to DUP
1 PICK is equivalent to OVER

```
Stack Base > 56 34 56

2 PICK  <return>  Ok
Stack Base > 56 34 56 56
```

ROLL The n´th stack item is removed and put on the top of the stack, the other items moving accordingly.

2 ROLL is equivalent to ROT
0 ROLL is a null operation

```
Stack Base > 12 34 56 78

3 ROLL  <return>  Ok
Stack Base > 34 56 78 12
```

-ROLL The top stack item is removed and placed n places down, the other items moving accordingly.

```
Stack Base > 34 56 78 12

3 -ROLL  <return>  Ok
Stack Base > 12 34 56 78
```

?DUP Copy the top stack item only if it is non-zero, otherwise ignore.

11.1 Memory Commands

One of the most useful features of FORTH is the ease of access to the host system's memory.

! Stores the second stack item at the location addressed by the top stack item.

```
Stack Base > 5555 3000          ( We are in Hex ! )

!   <return>  Ok
```

@ Fetches the value at the memory location addressed by the top item on the stack, and places it on the stack.

```
Stack Base > 3000

@   <return>  Ok
Stack Base > 5555  ( 5555 Hex is the contents of 3000H)
```

C@ & C! Are the 8 bit versions of the above words. The least significant 8 bits are used, the others being ignored. These words are used to fetch and store one byte at a time, rather than two bytes as above.

2@ & 2! Are the double number or 32-bit versions of @ and ! , these words dealing with 4 bytes at a time.

+! This is an indirect memory increment operator.

```
55 3000 !       <return>

3000 @          <return>
Stack Base > 55      ( So 3000 Hex contains 55 H)

3 3000 +!       <return>

3000 @          <return>
Stack Base > 58
```

11.2 Memory Manipulation

CMOVE

CMOVE>

This is one of a number of words used to move bytes around memory.

The easiest way to demonstrate one of these commands is to use the Video screen of the computer. If the user hits the "Break" key and does a cold start by typing "C" and then types HEX <return> the screen uses memory from 7C00 Hex to 7FFF Hex. The following sequence copies (or moves) what is on the top part of the screen to another part.

```
7C00 7C80 100 CMOVE>      <return>
```

The user should study the glossary definitions of the words mentioned in order to understand the exact behavior of each word.

Words that manipulate memory :-

CMOVE, CMOVE>, MOVE, FILL, ERASE, BLANK

12.1 Data Types

The user will be aware by now that Forth normally deals with 16-bit quantities but that any other size can be easily dealt with. Multi-FORTH 83 has a number of double (or 32-bit) number operators and the user has the ability to define other sizes if he so desires. The primary data types supplied on a standard Forth system are the Variable and Constant. A number of variables and constants are supplied pre-defined for system purposes and the user can define his own using the words VARIABLE, 2VARIABLE, CONSTANT and 2CONSTANT.

The user is also free to define arrays and matrices and other data types, for more details of these see Section 16 and Section 10.4

12.2 Variables

VARIABLE is a defining word used to create a variable which references a memory location used to store two bytes of information, usually an address or value. When ever the variable so created is executed by name, the parameter address of that variable is placed onto the stack. As the variable has random contents after being defined, the first thing to do after defining a variable is to initialise its contents.

Variables are used to keep track of values which may change from time to time, but where it would not be convenient to store them on the stack.

When you invoke a variable by its name, its address is placed on the stack. Invoking a constant, on the other hand, places its value on the stack.

Examples :

```
VARIABLE  TEST
```

which means "define a variable named TEST"

This variable has an unknown value, so it must be initialised. It can be set to zero like this :

```
0 TEST !
```

In RAM systems, the difference between CONSTANT and VARIABLE is mainly one of optimisation. CONSTANT is optimised for the case in which you want the value often and change it rarely. A VARIABLE is most appropriate when values will be changing often.

When the ultimate destination for a program is EPROM, the difference between VARIABLE and CONSTANT is more profound. a CONSTANT will be compiled into EPROM and therefore its value cannot be changed. A VARIABLE, however, compiles into EPROM a reference to a RAM location.

12.3 Constants

If a value is used frequently or if a value is associated with a specific function, you might want to name it. Often the name is easier to recall and enter correctly than is the number itself. A named value is a constant and CONSTANT is the Forth word used to assign dictionary names to constants. If you were converting ounces to pounds you could define a CONSTANT named POUNDS.

```
16 CONSTANT POUNDS          <return>
```

which creates a new word called POUNDS and gives it the value of 16.

After POUNDS has been defined, it can be used just as the value of 16 would be. Executing POUNDS will place the value 16 onto the stack.

```
POUNDS 3 *    <return>
```

works out the number of ounces in 3 pounds. Once a value is defined as a constant, its binary value is independent of the current number base.

In RAM systems, constants can be changed by using the Forth word ´ (called "tick"). Tick looks up a word in the dictionary and leaves the address of its code field on the stack. All that has to be done now is to convert that code field address into a parameter field address using >BODY because that is where the value assigned to the CONSTANT is stored and another value can be placed in top of the original value, so:

```
10 ´ POUNDS >BODY !       <return>
```

This will store the new value of 10 into the parameter field address of POUNDS, effectively changing its value.

```
POUNDS .      10  OK
```

One of the surprising aspects of programming in Forth is how few CONSTANTS or VARIABLEs are needed. Since the parameter stack is used to hold values which need not be named or need not take up dictionary space, the need to define every literal or temporary value as a CONSTANT or VARIABLE is eliminated. Only fundamental parameters in an application will need dictionary space.

The most common failing of inexperienced Forth programmers is the excessive use of constants and variables. To realise the most value from your Multi-FORTH 83 system, try to be alert to this tendency and resist it.

12.4 Arrays

Arrays can be created in Forth just as they can in BASIC, FORTRAN, or other languages. First, space must be allocated in the dictionary.

```
VARIABLE  VAL 22 ALLOT        <return>
```

This will create a variable called VAL. It then reserves 22 additional bytes for it in the dictionary. This gives VAL the capacity to hold 12 16-bit numbers (24 bytes).

Because the execution of a named variable causes the address of the first two bytes of that referenced variable to be put onto the stack, this address can be used to index into the other bytes of the array.

The array must of course be initialised before use and this can be done by putting the value and the array item number on the stack and using the following sequence of words :

```
2*  VAL + !     <return>
```

so to put 3000 hex into the first element the sequence would be :

```
HEX    3000 1  2* VAL + !      <return>
```

in order to access any 2 byte value in the array put the array item you wish to access on to the stack and use the following commands :

```
2*  VAL  +  @        <return>
```

This will access the proper array value by doubling the index, adding it to the variable address, and fetching the proper number from that address.

This is not the only way to construct arrays. A more efficient and elegant way is to use the CREATE ..... DOES> construct. This method will be shown in Section 16.2

As data structures, arrays are of fundamental importance in implementing solutions to programming problems.

13.0 Control Structures

The newcomer to Forth always seems to find it difficult to understand the lack of a "GOTO" command. Once it is realised that if you have structures you don't need GOTO, it is much easier to grasp what Forth is all about.

Structured programming is a frequently used term whose strictest usage says that all programming should use one of three means to control logical flow :

1/ Sequential operation where one step follows another.

2/ Conditional operation where a path through a program depends upon the results of logical tests.

3/ Repetitive operation where a program repeats until a certain logical test is true or false and then continues.

Forth has all these structures and allows you to define any other structures that you may desire to create.

A conditional structure allows a program to 'make a decision'. This is normally done by testing the stack and these structures must only be used from within a colon definition. The normal form of such a structure is :

```
: <name>  IF .... ELSE .... THEN .... ;
```

IF checks the stack. If the value is non-zero (true) the code between IF and ELSE is executed. If the value is zero, (false) the code between the ELSE and the THEN is executed. For example :

```
: TEST IF ." True " ELSE ." False " THEN ." End " ;
```

will give the following response :

```
1  TEST   <return>        True END  OK

0  TEST   <return>        False END  OK
```

The IF contains an inherent "equal to zero" test which destroys the value being tested. If this value is needed within the IF ... THEN structure it must be DUPed before the IF. The ELSE clause is optional.

Repetitive structures are generally two types of loops; one being a finite loop in that it repeats a given number of times and the other being indefinite in that it repeats until a given condition is met.

A finite loop is created by the following structure :

: <name> limit initial DO ...code... LOOP ;

where limit is the upper limit of the loop count and initial is the lower limit of the loop count. The index is incremented by one from the initial value to one less than the limit. The value of the index is accessable via the word I, and code is any sequence of Forth words. An example :

```
: TEST  DO I . LOOP ;

9 0 TEST        <return>

0 1 2 3 4 5 6 7 8    OK
```

Note that the upper limit of 9 is not displayed.

A variation of the DO ... LOOP is the DO ... +LOOP. This construct allows the user to increment or decrement the count by any value and it looks like this :

: <name> limit initial DO ...code... increment +LOOP ;

limit = is the upper or lower limit for the loop count.
initial = the value where the count is started.
code = any Forth word or words.
increment = any positive or negative value.

```
: TEST  DO I . -1 +LOOP ;      <return>

-5 0 TEST       <return>

0 -1 -2 -3 -4 -5  OK
```

The word LEAVE is used to terminate the loop at the next LOOP or +LOOP. Another version of LEAVE is available called ?LEAVE which carries out a test before LEAVEing a loop.

For nested loops a second index is available, the index J.

An indefinite loop can be created in a number of ways :

```
: <name>  BEGIN ...code... AGAIN ;
```

This structure will execute any code found between the BEGIN and AGAIN words. When AGAIN is reached control is transferred to BEGIN and the code is executed again thus creating an infinite loop.

```
: TEST    BEGIN ." Hello " CR  AGAIN ;

TEST      <return>

Hello
Hello
Hello
Hello
...
...
```

Hello's will continue to be printed for ever and ever if you want. On a lot of machines that would be the end of the matter, as the only way to restart the FORTH would be to switch off and start again. In this case it is only necessary to hit "Escape" and the loop is broken, and control returned to the user.

Another structure is :

```
: <name>  BEGIN ...code... flag  UNTIL ;
```

code = any FORTH word or words.
flag = a logical operation which leaves a true or false value on the stack which is tested by UNTIL. If the flag is true, (non-zero) the loop is terminated, otherwise the execution flow returns to BEGIN and carries on through the loop again. For example :

```
: TEST  BEGIN 1 + DUP DUP . 9 = UNTIL ;

0 TEST    <return>

1 2 3 4 5 6 7 8 9   OK
```

This routine takes the top stack value, (initially a 0) increments it by one, duplicates it twice in order to save the value before displaying it, and then performs the logical operation, a comparison to 9. In this case the 9 is printed out. This is done because the FORTH statements are executed before the UNTIL checks the top value of the stack against the 9.

Yet another structure is :

```
: <name>  BEGIN ..code.. flag WHILE ..code.. REPEAT ;
```

code = can be any FORTH word or words.
flag = is a logical operator which leaves a true or false value for WHILE to test. For example :

```
: TEST BEGIN 1 + DUP DUP . 5 < WHILE ." End " REPEAT ;

0 TEST    <return>

1 End 2 End 3 End 4 End 5
```

Notice that the effect of the test is opposite to that in the BEGIN ... UNTIL structure. Here the loop repeats <u>while</u> something is true (rather than <u>until</u> it´s true).

The indefinite loop structures lend themselves best to cases in which you´re waiting for some external event to happen, such as the closing of a switch or thermostat, or the setting of a flag by another part of an program that is running simultaneously.

14.0 Character Input and Output

Character I/O on the BBC is quite versatile and a number of words are supplied which can be altered by the user if required.

The standard FORTH word used for character output is EMIT. This word takes a byte off the stack and sends it to the output device. The byte on the stack is normally an ASCII character and the output device is normally the video screen. The word EMIT is actually a User Vector and it usually contains the word >OUTPUT. This word behaves in exactly the same way as EMIT, but it always sends output to the video screen.

When the word PRINTER is used, the word >PRTR is placed into EMIT. This has the same effect as >OUTPUT but the character is sent to the printer buffer instead.

There is no reason why the user may not create other destinations for output characters and place that word into EMIT instead. See: MAKE and Section 17.

Input characters are handled in a similar way using the User Vector KEY. This word contains the address of a word which waits on the keyboard until a key is pressed, and then puts the ASCII key value onto the stack. This word is called INPUT>. It also checks for input from other sources, such as a *EXEC file or a soft key. This word is used by the main task to obtain input. Others tasks use the word KYBD> which only reads the keyboard.

A word which is KEY taken a step further is the word ?KEY. This word also waits on the keyboard, but only for one specified character. For example :

```
: AA  BEGIN  ?KEY Z  UNTIL ;
```

When AA is executed, the system will stay in the loop until "Z" is pressed on the keyboard.

The old FORTH word ?TERMINAL has been updated by a word called KEY? which returns a flag if a key has been pressed. This routine does not actually wait on the keyboard, but just has "a quick look" to see if a key has been pressed. This routine only works with the main task ; for background tasks it is vectored to return a zero.

(continued)

(continued)

Another useful word is ?TABS which scans the "TAB" key and returns a true flag if the TAB key is pressed twice in succession. If TAB is pressed once the word just idles round its own loop. This word is used in other words which output a lot of text, for example, and the user may want to include ?TABS somewhere so that the word can be stopped by hitting "TAB" once, started again by hitting any other key, or returned to the keyboard by hitting "TAB" again. ?TABS is already used in words such as VLIST, DLIST, etc.

15.0 The Printer

The user will want to use a printer most often to make listings of FORTH blocks, which would normally contain Source Text used to compile into FORTH words.

In the author's case, the printer being used is an EPSON MX100 on the "Centronics" port. The printer is also fitted with an RS232 Interface for use on a SuperBrain QD, and has been set up so that it does not print "Linefeed" after "Carriage Return". Consequently, on the BBC a *FX command is required to tell the BBC to send "Linefeeds" otherwise the paper does not feed through the machine properly. This is *FX 6,0.

Having set up your printer correctly, it is now only necessary to send an output stream to the printer. This can be done by means of the word PRINTER and undone by means of the word SCREEN. An example :

```
*FX 6,0          <return>

PRINTER  VLIST  SCREEN   <return>
```

This will send output to the printer, the output being a VLIST. After the VLIST is finished, the screen will be re-enabled and the system returned to normal. The problem is that the machine is un-available to the user while the listing is taking place. And if you had any other FORTH, and not Multi-FORTH 83, that would be that !!

With Multi-FORTH 83 we can <u>easily</u> set up the printer as a task and carry on using the machine while the printing is taking place :

```
TASK  PRINTER  VLIST  SCREEN        <return>
```

You may notice that tasking control is passed in "CR" !!
See: Section 19 for details about tasking.

Another example :

```
TASK  PRINTER  1 16 SHOW  SCREEN   <return>
```

which would printout a range of blocks from the Disc.

16.0 Defining Words

The power and flexibility of FORTH stems directly from its ability to create new classes of words and structures as and when required. This is what defining words are all about. There are even words whose sole purpose is to define new defining words!!

The most common defining words are the colon ":" and semicolon ";". In fact if you did not have these two words FORTH would not be a compiler at all and it would become very difficult to write a program at all. These two words are used all the time in FORTH and are discussed in the next section. Words defined with colon and semicolon are known as "Colon Definitions".

While colon and semicolon are powerful words in their own right, the really mind-blowing words are CREATE and DOES>, because these words can be used to specify the compilation and execution behaviour of other new words and that is what defining new defining words is all about.

A variation of CREATE .... DOES> is CREATE .... ;CODE where the high level part following DOES> is replaced by machine-code following ;CODE.

We can not forget the Assembler, and it has its own defining words, equivalent to colon/semicolon called CODE and END-CODE. These words are explained in Section 18.

16.1 Colon / Semicolon

FORTH is different from many other languages in that it allows the user to define his or her own words to extend the language. The user can completely customise a set of words which can then be used in any program.

The basic construct of colon definitions is :

: <name> ...forth words.... ;

In Multi-FORTH 83 <name> is compiled into the dictionary as a word with a specific operation as defined after <name> and before semicolon. The references compiled into the dictionary are normally the code field addresses of the words invoked. Try :

: AVG + 2 / ; <return>

Successfully typing the above example will define a new word called AVG which adds the top two stack items and divides by two and puts the result back onto the stack. In other words, AVG finds the average of two numbers. To execute this word, type :

2 4 AVG . <return>

To forget any word in the dictionary (words that you have defined and that are in RAM) use the word FORGET, providing it is not protected by the word FENCE. Simply type :

FORGET <name> <return>

and the word along with any dictionary entries compiled after <name> will be removed from the dictionary.

You can protect a word in the dictionary from FORGET, by placing its name field address into the user variable FENCE like so :

' AVG FENCE ! <return>

will place the name field address of AVG into FENCE and return a Msg..26 if FORGETting is attempted.

16.2 CREATE ..... DOES>

This is one of the most important and powerful FORTH structures. With it you can define new defining words. What this means is that you can create new types of defining words, and using these words new types of data structures can be produced and great power can be given to the programmer. The format for this word is :

```
: new-defining-word  CREATE definition code DOES> run-time code ;
```

defining-word = the name of the new defining word.
definition code = the code which is executed when the defining-word is used to create a new word.
run-time code = this code is executed when the new word is used as a command word.

It is possible in a CREATE DOES> construct to have no definition code or run-time code. As an example:

```
: ARRAY  CREATE 20 ALLOT DOES> ;
```

Here is an example of the construct with no run-time code. This statement will allow the user to create arrays of ten words (20 ALLOT sets aside twenty bytes in the dictionary, enough for ten 16-bit variables). ARRAY is now a compiling word which is a lot like VARIABLE except that it reserves twenty bytes in the dictionary for user variables instead of just two and also uses no initialiser. An example of how to use ARRAY follows:

```
ARRAY NUMBER   <return>
```

This creates an array called NUMBER which will reserve twenty bytes for number storage in the dictionary. To access these twenty bytes we need some way to reference them, perhaps by placing the address of where they are found in memory onto the stack. It just so happens that this is exactly what executing NUMBER will do for us. NUMBER places the address of the first byte of the twenty bytes ALLOTed to NUMBER onto the stack.

The program can be expanded as shown below. We will create a one dimension array and will allow the user to access any number in the array by placing an index on the stack.

```
: ARRAY1 CREATE 40 ALLOT  DOES> SWAP 2* + ;
```

ARRAY1 is now a defining word which when used will create a twenty word array. Let's make one called XYZ

```
ARRAY1 XYZ      <return>
```

We have now created in the dictionary an array called XYZ. We can insert a number, say 123, into the 11th word in this array by typing:

```
 123 11 XYZ !    <return>
```

What has happened up to this point ? First, a 123 was placed onto the stack. Second, the index 11 was placed on the stack, and third, XYZ is encountered. XYZ first places the address of memory in the dictionary where the array is located and then initiates the execution of the code following DOES>. In this case the top two stack items are swapped (putting the index on the top and the address below it). Next, we double the index with 2* because we are dealing with 16-bit values and address memory in 8-bit bytes. The next thing we do is add the offset of the index to the address already on the stack. After this is done, the stack contains the address of the indexed array member and the value to be stored there. A store (!) will finally put the value in the array.

Another routine could be written to fetch values from the array and would look something like this :

```
11 XYZ @  <return>
```

Self Modifying Data Structures

A remarkable consequence of FORTH's ability to define new defining words is that we may build 'intelligent' data structures ; for example, arrays that automatically maintain averages, or lists that re-order themselves whenever any entry is altered.

To take the first of these examples, suppose we have a 10 element array 'READINGS' defined using a word similar to XYZ of the last example. To compute the arithmetic average of the contents of this array requires adding together all 10 entries and dividing by 10. A special definition could easily be written to do this as follows:

(continued)

```
: AVERAGE                    (take average of array 'READINGS' )
        0
        11 0 DO
              I READINGS @ +
           LOOP
        10 / ;
```

If our FORTH application needed us to calculate an average like this often and for many different arrays then, to simplify the overall program, we should define a new defining word *ARRAY with the averaging function built into the DOES> part of the definition:

```
: *ARRAY              ( 'special' array with running average )
          CREATE
                    DUP ,       ( save array size )
                    0 ,         ( set 'average' to zero )
                    0 DO        ( step through elements )
                        0 ,     ( defining and zeroing  )
                      LOOP
          DOES>
               DUP DUP @        ( get array size        )
               SWAP 4 +         ( point to start of array )
               OVER 0 SWAP
                    0 DO        ( step through array )
                        OVER @ +        ( add up   )
                        SWAP 2 + SWAP   ( bump up pointer )
                      LOOP
               SWAP DROP SWAP /     ( divide by array size )
               OVER 2 + !     ( store average in element 0 )
               2 + SWAP 2 * + ;     ( calculate address )
```

Arrays defined by *ARRAY may be used just like those defined by ARRAY1, for example :

```
10 *ARRAY READINGS

10 1 READINGS !             ( readings(1)=10 )

20 2 READINGS !             ( readings(2)=20 )

1000 10 READINGS !          ( readings(10)=1000 )


2 READINGS ?                ( print contents of readings(2) )
20 OK
```

Which is exactly how we would expect a 10 element array, with elements numbered from 1 to 10 to behave. But typing :

```
0 READINGS ?        103 OK
```

will print the average of the values currently contained in the array ((10+20+1000)/10 = 103). This average will be calculated afresh every time the name of the array 'READINGS' is executed and will always be true however many times we might have altered the values stored in the array.

For example :

```
870 10 READINGS !              ( alter readings(10) to 870 )

50 6 READINGS !                ( set readings(6) to 50 )

0 READINGS ?                   ( new average is 95 )
95 OK
```

and, of course, all arrays defined by *ARRAY will have this function built in !!

17.0 Vectored Execution

Vectors are words whose action may be changed at a later time. They are created by using the defining word VECTOR which works in a similar way to the word VARIABLE. A number of the system variables are defined as vectors and these are referred to as "User Vectors". This is done to allow the user a great deal of flexibility and power as these user vectors can be altered at will to reconfigure the system. The user vectors are listed in Section 21.1e along with their default values (values at power-up) and these will be dealt with in the individual sections of this manual where appropriate.

Vectors are defined in the form :

```
n  VECTOR  <name>
```

where n is the vector number (numbers 0 to 24 hex are already defined, so users should start at 26 hex, 38 decimal). Each vector uses two bytes, so n must always be even. (38, 40, 42, 44, etc.) Up to 24 vectors can be defined in the normal system, so this leaves 5 for the user to define. <name> is the name of the vector in the dictionary.

When vectors are first defined they are not initialised and they will return an error message if you attempt to use them.

```
38  VECTOR  TEST     <return>

TEST       <return>

TEST  Msg..20
```

An action can now be assigned to the user vector called TEST by using the word MAKE. For the purposes of this example we will assign the word . to the vector TEST, although the word to be assigned to a vector can be your own word if desired.

```
MAKE  TEST  .        <return>

56 TEST              <return>

56  Ok
```

TEST is now behaving as if it were the word . which of course is what it is at the moment, until such time as MAKE is used to change the vector contents of TEST.

Section 17.2 will deal specifically with the word MAKE.

17.1 System Vectors

The System Vectors are listed in detail in Appendix 21.1e and each task maintains its own list of User Vectors. The contents of the User Vectors is dependant on the type of task being run at the time.

The User Vectors can be changed at any time to reconfigure the system, but please be careful as the results may not always be what you think they will be. You should only even think about changing the User Vectors if you fully understand the workings of the Multi-FORTH 83 system.

A number of Multi-FORTH 83 words incorporate references to the User Vectors, but the most useful and powerful word is QUIT. The high-level source definition of QUIT in Multi-FORTH 83 would be:

```
: QUIT
          BLK @ IF
                    >IN @ BLK @          (used by WHERE)
               THEN
                    0 BLK !   [
          BEGIN
               PROMPT  RP!  QUERY
               INTERPRET
               STATE @
                   0=  IF    STATUS
                   THEN
          AGAIN  ;
```

The two User Vectors referenced are underlined ; PROMPT normally contains a CR word for the main task and the word TPROMPT for other tasks, and STATUS contains the word .OK which prints the "OK" message and a CR.

The reason the word QUIT is so important is that it is the word which controls the whole of the system. QUIT is the word that is executing when the system is waiting for user input and QUIT scans the keyboard, echoes characters to the screen, looks up words in the dictionary, executes or compiles them, and then prints the "OK" message on the screen. In other words, QUIT forms the Outer Interpreter loop of the whole system. Change the way QUIT behaves and you change the behaviour of the whole system.

Changes to QUIT can make the system "closed" if desired or behave in a completely different manner to normal. The User is advised not to attempt to change QUIT until he has a full understanding of the role that QUIT plays in the Multi-FORTH 83 system.

17.2 Changing Vectors

Changes to User Vectors should be made with great care and only when the user has a full understanding of the nature of the change attempted.

A special word called MAKE is provided change the contents of Vectors automatically. It is used in the form :

MAKE <Vector Name> <word name>

where <Vector Name> is the name of the Vector and <word name> is the name of a FORTH word to replace the contents of the vector.

As an illustration, lets play with the contents of the vector STATUS. As we mentioned before, this word is in the QUIT loop and is responsible for printing out the "OK" message. Lets define another word and put this into STATUS.

```
: .BLURB  CR ." Multi-FORTH 83   Ok" ;

MAKE STATUS .BLURB        <return>
```

Back to normal with :

```
MAKE STATUS .OK      <return>
```

It´s quite a novel effect, isn´t it ? Used in the right way, these features can be quite powerful. Its all up to you, the User.

18.0 The Assembler

Multi-FORTH 83 can assemble machine-language definitions of words. Among the many examples of words defined by machine-language instructions are the operations :

+ - EXECUTE BRANCH SWAP DROP DUP

Such words are called code definitions and are constructed by the use of the resident assembler command CODE. The assembler is not intended for conventional programs. FORTH code routines are distinguished by the fact that all end by returning to the inner interpreter rather than by executing a conventional subroutine return.

Assembler code is, by definition, machine dependant. There are many characteristics of FORTH assemblers that are relatively constant across all the processors on which FORTH has been implemented. The main uses of CODE definitions in FORTH is to interface to the operating system of the machine, to supply device drivers and to replace speed-conscious repetition in higher-level definitions.

18.1 Code Definitions

The FORTH defining word CODE creates a standard dictionary entry whose code field address (CFA) contains the address of the byte that follows (the PFA), in other words the CFA points towards the PFA. (In a high-level definition the CFA points towards the word DOCOLON, which is able to thread the following PFA´s together, the PFA´s of a high-level word being a list of addresses which in turn have the same structure, until a code definition is reached, which contains directly executable machine-code.) In a code definition the PFA is directly executable machine-code. CODE creates a smudged word heading, sets the CONTEXT vocabulary to ASSEMBLER, and sets the number base to HEX.

The form of a CODE definition is :

CODE <name> .... instructions .... code-ending END-CODE

CODE creates the definition, whose name is <name>. It also selects the ASSEMBLER vocabulary, in which the various instruction mnemonics, addressing modes, etc, are defined. These are used to build actual machine instructions, which are laid down in subsequent locations in the dictionary. The code ending is one of several constants, which represent actual machine addresses, all of which ultimately return to FORTH´s address interpreter.

Aside from the dictionary header (NFA) there is no overhead in either space or time within a code definition. All instructions are executed at full machine speed.

Very often a program will spend most of its time executing the same words over and over in a loop. By defining these words in CODE rather than in other FORTH words, the speed of a program can be greatly increased.

One of the advantages of FORTH is that it is easy to mix FORTH with machine-code. The same variable labels and arithmetic operators that are used in FORTH can be used to calculate addresses for opcodes. The same defining words that are used to define high level FORTH words can be used to create macros of any depth. The FORTH assembler uses branching constructions such as IF, ... THEN, which are similar to FORTH branching constructions. Furthermore, parameters can be easily passed between CODE words and other FORTH words on the parameter stack.

18.2 Code Endings

Most FORTH code routines end with a jump to the address interpreter, sometimes after modifying the stack. Exceptions are interrupt routines (which return to the code that was being executed before the interrupt occurred) and routines which initiate processes whose completion will be signaled by an interrupt. These end with a jump to a special word.

The most common ending is NEXT JMP, which simply is a jump to NEXT which is the entry point for the address interpreter. The word NEXT is defined as a constant which returns the address of NEXT in the inner interpreter. This has the effect of continuing to the next word to be executed.

Other code-endings are used to manipulate the stack or to place values onto the stack at the finish of a code word. They are :

POP JMP, - Remove top of stack and continue with next word.

POP2 JMP, - Remove two values from stack and continue with next word.

PUT JMP, - Replace top of stack with the contents of the X register (least significant byte) and the A register (most significant byte) and continue with the next word.

PUSH JMP, - Add the contents of the X and A registers to the stack as one word, (X= lsb, A=msb) and continue with the next word.

PUT0X JMP, - Replace the top of stack with the contents of the X register (lsb), the msb being set to 0.

On entry to Code words the registers are in the following states :

A = undefined.

X = undefined.

Y = 0

On exit A, X, & Y do not matter unless any of the above code-endings are being used (except where they do not effect the stack).

(continued)

(continued)

To fetch the top byte of the Parameter Stack :

```
0    # LDY,
SP  )Y LDA,          (Byte is in A Register)
```

To fetch the top byte of the Return Stack :

```
0    # LDY,
RP  )Y LDA,
```

18.3 Zero Page Usage & System Pointers

Zero Page in Multi-FORTH is assigned as follows :

Zero Page Pointer	Zero Page Name	FORTH Name	Usage
2A Hex		SP	Points to Parameter Stack
2E Hex		IP	Address of the next location for the address interpreter (IP)
22 Hex		WR	Address of the word being referenced; set by NEXT
28 Hex		UP	User Pointer; points to the User Area containing FORTH system variables
24 Hex		TP	Task Record Pointer.
26 Hex		PR	Task Priority.
2C Hex		RP	Return Stack Pointer

Use of the System Pointers

When a word is entered, IP and IP+1 contain the address containing the code field address of the next word to be executed. WR and WR+1 contain the code field address of the word currently being executed. WA returns the address of 16 bytes of Zero Page memory available for use by the code within a code definition.

18.4 Assembler Instructions

To compile a colon definition the interpreter enters a special compile mode, in which the words of the input string are not executed (unless designated as IMMEDIATE). Instead their addresses are placed sequentially in the dictionary. During assembly, on the other hand, the interpreter remains in execute mode. The mnemonics of the assembler are defined as words which, when executed, assemble the corresponding operation code at the next location in the dictionary. As elsewhere in FORTH, operands (addresses or registers) precede operators (instruction mnemonics.) Words that create machine-code end in "," i.e. LDA, or STA,

Depending on the processor, several kinds of instuctions and addressing modes are possible. The decision to use Zero Page or absolute addressing is made automatically, and depends on the magnitude of the operand. The variable called MODE is examined to determine the addressing mode and is set by the words ,X ,Y .A #) X) and)Y

One Byte Opcodes

BRK, RTI, RTS, PHP, CLC, PLP, SEC, PHA, CLI, PLA, SEI, DEY, TYA, TAY, CLV, INY, CLD, INX, SED, TXA, TXS, TAX, TSX, DEX, NOP,

These words compile the appropriate one byte opcode.

Addressing Modes

,X ,Y .A # X))Y) MEM

These words set the addressing mode of the next op-code instruction by changing the value of a variable called MODE.

Op-code and Operand Instructions

ORA, AND, EOR, ADC, STA, LDA, CMP, SBC, ASL, ROL, LSR, ROR, STX, LDX, DEC, INC, BIT, STY, LDY, CPY, CPX, JSR, JMP,

n ---

These words compile the appropriate op-code and operand. The addressing mode is determined by the value in MODE. When the addressing mode is not .A (accumulator), n is expected on the stack and will be compiled as the operand. The magnitude of n determines whether zero page or absolute addressing will be used.

Conditionals

EQ MI CS NOT VS

--- n nl

These words indicate the branching conditions : equal; not equal; plus; minus; carry set; and carry clear. They are used with words such as BEGIN, END, IF, and THEN, to compile branching instructions. n indicates the condition, nl is pushed on the stack for error checking.

18.5 Macros

You may define macros easily in FORTH by using colon-definitions that contain assembler instructions.

For example:

```
ASSEMBLER  HEX    : PPOP
                        0   # LDY,
                        SP )Y LDA, ;

                  : RPOP
                        0   # LDY,
                        RP )Y LDA, ;
```

As these words do not contain proper code-endings they can only be used within a definition that does have a code-ending. I.e. within other code-definition.

Macros are mainly a notational convenience; PPOP is merely the sequence as described in the definition, just as if you had written the expression out in full.

The words used to implement the assembler structures (loops and conditionals) are defined as macros, as are the code endings.

18.6 Logical Structures

Control of logical flow is handled by FORTH's assembler using the same structured approach as high-level FORTH, although the implementation of the commands is necessarily different. The commands even have the same names as their high-level analogues; ambiguity is prevented by the use of separate vocabularies. The following are implemented as standard macros:

BEGIN, Puts an address on the stack (HERE) Used with UNTIL, as in :

05 3 LDY, BEGIN, DEY, 0 = UNTIL, which would compile as :

```
L1:  LDY #03
     DEY
     BNE L1
```

UNTIL, Assembles a conditional jump back to the address left by BEGIN, It is preceded by a condition code. The loop is ended if the condition is met. Common condition codes are 0= and 0<, as appropriate to the various CPUs. Can also be used to compile a branch back to a specific address. E.g. :

```
     4BCF 0< UNTIL,
```

would compile to : BPL $4BCF

AGAIN, Assembles an unconditional jump back to the address left by BEGIN,

WHILE, addr1 n 4 --- addr2 addr1

Used in the form:

```
BEGIN, ... (condition) WHILE, ...
                          (condition) REPEAT,
```

For example:

```
BEGIN, 7000 ,Y LDA, DEY, 0= NOT WHILE,
8000 ,Y STA, 0= NOT REPEAT, NOP,
```

Would compile:

```
L1:  LDA 7000,Y      ; begin loop
     DEY
     BEQ L2:         ; exit loop
```

```
     STA 8000,Y
     BEQ L1:          ; end of loop, branch
L2:  NOP                      always
```

NOT — Negates condition code. Used after conditional words such as EQ to reverse their meaning.

IF, — Assembles a conditional forward jump, to be taken if the preceding condition is false, leaving the address of this instruction on the stack.

E.g. 4000 LDA, 0= IF, BRK, THEN, RTS, NOP,

would compile as:

```
     LDA 4000
     BNE L1:
     BRK
L1:  NOP
```

ELSE, — Provides the destination of IF's jump (whose address was on the stack) and assembles an unconditional forward jump (whose location is left on the stack).

For example:

```
0= IF, 00 # LDY, 0= ELSE, 01 # LDY, THEN,
```

would compile as:

```
     BNE L1:
     LDY #00    ; executed if equal
     BEQ L2:    ; branch always
L1:  LDY #01    ; executed if not equal
L2:  RTS
```

THEN, — Provides the destination for a jump instruction whose location is on the stack at assembly time (left by IF, or ELSE,).

The ELSE clause may be omitted entirely. This construction is functionally analogous to the IF ... ELSE ... THEN construction provided by FORTH's compiler.

Since the locations or destinations of branches are left on the stack at assembly time, the structures may be nested naturally. By manipulating the stack during assembly, however, you can assemble any branching structure.

If you wish to branch forward, use IF to leave the location of the branch on the stack. At the branch´s destination, bring the location back to the top of the stack (if it is not there already) and use ELSE or THEN to complete the branch by filling in its destination at the location that is on the top of the stack.

If you wish to branch back to an address, leave it on the stack with BEGIN. At the branch´s source, bring the address to the top of the stack and use UNTIL or a jump mnemonic to assemble a conditional or unconditional branch back. Be sure to manipulate the branch address before the condition mnemonic since each condition code adds one item to the stack.

18.7 Literals

Some processors allow you to define instructions to reference literals. (6502 for example). For these, the standard FORTH word for identifying a literal is # .

Thus the instruction:

```
1000 # 0 MOV
```

would move the literal 1000 into register 0. A few processors allow a short instruction format for small literals and a long format for larger ones. In such cases the FORTH assembler automatically examines the literal and generates the appropriate format.

18.9 Monitor Calls

There is nothing special in performing calls to external subroutines, provided some rules are followed. You need to be throughly familiar with the machine that you are using, and you require accurate documentation on the machine´s internal routines.

The User should become familiar with the BBC ´User Guide´, Page 450 onwards which deals with the operating system calls.

Operating System Words :

OSBYTE

OSCALL

OSCLI

OSWORD

OSWRCH

19.0 Multi-Tasking

Multi-tasking is a time-multiplexing technique whereby a number of programs can appear to be running simultaneously. Of course they are not all running at once, because the micro-processor is a sequential device. Those who have done some Assembly Language programming will be well aware of this fact!

What is done is that a program will be run for a very short period of time, say one hundredth of a second, and then all the machine registers and pointers relating to that program are preserved somewhere and another program is run for the same period, and so on. It all happens so quickly that to us slow humans it seems to be happening at once.

The beauty of this technique is that programs can be scheduled with respect to a clock and can be repetitive. This becomes very relevant to solving programming problems in the Real-Time world as multi-tasking allows an overall solution to be created in a number of interactive program "modules" (or tasks) and this can be very effective. It is also useful purely from a human convenience viewpoint.

The following sections will describe multi-tasking in some detail. The initial user only needs to use the words RUN and TASK to start with, and even using just these two words allows very powerful and sophisticated applications to be created. Don´t worry if you don´t understand all the details on tasking. Just keep reading it over and over and eventually something will stick. You don´t need to understand all the technical detail to use tasks effectively. The information is really given for the more advanced and experienced user who will be able to make use of the information given.

19.1 How to use Tasks

This section should be read in conjunction with Appendix 21.1c which details the allocation of task memory for both types of tasking words, RUN and TASK.

All the user really needs to get tasks going are two words, RUN and TASK.

RUN is used to start a Non-interpreting task. This is a task which only requires one page (256 bytes) of memory because it is comprised of a pre-compiled word which has already been interpreted during the process of compilation. Also it would not need to interpret the input stream.

An example :

```
: TEST  BEGIN  ." This is a test " CR  AGAIN ;
```

We have now compiled a word called TEST, which is an infinite loop with a string in the middle.

```
RUN  TEST      <return>
```

This will create a task and execute the word TEST, which being an infinite loop will execute forever and also will execute as fast as it can. While TEST is executing the system can be used as normal (well, almost, because continuous text output to the screen is somewhat distracting).

Let's alter TEST to slow it down a bit. We'll insert a time delay by using the word DELAY :

```
: TEST  BEGIN ." This is a test " CR 500. DELAY AGAIN ;

RUN  TEST      <return>
```

TEST will now execute every 5 seconds and we can now use another word called DISPLAY to have a look at the task queue.

```
DISPLAY        <return>

1536()  READY
1552(TEST)  TIMER
```

(continued)

(continued)

1536 represents the address in the Task Queue of the main task. This is 600 Hex. READY is the current status of this task. 1552 is the address (610 Hex) of task TEST in the Task Queue and TIMER means that it is running on its own clock. This is due to the use of the word DELAY.

Should the task TEST need to be stopped the word STOP is used :

```
1552 STOP       <return>
```

This puts the task "to sleep" until it is executed again using the word WAKE.

```
DISPLAY         <return>

1536()  READY
1552(TEST)  SLEEPING
```

So to get task TEST going again :

```
1552 WAKE       <return>
```

On the other hand, we may want to forget task TEST altogether, in which case :

```
1552 KILL       <return>
```

would have the desired effect.

A word which has a similar effect to RUN is the word IN. This is used to schedule a task to RUN sometime in the future. It again uses the task´s clock like DELAY, and an example would be :

```
500. IN  TEST       <return>
```

which schedules TEST to execute in 5 seconds time.

The word which defines an interpretive task is TASK. This allocates 2 Pages (512 Bytes) of memory and is used for programs which require an input buffer in order to be interpretive. There is a very subtle difference between interpretive and non-interpretive tasks and the best way of showing this is by an example.

If, for instance, we wanted to run a buffered printer we could use this sequence with the word TASK :

```
TASK  PRINTER  DLIST  SCREEN
```

This would take two pages of memory because the sequence as shown is interpretive. It could not be run using RUN. If it was rewritten it could.

```
: TEST  PRINTER  DLIST  SCREEN ;

RUN  TEST       <return>
```

This would now take only one page of memory using RUN because the use of the compiler via the colon definition has removed the need for interpretation (as the compiler interprets during compilation).

That is really all there is to be said about RUN and TASK so far as the casual user is concerned. The information given so far will allow the user to use quite complex and sophisticated tasks, and in order to use some of the other tasking words which are described in the following sections, the user will have to have quite a high degree of understanding of the system.

When the disc drive is used, and any other operation which suspends the system interrupts, such as the cursor keys, tasking will also be suspended. This is a limitation of the operating system of the BBC rather than a limitation of Multi-FORTH 83.

If you do not wish another task to interrupt a sequence of instructions or words use GIVE and KEEP at the beginning and end of the sequence. This must be done for words which output control characters to the screen for instance. (Such as when EMITting strings of characters for the VDU function, etc.)

19.2 Concepts used in Task Handling

The information given in this section must be regarded as background information only. The more intelligent user will have no difficulty using the information given, but a more detailed description is beyond the scope of this manual and will be covered in depth in the De-Luxe System Manual in any case.

Task Records

In order to run a separate task the system needs to keep certain information about each task. It does this in the Task Record. See: Section 21.1b. This is a block of 16 bytes of memory which contains various data concerning the task status.

The system fetches such a record by using the word RECORD which returns the starting address of the next Task Record area. This area is defined by the values of the contents of locations 7D8 and 7DA Hex.

7D8 = Number of Task Records available (default = 28)
7DA = Starting address of Task Records (default = 600H)

Therefore memory from 600 to 7BF hex is used by the system to hold Task Records. This may be changed by altering 7D8 and 7DA to another area in memory allowing more than 28 tasks to run !!

Unassigned Task Records are marked with a value of 0 in the first word. RECORD returns the starting address of the first unassigned record and sets up some default values within the record. These are :

Word 0 = 1 - Indicating Task Record is assigned but "sleeping"

Word 1 = -2 - Priority = 2 (All tasks have this priority except the main task which has Priority 10. This number corresponds to the number of clock ticks before the task is swapped out.

Theoretically a Task Record is all that is needed to run a separate task, however if any high-level Forth words are to be executed it is highly likely that the task will need a copy of its own user variables in order to run. The initiating task does this using {ASSIGN}.

{ASSIGN} takes the address of a Task Record from the stack and returns the address of the next memory page available from the "Memory Page List" and assigns that page to the task. ASSIGN does the same as {ASSIGN} but it applies only to the calling task whereas {ASSIGN} can assign memory to any task.

This notation is used for all task words when {} around a word implies that it will do it for any task whereas the word itself will only do it for the calling task.

Memory Pages

In a similar way to RECORD locations 7DC to 7E7 contain values used to assign memory pages.

7DC,7DD - Address of a block of memory containing the "Page Records". These are two byte entries containing the Task Record address of the task to which the corresponding memory page is assigned. It = 0 if the memory page is not assigned. (The default value is 7D0. Subsequent records are lower in memory and memory 7C0 to 7CF is allocated to memory page records being the default system permitting eight memory pages.)

7DE - 1 Byte - Lowest page available for tasks.
7DF - 1 Byte - Highest page available + 1.

These two values describe a block of memory which is set aside as Memory Pages for tasks to use as needed. (In the default system 4 pages (1k) are made available from OSHWM at system start up).

These addresses may be changed (this should be done when only the system task is running) to any part of memory, the only condition being that the memory between the lowest and highest page is a continuous block of memory.

7E0,7E7 - Form the SEMAPHORE called MEMORY with the location 7E6-7E7 containing the number of memory pages available.

This allows tasks to dynamically assign memory pages and wait if none are available using :

```
WAIT  MEMORY TP @ MEM?
```

If ASSIGN is used then if no pages are available the task will not wait. Having assigned a memory page to a task it is then necessary to initialise the user variables and vectors. This is done using {INSTALL}. In fact, a word called PAGE1 has been set up to do all this - fetch a record, assign 1 page to the task and install the user variables. PAGE1 also carries out two other functions :

1/ Copies the current task's vocabulary table into the memory page so that the task can do VLIST etc. The task is still not able to do interpreting as it has no input buffer !! The vocabulary table can be overwritten by the data stack if this is used heavily.

2/ The disc file currently in use is allocated to the task also, so that the task can be used to print out blocks that the main task is using, for example. Consequently two tasks are now using the same file and the main task will not be able to close that file using USE as there will be an "In Use" error. In order to use another file the main task must therefore use F1, F2, etc. and open a file associated with that. See: Section on Disc Buffers.

If a task needs to interpret input whilst running or uses PAD then it needs to assign another memory page to hold the TIB and PAD, etc. After assigning the page the task should call INTERPRETER. This word takes the page address from the stack and sets up TIB and PAD to refer to it. INTERPRETER also changes RP0 and SP0 so that the stacks are relocated to allow more room and to stop the vocabulary table being overwritten. INTERPRETER should be followed by RP! and SP! in order to set up the stack properly.

When a task has been assigned a task record and memory all that is now needed is to start it running. This is done by putting it on the READY or TIMER queues.

Queues

The running of tasks is handled by a set of task queues. These are linked lists to which tasks may be added at the tail (using QUEUE) and removed (using UNQUEUE).

The most important queue is the READY queue which is a list of tasks, excluding the currently running task, which are ready to run. In the normal course of events, the current task will run until it's priority is incremented to 0 by calls to the clock routine on each timer interrupt. At this time, on the next call to the NEXT routine the current task's registers are copied back to it's Task Record and the Task Record is attached to the end of the READY queue.

The task at the top of the READY queue is then started by removing it's Task Record from the READY queue and copying it's registers from the Task Record to zero page. The NEXT routine is then entered which starts the task at the point where it was last running.

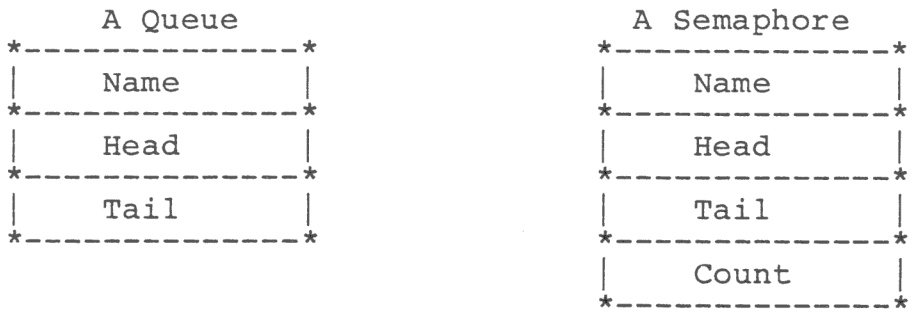
The TIMER queue is another queue in the system. Tasks put in this queue should have a four byte count (lsb in low address) in the "Delay" field of it's Task Record. At each timer interrupt (every 10mS) the system checks each task in the TIMER queue, decrements its delay count and if it is zero the task is removed from the TIMER queue and put on the READY queue.

This is how DELAY works. It takes a double number delay count from the stack and puts it in the current task's Task Record (found from TP), and puts itself in the TIMER queue and stops - using {SLEEP}.

Semaphores

Another type of queue is the Semaphore. There are two semaphores already defined in the system called BUFFER and MEMORY, but others may be defined by the user using the defining word SEMAPHORE. See: SEMAPHORE (in glossary).

A Semaphore is set up in memory like a queue but with an extra field - the count.



NAME points to the address of a counted string giving the name of the queue or semaphore (usually the Name Field Address of the semaphore when defined).

HEAD contains the address of the Task Record of the task at the top of the queue (or 00FF of no task is in the queue).

TAIL contains the address of the Task Record of the task at the end of the queue (or the address of HEAD if no task is in the queue).

COUNT contains the semaphore count as initialised by SEMAPHORE or IS and modified by WAIT and SIGNAL.

When a task executes the sequence :

```
SS   WAIT          (where SS has been defined to be a
                                              semaphore)
```

the count is examined. If the count is > 0 it is decremented and the task continues. If it is = 0 the task is stopped and its Task Record is added to the semaphore's queue.

When the sequence :

```
SS   SIGNAL
```

or the code equivalent :

```
SS    256 MOD   LDX,          \ Load semaphore address
SS    256  /    LDY,          \ to X,Y
      SIGNAL    JSR,
```

is executed by another task, the semaphore's count field is incremented by 1 and the count is checked to see if it was 0. If so the task at the top of the queue (if any) is put on the READY queue to start running.

The address returned by a semaphore or queue is that of the HEAD field.

Task Management

Apart from the normal way in which tasks are started and stopped by semaphores and the timer, they may also be affected by other words.

These words fall into two groups:

1/ Words which affect the current task - i.e. tasks call them themselves :

PAUSE - Causes the task to put itself on the READY queue, save its registers and restart the next task at the head of the READY queue. The effect of this is that the current task is temporarily halted ("swapped out") to give other tasks a chance to run.

SLEEP - The task copies its registers to the Task Record, sets the link field in the Task Record to 1 (which indicates task "sleeping") and moves in the next task from the READY queue and executes it.

DIE - The task DEASSIGNS any memory pages it was using, gives up the use of any block file allocated to it, sets its Task Record link field to 0 (i.e. unallocated) and stops itself using {SLEEP} (The task now no longer exists - it has effectively committed suicide !!)

This is the last word executed by RUN and TASK and is also called when an error occurs in any task other than the main task (unless ABORT is revectored).

KEEP - This sets the priority of the current task to a positive value which stops it being swapped out during the NEXT routine. The task will still be swapped out by PAUSE SLEEP DIE and other words which use them, such as CR.

GIVE - Restores the negative value of the priority field so that the task may be swapped out.

2/ Words which affect other tasks :

STOP - Removes the task from the queue in which it is and sets its Task Record's link field to 1 ("sleeping"). Equivalent to SLEEP.

KILL - Removes the task from its queue, deassigns its memory and block file and deallocates the Task Record. Equivalent to DIE.

WAKE - Puts task on the READY queue where it will start up again. This should only be done to a "Stopped" task.

LOCAL - This allows a task to read or write a user variable in another task's user area. This may be used by one task to monitor what another task is up to.

TSK & GO> - These words are used internally by RUN & IN

DISPLAY - This looks at the allocated Task Records and prints information on each one. For each Task Record whose link field is not zero, (allocated) it uses .TASK to print the contents of the user variable ID . ID contains the address of the name field of the task running, this being set by IN and RUN to the name field of the word to execute. TASK sets ID to 0 indicating "no name". DISPLAY then uses the contents of the user variable TSW. This contains the address of the queue containing the task. Display converts this to the address of the name of the queue and prints it.

DISPLAY would give the following :

```
Task not named     (set up by TASK or main task)
      ↘
1536 (  ) READY
↖                  ↖
Address of         Task is in READY queue
Task Record


1552 ( VLIST ) READY
       ↖
User typed
     RUN VLIST


            Task is waiting in
               TIMER queue
                  ↙
1568 ( NN ) TIMER
       ↖
     Word to execute is NN
```

Words such as RUN IN .TASK set up ID appropriately when used, as do QUEUE DELAY etc, set up TSW. Tasks should always have ID and TSW set properly or DISPLAY will give rubbish.

20.0 FORTH Glossary

Glossary Notation.

Stack Notation:

The stack parameters input to and output from a definition are described using the notation :

before --- after

before	stack parameters before execution
after	stack parameters after execution

In this notation, the top of the stack is to the right. Words may also be shown in context when appropriate.

Unless otherwise noted, all stack notation describes execution time. If it applies at compile time, the line is followed by : (compiling)

Serial Number: These show when appropriate the year of the standard in which these words are defined.

Pronounciation : The natural language pronounciation of word names is given in double quotes (") where it differs from English pronounciation.

Attributes:
Capitalised symbols indicate attributes of the defined words :

C The word may only be used during compilation of a colon definition.

I Indicates that the word is IMMEDIATE and will execute during compilation, unless special action is taken.

M This word has a multi-tasking impact. See Section 19

U A User Variable

V A User Vector

Stack Parameters

Unless otherwise stated, all references to numbers apply to 16-bit signed integers. The implied range of values is shown as {from ... to}. The content of an address is shown by double braces, particularly for the contents of variables, i.e. BASE {{2 ... 72}}.

The following are the stack parameter abbreviations and types of numbers used throughout the glossary. These abbreviations may be suffixed with a digit to differentiate multiple parameters of the same type.

Stack Abbrv.	Number Type	Range in Decimal	Minimum Field
flag	boolean	0=false, else=true	16
true	boolean	-1 (as a result)	16
false	boolean	0	16
b	bit	{0 ... 1}	1
char	character	{0 ... 127}	7
l	length of string	{0 ... 255}	8
8b	8 arbitrary bits (byte)	not applicable	8
16b	16 arbitrary bits	not applicable	16
n	number (weighted bits)	{-32,768..32,767}	16
+n	positive number	{0..32,767}	16
u	unsigned number	{0..65,535}	16
w	unspecified weighted number (n or u)	{-32,768..65,535}	16
addr	address (same as u)	{0..65,535}	16
32b	32 arbitrary bits	not applicable	32
d	double number	{-2,147,483,648... 2,147,483,647}	32
+d	positive double number	{0..2,147,483,647}	32
ud	unsigned double number	{0..4,294,967,295}	32
wd	unspecified weighted double number (d or ud)	{-2,147,483,648.. 4,294,967,295}	32
sys	0,1, or more system dependent stack entries	not applicable	

Any other symbol refers to an arbitrary signed 16-bit integer in the range {-32,768 ... 32,767}, unless otherwise noted.

Because of the use of two's complement arithmetic, the signed 16-bit number (n) -1 has the same bit representation as the unsigned number (u) 65,535. Both of these numbers are within the set of unspecified weighted numbers (w).

Input Text

<name>

An arbitrary FORTH word accepted from the input stream. This notation refers to text from the input stream, not to values on the data stack.

ccc

A sequence of arbitrary characters accepted from the input stream until the first occurence of the specified delimiting character. The delimiter is accepted from the input stream, but it is not one of the characters ccc and is therefore not otherwise processed. This notation refers to text from the input stream, not to values on the data stack. Unless otherwise noted, the number of characters accepted may be from 0 to 255.

References to other words and definitions.

Glossary definitions may refer to other glossary definitions or to definitions of terms. Such references are made using the expression "See :". These references provide additional information which apply as if the information is a portion of the glossary entry using "See :".

Start of Glossary (Note : The Assembler & Editor Glossaries are listed separately)

! 16b addr --- 79 "store"

16b is stored at addr.

+d1 --- +d2 79 "sharp"

The remainder of +d1 divided by the value of BASE is converted to an ASCII character and appended to the output string toward lower memory addresses. +d2 is the quotient and is maintained for further processing. Typically used between <# and #>

#> 32b --- addr +n 79 "sharp-greater"

Pictured numeric output conversion is ended dropping 32b. addr is the address of the resulting output string. +n is the number of characters in the output string. addr and +n together are suitable for TYPE

#S +d --- 0 0 79 "sharp-s"

+d is converted appending each resultant character into the pictured numeric output string until the quotient (see: #) is zero. A single zero is added to the output string if the number was initially zero. Typically used between <# and #>

#TIB --- addr U,83 "number-t-i-b"

The address of a user variable containing the number of bytes in the text input buffer. #TIB is accessed by PARSE when BLK is zero. {{0...capacity of TIB}} See: "input stream"

$ addr n1 n2 -L --- addr n3 n4 -L+1
--- addr n1 n2+L

This word allows pictured numeric input in a similar way to pictured output. Convert 1 char at address addr-L, multiply n1 n2 by BASE and add to form n3 n4 and increment -L. If it cannot convert, L is negated, i.e. +ve 1 indicates an error in conversion. Typically used between <$ and $>

$> s addr n1 n2 L --- n3 n4 true
--- false

This word is used during pictured numeric input in a similar way to pictured output. End conversion of string. If top of stack L does not equal 0 an error occurred so return false, otherwise return double number modified by sign s + a true flag.

$S addr n1 n2 -L1 --- addr n3 n4 +L2

Convert characters until the end of string (after L1 characters) or unconvertable character. In both cases L2 is positive on return.

$LIT --- addr L

Put address and length of in-line string onto the stack. Compiled by $LITERAL

$LITERAL --- addr L
addr L --- (compiling)

Works the same way as LITERAL except that it returns an address and length of a string. Compiles a system word $LIT and the string indicated by address addr and length L so that when executed the address and length of the string are put onto the stack.

' --- addr M,83 "tick"

Used in the form:

' <name>

addr is the compilation address of <name>. An error condition exists (Msg..6) if <name> is not found in the currently active search order.

(--- I,M,83 "paren"
--- (compiling)

Used in the form :

(ccc)

The characters ccc, delimited by) (closing parenthesis), are considered comments. Comments are not otherwise processed. The blank following (is not part of ccc. (may be freely used while interpreting or compiling. The number of characters in ccc may be from zero to the number of characters remaining in the input stream up to the closing parenthesis.

* w1 w2 --- w3 83 "times"

w3 is the least-significant 16 bits of the arithmetic product of w1 times w2.

*/ n1 n2 n3 --- n4 83 "times-divide"

n1 is first multiplied by n2 producing an intermediate 32-bit result. n4 is the floor of the quotient of the intermediate 32-bit result divided by the divisor n3. The product of n1 times n2 is maintained as an intermediate 32-bit result for greater precision than the otherwise equivalent sequence : n1 n2 * n3 / . An error condition results if the divisor is zero or if the quotient falls outside the range {-32,768...32,767}, but no action is taken.

*/MOD n1 n2 n3 --- n4 n5 83 "times-divide-mod"

n1 is first multiplied by n2 producing an intermediate 32-bit result. n4 is the remainder and n5 is the floor of the quotient of the intermediate 32-bit result divided by the divisor n3. A 32-bit intermediate product is used as for */. n4 has the same sign as n3 or is zero. An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768...32,767}, but no action is taken.

\+ w1 w2 --- w3 79 "plus"

w3 is the arithmetic sum of w1 plus w2.

+! w1 addr --- 79 "plus-store"

w1 is added to the w value at addr using the convention for + . This sum replaces the original value at addr.

+- n1 n2 --- n3

Apply the sign of n2 to n1, which is left as n3.

+BUFS addr n ---

Generate n buffers at address addr and add them to the buffer list.

+LOOP n --- C,I,83 "plus-loop"
sys --- (compiling)

n is added to the loop index. If the new index was incremented across the boundry between limit-1 and limit, then the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding DO. sys is balanced with its corresponding DO. See: DO

, 16b --- 79 "comma"

ALLOT space for 16b then store 16b at HERE -2 .

- w1 w2 --- w3 79 "minus"

w3 is the result of subtracting w2 from w1.

--> --- I,M,79 "next-block"

Continue interpretation on the next sequential block. May be used within a colon definition that crosses a block boundary.

-? 0 addr1 n1 n2 -L1 --- s addr2 n1 n2 -L2

This word is used during pictured numeric input. It checks for the "-" character as the next character in the input string and if found changes the sign flag s (under addr) to -1 and alters the string length L appropriately. See: <$ $ $S $>

-ROLL +n ---

The top stack item, not including n, is removed and placed n places down, moving the remaining values into the vacated position. {0...the number of elements on the stack-1}. This word is the same as ROLL, but the direction of movement is reversed. See: ROLL

-ROT 16b1 16b2 16b3 --- 16b3 16b1 16b2

The top three stack entries are rotated, bringing the top to the deepest. This word is the same as ROT, but the direction of movement is reversed. See: ROT

-TRAILING addr +n1 --- addr +n2 79 "dash-trailing"

The character count +n1 of a text string beginning at addr is adjusted to exclude trailing spaces. If +n1 is zero, the +n2 is also zero. If the entire string consists of spaces, then +n2 is zero.

. n --- M,79 "dot"

The absolute value of n is displayed in a free field format with a leading minus sign if n is negative.

." --- C,I,83 "dot-quote"
--- (compiling)

Used in the form :

." ccc"

Later execution will display the characters ccc up to but not including the delimiting " (close-quote). The blank following ." is not part of ccc.

.(--- I,M,83 "dot-paren"
--- (compiling)

Used in the form :

.(ccc)

The characters ccc up to but not including the delimiting) (closing-parenthesis) are displayed. The blank following .(is not part of ccc.

.? addr1 n1 n2 +L1 --- addr2 n1 n2 -L2

This word is used during pictured numeric input. It checks for the "." character in the next character in the input string and if found sets DPL to 0 and alters the string length n1 appropriately. See: <$ $ $S $>

.LINE line blk ---

Output a line of text from the disc by its line and block number. Trailing blanks are suppressed.

.OK ---

Prints out the string "OK". This word is contained in the user vector STATUS, which is part of the QUIT loop. See: Section 17.

.R n +n --- M,83 "dot-r"

n is converted using BASE and then displayed right aligned in a field +n characters wide. A leading minus sign is displayed if n is negative. If the number of characters required to display n is greater than +n, then that number of characters is displayed.

.TASK ---

This word is used by DISPLAY and ERROR to print out information on a task. See: Section 19.

/ n1 n2 --- n3 83 "divide"

n3 is the floor of the quotient of n1 divided by the divisor n2. An error condition results if the divisor is zero or if the quotient falls outside the range {-32,768...32,767}, but no action is taken.

/MOD n1 n2 --- n3 n4 83 "divide-mod"

n3 is the remainder and n4 the floor of the quotient of n1 divided by the divisor n2. n3 has the same sign as n2 or is zero. An error condition results if the divisor is zero or if the quotient falls outside the range {-32,768...32,767}, but no action is taken.

0 --- n
1
2 These small numbers are used so often that it is attractive
3 to define them by name in the dictionary as constants in order to save search time.

0< n --- flag 83 "zero-less"

flag is true if n is less than zero (negative).

0= w --- flag 83 "zero-equals"

flag is true if w is zero.

0> n --- flag 83 "zero-greater"

flag is true if n is greater than zero.

0PAGE --- addr

Returns address addr of 32 bytes of workspace in zero page used when calling O.S routines. Actual value returned is 0. See: Sections 5.1 and 19.

1+ w1 --- w2 79 "one-plus"

w2 is the result of adding one to w1 according to the operation of + .

1- w1 --- w2 79 "one-minus"

w2 is the result of subtracting one from w1 according to the operation of - .

2! 32b addr --- 79 "two-store"

32b is stored at addr.

2* w1 --- w2 83 "two-times"

w2 is the result of shifting w1 left one bit. A zero is shifted into the vacated bit position.

2+ w1 --- w2 79 "two-plus"

w2 is the result of adding two to w1 according to the operation of + .

2, 32b ---

ALLOT space for 32b then store 32b at HERE -4. The double number version of , See: ,

2- w1 --- w2 79 "two-minus"

w2 is the result of subtracting two from w1 according to the operation of - .

2/ n1 --- n2 83 "two-divide"

n2 is the result of arithmetically shifting n1 right one bit. The sign is included in the shift and remains unchanged.

2@ addr --- 32b 79 "two-fetch"

32b is the value at addr.

2CONSTANT 32b --- M,83 "two-constant"

A defining word executed in the form :

32b 2CONSTANT <name>

Creates a dictionary entry for <name> so that when <name> is later executed, 32b will be left on the stack.

2DROP 32b --- 79 "two-drop"

32b is removed from the stack.

2DUP 32b --- 32b 32b 79 "two-dupe"

Duplicate 32b.

2OVER 32b1 32b2 --- 32b1 32b2 32b3 79 "two-over"

32b3 is a copy of 32b1.

2LITERAL --- 32b C,I
32b --- (compiling)

Typically used in the form :

[32b] 2LITERAL

If compiling, compile a stack double number, 32b, into a literal. Later execution of the definition containing this literal will push it to the stack. If executing, the number will remain on the stack. This is the double number version of LITERAL. See: LITERAL

2ROT 32b1 32b2 32b3 --- 32b2 32b3 32b1 79 "two-rote"

The top three double numbers on the stack are rotated, bringing the third double number to the top of the stack.

2SWAP 32b1 32b2 --- 32b2 32b1 79 "two-swap"

The top two double numbers are exchanged.

2VARIABLE --- M,79 "two-variable"

A defining word executed in the form :

2VARIABLE <name>

A dictionary entry for <name> is created and four bytes are ALLOTted in its parameter field. This parameter field is to be used for the contents of the variable. The application is reponsible for initialising the contents of the variable which it creates. When <name> is later executed, the address of its parameter field is placed on the stack. See : VARIABLE

4+ w1 --- w2

w2 is the result of adding four to w1 according to the operation of +.

: --- sys M,79 "colon"

A defining word executed in the form :

: <name> ... ;

A word definition for <name> is created in the compilation vocabulary and the compilation state is set. The search order is changed so that the first vocabulary in the search order is replaced by the compilation vocabulary. The compilation vocabulary is unchanged. The text from the input stream is subsequently compiled. <name> is called a "colon definition". The newly created word definition for <name> cannot be found in the dictionary until the corresponding ; or ;CODE is successfully processed.

An error condition exists if a word is not found and cannot be converted to a number (Msg..6) or if, during compilation from mass storage, the input stream is exhausted before encountering ; or ;CODE. sys is balanced with its corresponding ; . If no <name> is given, Msg..7 occurs.

; --- C,I,79 "semi-colon"
sys --- (compiling)

Stops compilation of a colon definition, allows the name of this colon definition to be found in the dictionary, sets interpret state and compiles EXIT. sys is balanced with its corresponding : .

;CODE --- C,I,79 "semi-colon-code"
sys1 --- sys2 (compiling)

Used in the form :

: <namex> ... <create> ... ;CODE ... END-CODE

Stops compilation, terminates the defining word <namex> and executes ASSEMBLER. When <namex> is executed in the form :

<namex> <name>

to define the new <name>, the execution address of <name> will contain the address of the code sequence following the ;CODE in <namex>. Execution of any <name> will cause this machine code sequence to be executed. sys1 is balanced with its corresponding : sys2 is balanced with its corresponding END-CODE. See : CODE DOES>

< n1 n2 --- flag 83 "less-than"

flag is true if n1 is less than n2

```
-32768 32767 <      must return true.
-32768 0  <         must return true.
```

<# --- 79 "less-sharp"

Initialise pictured numeric output conversion. The words :

#> #S <# HOLD SIGN

can be used to specify the conversion of a double number into an ASCII text string stored in right-to-left order.

<$ addr1 L --- s addr2 0 0 -L

This word is used to initialise pictured numeric input.

s = flag set to false = sign of result.
addr2 = address of last character in string +1 (= a1+L)
0 0 = double number to be added to.
-L = negative length of string.
addr1 = address of character string to convert.
L = length of character string to convert.
See: $ $S $>

<;CODE> ---

The run-time word compiled by ;CODE. See : ;CODE.

<+LOOP> n ---

The run-time word compiled by +LOOP, which increments the loop index by n and tests for loop completion. See: +LOOP

<?LEAVE> ---

Run-time word compiled by ?LEAVE. See: ?LEAVE

<DEFER> ---

The run-time word compiled by DEFER{. See: DEFER{

<DO> w1 w2 ---

The run-time word compiled by DO which moves the loop control parameters to the return stack. See: DO

<LEAVE> ---

The run-time word compiled by LEAVE which removes the loop control parameter from the return stack and jumps to the word following the end of the loop in which it is contained. See: LEAVE

<LOOP> ---

The run-time word compiled by LOOP which increments the loop index and tests for loop completion. See: LOOP

<MARK --- addr C,83 "backward-mark"

Used at the destination of a backward branch. addr is typically only used by <RESOLVE to compile a branch address. Used for the definition of control structures.

<RESOLVE addr --- C,83 "backward-resolve"

Used at the source of a backward branch after either BRANCH or ?BRANCH. Compiles a branch address using addr as the destination address. Used for the definition of control structures.

= w1 w2 --- flag 83 "equals"

flag is true if w1 is equal to w2.

`>` n1 n2 --- flag 83 "greater-than"

flag is true if n1 is greater than n2.

`><` 16b1 --- 16b2 "byte-swap"

Swap the high and low bytes within 16b1.

`>BODY` addr1 --- addr2 83 "to-body"

addr2 is the parameter field address coresponding to the compilation address addr1.

`>BUFF` n1 n2 ---

Store byte n2 in operating system I/O buffer n1. See: OSBYTE call no. 138.

`>DSK` addr ---

Writes the buffer contents to disc if the UPDATE bit is set. addr = address of link field of buffer.

`>EOL` addr1 L1 --- addr2 L2

Given the address and length of the next word in the input stream, this word readjusts the input stream by parsing for the End-of-Line character (held in the user variable EOL) and returns the address and length of the line. If no EOL character occurs the end of line is defined to be the end of the current input stream if input is from the terminal or the end of the current 64 character screen line if the input is from blocks.

`>IN` --- addr U,79 "to-in"

The address of a user variable which contains the present character offset within the input stream {{ 0 ... the number of characters in the input stream}}.

`>LINK` addr1 --- addr2 "to-link"

addr2 is the link field address corresponding to the compilation address addr1.

>MARK --- addr C,83 "forward-mark"

Used at the source of a forward branch. Typically used after either BRANCH or ?BRANCH. Compiles space in the dictionary for a branch address which will later be resolved by >RESOLVE. Used in defining control structures.

>NAME addr1 --- addr2 "to-name"

addr2 is the name field address corresponding to the compilation address addr1.

>OUTPUT n ---

Send byte n to current BBC operating system output stream(s). (As selected by *FX 3).

>PRTR char ---

Send char to printer only.

>R 16b --- C,79 "to-r"

Transfers 16b to the return stack.

>RESOLVE addr --- C,83 "forward-resolve"

Used at the destination of a forward branch. Calculates the branch address (to the current location in the dictionary) using addr and places this branch address into the space left by >MARK. Used during definition of control structures.

? addr --- 79

Display the number at addr, using the format of "." Equivalent to @ .

?BRANCH flag --- C,83 "question-branch"

When used in the form: COMPILE ?BRANCH a conditional branch operation is compiled. See BRANCH for further details. When executed, if flag is false, the branch is performed as with BRANCH. When flag is true, execution continues at the compilation address immediately following the branch address.

?BUFF --- M

Puts the buffer currently being used by this task into the buffer list and then waits until a buffer is free. Allows other tasks waiting for a buffer to continue.

?COMP ---

Issue an error message (Msg..8) if not compiling.

?CSP ---

Issue an error message (Msg..12) if the stack position differs from the value saved in CSP. Used in ;

?DUP 16b --- 16b 16b 79 "question-dupe"

Duplicate 16b if it is non-zero.

?ERROR f n ---

Issue an error message number n, {{0 .... 255}} if the boolean flag is true.

?EXEC ---

Issue an error message (Msg..9) if not executing.

?KEY --- f
--- (compiling)

Returns a flag if the key corresponding to the next non-blank character in the input stream is pressed.

```
: AA BEGIN ?KEY Z UNTIL ;
```

When AA is executed, the system will stay in the loop until "Z" is pressed on the keyboard.

?LEAVE f ---
--- (compiling)

A conditional version of LEAVE. LEAVE's if flag f is true.
See: LEAVE

?LOAD ---

Issue an error message (Msg..10) if not loading.

?PAIRS n1 n2 ---

Issue an error message (Msg..11) if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK ---

Issue an error message (Msg..1) if the stack is out of bounds.

?TABS --- f

Returns a true flag if the TAB key is pressed twice in succession. When TAB is pressed once the word loops until another key is pressed. If this is a TAB a true flag is returned otherwise a false flag is returned. This is used in words such as VLIST when TAB stops the VLIST, and any key restarts it, but TAB TAB stops VLIST.

@ addr --- 16b 79 "fetch"

16b is the value at addr.

ABORT 79,V

Clears the data stack and performs the function of QUIT. No message is displayed. This function may be vectored. See: MAKE.

ABORT" flag --- C,I,83 "abort-quote"
--- (compiling)

Used in the form :

flag ABORT" ccc"

When later executed, if flag is true the characters ccc, delimited by " (close-quote), are displayed and then an error Msg..19 is performed. If flag is false, the flag is dropped and execution continues. The blank following ABORT" is not part of ccc.

ABS n --- u 79 "absolute"

u is the absolute value of n. If n is -32,768 then u is the same value.

AGAIN --- C,I
sys --- (compiling)

Effect an unconditional jump back to the start of a BEGIN-AGAIN loop. sys is balanced with its corresponding BEGIN. See : BEGIN.

ALLOT w --- 79

Allocates w bytes in the dictionary. The address of the next available dictionary location is updated accordingly.

ALSO --- ONLY

The transient vocabulary becomes the first vocabulary in the resident portion of the search order. Up to the last two resident vocabularies will also be reserved, in order, forming the resident search order.

AND 16b1 16b2 --- 16b3 79

16b3 is the bit-by-bit logical 'and' of 16b1 with 16b2.

ASCII --- char I,M "as-key"
--- (compiling)

Used in the form : ASCII ccc

where the delimiter of ccc is a space. char is the ASCII character value of the first character in ccc. If interpreting, char is left on the stack. If compiling, compile char as a literal so that when the colon definition is later executed, char is left on the stack.

ASCII? addr n1 n2 -L --- addr n1 n2 -L f
--- (compiling)

Used in the form : ASCII? ccc

This word is used for pictured numeric input. It compares the current character in the input string with the first character in ccc and returns flag f which is true if the characters are equal.

ASSEMBLER --- 83

Execution of this word replaces the first vocabulary in the search order with the ASSEMBLER vocabulary. See : VOCABULARY

ASSIGN See: Section 19

AWORD --- addr U

The address of a user variable containing the address of the last string parsed in the input stream. See: LWORD PARSE

BASE --- addr U,83

The address of a user variable containing the current numeric conversion base. {{2...72}}

BEGIN ---
--- sys (compiling) C,I,79

Used in the form :

BEGIN ... flag UNTIL

or

BEGIN ... flag WHILE ... REPEAT

BEGIN marks the start of a word sequence for repetitive execution. A BEGIN-UNTIL loop will be repeated until flag is true. A BEGIN-WHILE-REPEAT loop will be repeated until flag is false. The words after UNTIL or REPEAT will be executed when either loop is finished. sys is balanced with its corresponding UNTIL or WHILE.

BL --- 32 79 "b-l"

Leave the ASCII character value for space (decimal 32).

BLANK addr u --- 83

u bytes of memory beginning at addr are set to the ASCII character value for space. No action is taken if u is zero.

BLK --- addr U,79 "b-l-k"

The address of a user variable containing the number of the mass storage block being interpreted as the input stream. If the value of BLK is zero the input stream is taken from the text input buffer. {{0...the number of blocks available - 1}}.

BLK? n --- n addr f

Searches the buffer list for block n of the current file. addr = address of link field of buffer before the one containing block n or the last buffer in the list if block n is not found. f = true if block n is found otherwise false.

BLOCK u --- addr M,83,V

addr is the address of the assigned buffer of the first byte of block u. If the block occupying that buffer is not block u and has been UPDATEd, it is transferred to mass storage before assigning the buffer. If block u is not already in memory, it is transferred from mass storage into an assigned block buffer. A block will not be assigned to more than one buffer accessible by a task. Only data within the last buffer referenced by BLOCK or BUFFER is valid. The contents of a block buffer must not be changed unless the change may be transferred to mass storage. When using discs if block u is not available within the current file an attempt is made to extend the file up to and including block u. This function may be vectored. See: Section 17, and MAKE

BODY> addr1 --- addr2 "from-body"

addr2 is the compilation address corresponding to the parameter field address addr1.

BRANCH --- C,83

When used in the form: COMPILE BRANCH an unconditional branch operation is compiled. A branch address must be compiled immediately following this compilation address. The branch address is typically generated by following BRANCH with <RESOLVE or >MARK. Used during the definition of control structures.

BS --- addr U

The address of a user variable containing the character used to denote "backspace".

BUFF> n1 --- n

Fetch next byte from BBC operating system I/O buffer n1. See: OSBYTE call no. 145.

BUFF? n1 --- n2

Test BBC operating system buffer n1. n2 is the number of bytes available for input buffers and space left in output buffers. See: OSBYTE call no. 128.

BUFFER u --- addr M,83

Assigns a block buffer to block u. addr is the addr of the first byte of the block within its buffer. This function is fully specified by the definition for BLOCK except that if the block is not already in memory it is not transferred from mass storage. The contents of the block buffer assigned to block u by BUFFER are unspecified.

BUFFERS n ---

Generates n more buffers on the buffer list using dictionary space and increments dictionary pointer, DP, accordingly.

C! 16b addr --- 79 "c-store"

The least-significant 8 bits of 16b are stored into the byte at addr.

C, 16b --- 83 "c-comma"

ALLOT one byte then store the least-significant 8 bits of 16b at HERE 1-

C/L --- n

A constant which leaves the number of characters per line on an editing screen.

C@ addr --- 8b 79 "c-fetch"

8b is the contents of the byte at addr.

CLG ---

This word clears the graphics screen.

CLOSE n ---

Close file with handle n. If n=0 all files are closed.

CLS ---

This word will clear the text screen.

CMOVE addr1 addr2 u --- 83 "c-move"

Move u bytes beginning at address addr1 to addr2. The byte at addr1 is moved first, proceeding toward high memory. If u is zero nothing is moved.

CMOVE> addr1 addr2 u --- 83 "c-move-up"

Move the u bytes at address addr1 to addr2. The move begins by moving the byte at (addr1 plus u minus 1) to (addr2 plus u minus 1) and proceeds to successively lower addresses for u bytes. If u is zero nothing is moved. (Useful for sliding a string towards higher addresses).

CODE --- sys M,83

A defining word executed in the form :

CODE <name> ... END-CODE

Creates a dictionary entry for <name> to be defined by a following sequence of assembly language words. Words thus defined are called code definitions. This newly created word definition for <name> cannot be found in the dictionary until the corresponding END-CODE is successfully processed (See : END-CODE). Executes ASSEMBLER. sys is balanced with its corresponding END-CODE.

COLOUR n ---

This word is used to set the colour attributes. See: Section 5.4

COMPILE --- C,83

Typically used in the form :

: <name> ... COMPILE <namex> ... ;

When <name> is executed, the compilation address compiled for <namex> is compiled and not executed. <name> is typically immediate and <namex> is typically not immediate.

CONSTANT 16b --- M,83

A defining word executed in the form :

16b CONSTANT <name>

Creates a dictionary entry for <name> so that when <name> is later executed, 16b will be left on the stack.

CONTEXT --- addr U,79

The address of the dictionary search order table.

CONVERT +d1 addr1 --- +d2 addr2 79

+d2 is the result of converting the characters within the text beginning at addr1+1 into digits, using the value of BASE, and accumulating each into +d1 after multiplying +d1 by the value of BASE. Conversion continues until an unconvertable character is encountered. addr2 is the location of the first unconvertable character.

COUNT addr1 --- addr2 +n 79

addr2 is addr1 +1 and +n is the length of the counted string at addr1. The byte at addr1 contains the byte count +n. Range of +n is {0...255}.

CR --- M,79 "c-r"

Displays a carriage-return and line-feed. This function may be vectored. See: Section 17, and MAKE

CR/W addr n ch f ---

The word invoked by R/W to handle block transfer to and from the cassette and ROM filing systems. See: R/W

CREATE --- M,79

A defining word executed in the form :

CREATE <name>

Creates a dictionary entry for <name>. After <name> is created, the next available dictionary location is the first byte of <name>'s parameter field. When <name> is subsequently executed, the address of the first byte of <name>'s parameter field is left on the stack. CREATE does not allocate space in <name>'s parameter field.

CSP --- addr U

The address of a user variable temporarily containing the stack pointer position for compilation error checking.

CURRENT --- addr U,79

The address specifying the vocabulary in which new word definitions are appended.

D+ wd1 wd2 --- wd3 79 "d-plus"

wd3 is the arithmetic sum of wd1 plus wd2.

D+- d1 n --- d2

Apply the sign of n to the double number d1, leaving it as d2.

D- wd1 wd2 --- wd3 79 "d-minus"

wd3 is the result of subtracting wd2 from wd1.

D. d --- M,79 "d-dot"

The absolute value of d is displayed in a free field format. A leading negative sign is displayed if d is negative.

D.R d +n --- M,83 "d-dot-r"

d is converted using the value of BASE and then displayed right aligned in a field +n characters wide. A leading minus sign is displayed if d is negative. If the number of characters required to display d is greater than +n, then that number of characters is displayed.

D0= wd --- flag 83 "d-zero-equals"

flag is true if wd is zero.

D2/ d1 --- d2 83 "d-two-divide"

d2 is the result of d1 arithmetically shifted right one bit. The sign is included in the shift and remains unchanged.

D< d1 d2 --- flag 83 "d-less-than"

flag is true if d1 is less than d2 according to the operation of < except extended to 32 bits.

D= wd1 wd2 --- flag 83 "d-equal"

flag is true if wd1 equals wd2.

D> d1 d2 --- flag

flag is true if d1 is greater than d2 according to the operation of > except extended to 32 bits.

DABS d --- ud 79 "d-absolute"

ud is the absolute value of d. If d is -2,147,483,648 then ud is the same value.

DEASSIGN See: Section 19.

DECIMAL --- 79

Set the input-output numeric conversion base to ten.

DEF? addr L --- addr L true
--- false

Used by INTERPRET to check if the string given by a,l is a valid word name in the dictionary. I.e. is it a defined word.

false indicates word was found and interpreted or compiled.

DEFINITIONS --- 79

The compilation vocabulary is changed to be the same as the first vocabulary in the search order.

DELAY 32b --- M

This word takes a double number, 32b, and idles for that number of clock ticks. (1 tick = 10mS)

DEPTH --- +n 79

+n is the number of 16 bit values contained in the data stack before +n was placed on the stack.

DIE See: Section 19.

DIGIT c n1 --- n2 tf (ok)
c n1 --- ff (bad)

Converts the ASCII character c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, it leaves only a false flag.

DISPLAY ---

This word displays the system state showing the address, name and status of all currently defined tasks. See: Section 19 for more details.

DLIST ---

This word lists the word names of all the vocabularies in the vocabulary table, in the order of chaining, with the most recently defined word first.

DMAX d1 d2 --- d3 79 "d-max"

d3 is the greater of d1 and d2.

DMIN d1 d2 --- d3 79 "d-min"

d3 is the lesser of d1 and d2.

DNEGATE d1 --- d2 79 "d-negate"

d2 is the two's complement of d1.

DO w1 w2 --- C,I,83
--- sys (compiling)

Used in the form :

DO ... LOOP

or

DO ... +LOOP

Begins a loop which terminates based on control parameters. The loop index begins at w2, and terminates based on the limit w1. See LOOP and +LOOP for details on how the loop is terminated. The loop is always executed at least once. For example : w DUP DO ... LOOP executes 65,536 times. sys is balanced with its corresponding LOOP or +LOOP.

An error condition exists if insufficient space is available for at least three nesting levels, but no action is taken.

DOES> --- addr C,I,83 "does"
--- (compiling)

Defines the execution-time action of a word created by a high-level defining word. Used in the form :

: <namex> ... <create> ... DOES> ... ;

and then

<namex> <name>

where <create> is CREATE or any user defined word which executes CREATE.

DOES> marks the termination of the defining part of the defining word <namex> and then begins the definition of the execution-time action for words that will later be defined by <namex>. When <name> is later executed, the address of <name>'s parameter field is placed on the stack and then the sequence of words between DOES> and ; are executed.

DP --- addr U

The address of a user variable containing the Dictionary Pointer.

DPL --- addr U "d-p-l"

The address of a user variable containing the number of places after the decimal point after input number conversion.

DR/W addr n ch f ---

The word invoked by R/W to handle block transfer to and from the disc and other random access filing systems.
See: R/W

DROP 16b --- 79

16b is removed from the stack.

DUP 16b --- 16b 16b 79 "dupe"

Duplicate (or copy) 16b.

DU< udl ud2 --- flag 83 "d-u-less"

flag is true if udl is less than ud2. Both numbers are unsigned.

EDIT +n ---

This word invokes the built-in editor, to edit screen n. A disc file must have been opened first with USE otherwise an error condition exists. See Section 6.

EDITOR --- 83

Execution of this word replaces the first vocabulary in the search order with the EDITOR vocabulary. See VOCABULARY.

ELSE --- C,I,79
sys1 --- sys2 (compiling)

Used in the form :

flag IF ... ELSE ... THEN

ELSE executes after the true part following IF. ELSE forces execution to continue at just after THEN. sys1 is balanced with its corresponding IF. sys2 is balanced with its corresponding THEN.

EMIT 16b --- M,83,V

The least-significant 8-bits of 16b are sent to the current operating system output stream to be displayed. This word may be vectored and normally contains the word >OUTPUT.
See: >OUTPUT MAKE

EMPTY-BUFFERS --- M,79 "empty-buffers"

Unassign all block buffers. UPDATEed blocks are not written to mass storage. See : BLOCK.

END-CODE sys --- 79 "end-code"

Terminates a code definition and allows the <name> of the corresponding code definition to be found in the dictionary. sys is balanced with its corresponding CODE or ;CODE. See : CODE.

ENVELOPE See: Section 5.3

EOL --- addr U

The address of a user variable containing the character used to denote "end-of-line".

EOR --- addr U

The address of a user variable containing the character used to denote "end-of-record".

ERASE addr u --- 79

u bytes of memory beginning at addr are set to zero. No action is taken if u is zero.

ERROR n --- V

Generates error message n. This function may be vectored. See: Section 17 and MAKE

EXECUTE addr --- 79

The word definition indicated by addr is executed. An error condition exists if addr is not a compilation address which will probably cause the system to crash.

EXIT --- C,79

Compiled within a colon definition such that when executed, that colon definition returns control to the definition that passed control to it by returning control to the return point on the top of the return stack. An error condition exists if the top of the return stack does not contain a valid return point. May not be used within a DO-LOOP.

EXPECT addr +n --- M,83

Receive characters and store each into memory. The transfer begins at addr proceeding towards higher addresses one byte per character until either an "end-of-record" is received or until +n characters have been transferred. No more than +n characters will be stored. The "end-of-record" is not stored into memory. No characters are received or transferred if +n is zero. All characters actually received and stored into memory will be displayed, with the "end-of-record" displaying as a space. See : SPAN

EXT n --- d

Returns a double number d representing the length of the file (in bytes) with file handle n.

EXTRACT char addr1 L1 --- addr2 L2

Used in PARSE to find address and length of the next word in the string at address addr1, length L1 delimited by char.

F0 ---
F1
F2 Select file (defined using USE) to be used for block I/O.
F3 The filing system may have to be changed also to that appropriate to the file. See: USE

FALSE --- 0

This word places a logical "false" value onto the stack.

FDEC addr ---

Decrement task count of file. addr = address held in FILE.

FENCE --- addr U

The address of a user variable containing a name field address below which FORGETting is trapped. To forget below this point the user must alter the contents of FENCE.

FETCH addr1 --- addr2

Removes the buffer pointed to by address addr1 from the buffer list and puts the address of the data part of the buffer in TOP, returning the link address of the buffer.

FF ---

This word executes PAUSE and then outputs ASCII 12, (0C) which is the "Form-feed" character.

FINC addr ---

Increment task count of file. addr = address held in FILE.

FILE --- addr U

The address of a user variable containing the address of byte containing the DFS File Handle for the current block file.

FILL addr u 8b --- 83

u bytes of memory beginning at addr are set to 8b. No action is taken if u is zero.

FIND addr1 --- addr2 n 83

addr1 is the address of a counted string. The string contains a word name to be located in the currently active search order. If the word is not found, addr2 is the string address addr1, and n is zero. If the word is found, addr2 is the compilation address and n is set to one of two non-zero values. If the word found has the immediate attribute, n is set to one. If the word is non-immediate, n is set to minus one (true).

FLUSH --- M,83

Performs the function of SAVE-BUFFERS then un-assigns all block buffers. (This may be useful for mounting or changing mass storage media).

FORGET --- M,83

Used in the form :

FORGET <name>

If <name> is found in the compilation vocabulary, delete <name> from the dictionary and all words added to the dictionary after <name> regardless of their vocabulary. Failure to find <name> is an error condition. An error condition also exists if the compilation vocabulary is deleted.

FORTH --- 83

The name of the primary vocabulary. Execution replaces the first vocabulary in the search order with FORTH. FORTH is initially the compilation vocabulary and the first vocabulary in the search order. New definitions become part of the FORTH vocabulary until a different compilation vocabulary is established. See : VOCABULARY.

FORTH-83 --- 83

Assures that a FORTH-83 Standard System is available, otherwise an error condition exists. (Msg..6)

FREEZE ---

Sets TOP to zero so that buffer is effectively 'lost'. May be used to hold on to a buffer for some reason. The buffer is also marked as unassigned so that if it is later added to the buffer list (using THAW) the contents will be ignored. See: Section 19.

FS --- n

Returns a value corresponding to the Filing System in use.

0 = No Filing System
1 = 1200 Baud Cassette
2 = 300 Baud Cassette
3 = ROM Filing System
4 = DISC Filing System
5 = ECONET
6 = TELESOFTWARE

See: OSARGS Page 337 Advanced User Guide.

FSET n ---

Set file to file number n. I.e. F0 is 0 FSET etc.

GCOL n ---

Used to set the graphics colour attributes. See: Section 5.4

GIVE ---

Allows a Task to be swapped out when it's priority count reaches 0. KEEP and GIVE should only be used when necessary to ensure a Task is not swapped out. See: KEEP & Section 19

GO> See: Section 19.

HERE --- addr 79

The address of the next available dictionary location.

HEX --- 79

Sets the numeric input-output conversion base to sixteen.

HIMEM --- addr

Returns the address addr of the top of memory used by the system. Initially set to 3000 Hex. This is tested by MODE, so that setting a mode does not overwrite the dictionary. See: LOMEM

HLD --- addr U

The address of a user variable containing the address of the latest character of text during numeric output conversion.

HOLD char --- 79

char is inserted into a pictured numeric output string. Typically used between <# and #>.

I --- w C,79

w is a copy of the loop index. May only be used in the form

```
DO ... I ... LOOP
```

or

```
DO ... I ... +LOOP
```

ID --- addr U

A user variable which contains the name field address of the task name.

ID. addr ---

Prints a definition's name from its name field address.

example : ' LIST >NAME ID. <return> LIST

IF flag --- C,I,79
--- sys (compiling)

Used in the form :

flag IF ... ELSE ... THEN
or
flag IF ... THEN

If flag is true, the words following IF are executed and the words following ELSE until just after THEN are skipped. The ELSE part is optional.

If flag is false, words from IF through ELSE, or from IF through THEN (when no ELSE is used), are skipped. sys is balanced with its corresponding ELSE or THEN.

IMMEDIATE --- 79

Marks the most recently created dictionary entry as a word which will be executed when encountered during compilation rather than compiled.

IN d --- M

Used to schedule a task to execute sometime in the future. d is a 32-bit value representing the number of clock ticks (1 tick = 10mS) until the task executes. See: Section 19.

INDEX u1 u2 --- M

Print the first line of each screen over the range {u1...u2}. This displays the first line of each screen of source text, which conventionally contains a title.

INPUT> --- n

Fetch byte from BBC operating system input stream.

INSTALL See: Section 19.

INTERPRET --- M,83

Reads words from the input stream and interprets them until input stream is exhausted. This is in 5 parts :

1) Parse the input stream to find the next word. If length of word = 0, then finish.

2) Check if word is defined. If so interpret it and go to stage 1.

3) Is word a number ? If so convert it and go to stage 1.

4) Does word begin with "*" (i.e. an O.S. call) if so pass it to the operating system and go to stage 1.

5) Error - word not defined and not a number.

See: STREAM PARSE

INTERPRETER See: Section 19. M

IS See: Section 19. M

J --- w C,79

w is a copy of the index of the next outer loop. May only be used within a nested DO-LOOP or DO-+LOOP in the form, for example :

DO ... DO ... J ... LOOP ... +LOOP

KEEP --- M

Stops a Task being swapped out unless it executes a WAIT PAUSE SLEEP etc. command. See: GIVE WAIT PAUSE SLEEP and Section 19.

KEY --- 16b M,83

The least-significant 7 bits of 16b is the next ASCII character received. All valid ASCII characters can be received. Control characters are not processed by the system for any editing purpose. Characters received by KEY will not be displayed. The word KEY is a user vector which normally contains the word INPUT> or KYBD>. See: INPUT> and KYBD>.

KEY? --- f

Returns a true flag if a key has been pressed otherwise false. For background tasks this is vectored to zero so that background tasks effectively do not "see" the keyboard.

KILL See: Section 19.

KYBD> --- n

Fetch byte from keyboard buffer.

LEAVE --- C,I,83
--- (compiling)

Transfers execution to just beyond the next LOOP or +LOOP. The loop is terminated and loop control parameters are discarded. May only be used in the form :

```
DO ... LEAVE ... LOOP
```
or
```
DO ... LEAVE ... +LOOP
```

LEAVE may appear within other control structures which are nested within a DO-LOOP.

LINE +n --- addr M

addr is the address of the beginning of line +n for the screen whose number is contained in SCR. The range of +n is {0...15}.

LINK> addr1 --- addr2 "from-link"

addr2 is the compilation address corresponding to the link field address addr1.

LIST u --- M,79

The contents of screen u are displayed. SCR is set to u. See BLOCK.

LIT --- n C

Within a colon-definition, LIT is automatically compiled before each 16-bit literal number encountered in the input stream. Later execution of LIT causes the contents of the next dictionary address to be pushed onto the stack.

LITERAL --- 16b C,I,79
16b --- (compiling)

Typically used in the form :
[16b] LITERAL

Compiles a system dependant operation so that when later executed, 16b will be left on the stack.

LL --- addr U

A user variable containing the line length in characters of the current display screen. The default value is 80. It is used by some words (e.g. VLIST) to format output to the screen or printer.

Try 20 LL ! VLIST <return>

LOAD u --- M,79

The contents of >IN and BLK, which locate the current input stream, are saved. The input stream is then redirected to the beginning of screen u by setting >IN to zero and BLK to u. The screen is then interpreted. If interpretation from screen u is not terminated explicitly it will be terminated when the input stream is exhausted and then the contents of >IN and BLK will be restored. An error condition exists if u is zero. See : >IN , BLK , BLOCK.

LOCAL See: Section 19.

LOMEM --- addr

Returns the address addr of the top of memory available for the dictionary, initially set to 7BFF Hex. This is tested by ALLOT so that the dictionary cannot overwrite anything above LOMEM. LOMEM and HIMEM are currently set by default to values which have no effect on the system. They can be changed by the user to protect parts of memory if required. See: HIMEM

LOOP --- C,I,83
sys --- (compiling)

Increments the DO-LOOP index by one. If the new index was incremented across the boundary between limit-1 and the limit the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding DO. sys is balanced with its corresponding DO.

LSB n --- n1

Returns the least significant byte of n.

LWORD --- addr U

A user variable containing the length of the last word found in the input stream. This is used with the contents of AWORD by the error words (REPORT) to indicate which word was being interpreted when an error occurred.

M* n1 n2 --- d

A mixed magnitude math operator which leaves the double number signed product to two signed numbers.

M/ d n1 --- n2 n3

A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

M/MOD ud1 ud2 --- u3 ud4

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single precision divisor u2.

MAKE ---

This word is used to define the action of words created using VECTOR. See: VECTOR and Section 17.

MAX n1 n2 --- n3 79 "max"

n3 is the greater of n1 and n2 according to the operation of > .

MEM? See: Section 19.

MEMORY --- addr

Returns address addr of the Memory Semaphore used for assigning pages of memory to tasks. See: Section 19.

MESSAGE n ---

If no text is given when the system finds an error and executes a BRK op-code, then REPORT executes this word with the error number n on the stack. At present this prints out the error message Msg..n See: Section 17.

MIN n1 n2 --- n3 79 "min"

n3 is the lesser of n1 and n2 according to the operation of < .

MOD n1 n2 --- n3 83

n3 is the remainder after dividing n1 by the divisor n2. n3 has the same sign as n2 or is zero. An error condition results if the divisor is zero or if the quotient falls outside the range {-32,768...32,767}, but no action is taken.

MODE +n ---

This word is used to set up the BBC´s screen mode.

MOVE addr1 addr2 u ---

The u bytes at address addr1 are moved to address addr2. The data bytes are moved such that the u bytes remaining at address addr2 are the same data bytes as was originally at address addr1. If u is zero nothing is moved.

MSB n --- n1

Returns the most significant byte of n.

N>LINK addr1 --- addr2 "name-to-link"

addr2 is the link field address corresponding to the name field address addr1.

NAME> addr1 --- addr2 "from-name"

addr2 is the compilation address corresponding to the name field address addr1.

NEGATE n1 --- n2 79

n2 is the two's complement of n1, i.e, the difference of zero less n1

NOT 16b1 --- 16b2 83

16b2 is the one's complement of 16b1.

NUM? addr L --- addr L f

Used by INTERPRET to check if the string given by address addr, length L is a valid number. Returns a false flag if word was a number, otherwise a true flag.

OFF ---

This word revectors the O.S timer interrupts. OFF reverses the effect of the word ON. See: ON and Section 5.6

ON ---

This word vectors the O.S timer interrupt to the Multi-FORTH clock routine, thereby effectively switching the timer on. The word OFF reverses this process. ON should never be used when the system is already ON or the system will lock-up. See: OFF and Section 5.6

ONLY --- ONLY

Select just the ONLY vocabulary as both the transient vocabulary and the resident vocabulary in the search order.

OPEN n1 --- n

Opens file using n1 to determine whether reading, writing or updating. Returns file handle n.

OPENIN --- n

Open file for input. Returns file handle n.

OPENOUT --- n

Open file for writing. Returns file handle n.

OPENUP --- n

Open file for update. Returns file handle n.

OR 16b1 16b2 --- 16b3 79

16b3 is the bit-by-bit inclusive-or of 16b1 with 16b2.

ORDER --- ONLY

Display the vocabulary names forming the search order in their present search order sequence. Then show the vocabulary into which new definitions will be placed.

OS? addr L --- addr L f

Used in INTERPRET to check if the string at address addr, length L starts with a "*", in which case it is an O.S call. f = false if the string is an O.S call.

OSBYTE YX n --- YX SA

BBC OSBYTE call number n, returns machine code registers in ASYX.

OSCALL YX SA addr --- YX SA

General BBC OS call at address addr, returns machine code registers in ASYX. Sets the carry flag according to bit 8 of the word on top of the stack.

OSCLI addr ---

Execute BBC command at address addr.

OSWORD addr n --- YX SA

BBC OSWORD call number n, parameter address is addr, returns machine code registers in ASYX.

OSWRCH n ---

Sends the LSB of n to the output stream. It returns nothing.

OUT --- addr U

The address of a user variable containing a count of the number of characters sent to the output stream.

OVER 16b1 16b2 --- 16b1 16b2 16b3 79

16b3 is a copy of 16b1

PAD --- addr M,83

The lower address of a scratch area used to hold data for intermediate processing. The address or contents of PAD may change and the data lost if the address of the next available dictionary location is changed. The minimum capacity of PAD is 84 characters.

PAGE --- M

Clear the terminal screen or perform a form-feed action suitable to the output device currently active. This word is a user vector which normally contains the word CR. See: CR.

PAGE1 See: Section 19.

PARSE del --- addr L

Searches the input stream for the delimiter character "del" and returns the address addr and length L of the string up to the delimiter. (>IN is adjusted accordingly and LWORD and AWORD are set). This is a more flexible method than using WORD as no text is moved and the addr,L format is better than leaving the address of a counted string.

PAUSE ---

This forces the current task to be swapped out, this means that the Task is put on the READY queue and another Task is swapped in. See: Section 19.

PICK +n --- 16b 83

16b is a copy of the +nth stack value, not counting +n itself. {0...the number of elements on stack - 1)
0 PICK is equal to DUP, 1 PICK to OVER.

PLOT nl n2 n3 ---

See: Section 5.5

PRINTER ---

This word redirects the output text stream by making EMIT send to the printer buffer via >PRTR and sets PAGE to "form-feed".

PROMPT --- addr V

This word is part of the QUIT loop and is a user vector which redirects the main task to CR and other tasks to TPROMPT by default which prints out the task name and address. See: QUIT, CR, TPROMPT.

PTR! d n ---

Set the sequential pointer of the file with file handle n to the value given by the double number d.

QUEUE See: Section 19. M

QUERY --- M,83

Characters are received and transferred into the memory area addressed by TIB. The transfer terminates when either a "return" is received or the number of characters transferred reaches the size of the area addressed by TIB. The values of >IN and BLK are set to zero and the value of #TIB is set to the value of SPAN. WORD may be used to accept text from this buffer. See : EXPECT.

QUIT --- 79

Clears the return stack, sets interpret state, accepts new input from the current input device, and begins text interpretation. No message is displayed.

R/W addr n ch f ---

This is the disc read-write linkage. It reads or writes a disc buffer at address addr. n = block number, ch = DFS channel number (file handle), f = R/W flag, being 1 for read and 0 for write. Depending on the value of ch, R/W uses DR/W to access the disc or CR/W to access the TAPE or ROM. For disc "USE" is neccessary to open a disc file as blocks. For Tape or ROM do not use "USE" as blocks are read or written to files "BLKnnn", where nnn is the number of the block. See: USE CR/W DR/W ^R/W and Section 5.7

R> --- 16b C,79 "r-from"

16b is removed from the return stack and transferred to the data stack.

R@ --- 16b C,79 "r-fetch"

16b is a copy of the top of the return stack.

READY --- addr M

Returns the address addr in zero page of the READY queue. See: Section 19.

RECORD See: Section 19 M

REPEAT --- C,I,79
sys --- (compiling)

Used in the form :

BEGIN ... flag WHILE ... REPEAT

At execution time, REPEAT continues execution to just after the corresponding BEGIN. sys is balanced with its corresponding WHILE. See : BEGIN

REPORT ---

Prints out the error message associated with the last error that occurred. This is automatically executed when an error occurs.

ROLL +n --- 83

The +nth stack value, not counting +n itself is first removed and then transferred to the top of the stack, moving the remaining values into the vacated position. {0...the number of elements on the stack -1}.

2 ROLL is equal to ROT
0 ROLL is a null operation

ROT 16b1 16b2 16b3 --- 16b2 16b3 16b1 79 "rote"

The top three stack entries are rotated, bringing the deepest to the top.

RP! ---

A computer dependant procedure to initialise the return stack pointer from the user variable RP0.

RP0 --- addr U

A user variable containing the initial value of the return stack pointer.

RUN See: Section 19. M

S->D n --- d

Sign extend a single number to form a double number.

SAVE-BUFFERS --- M,79

The contents of all block buffers marked as UPDATEd are written to their corresponding mass storage blocks. All buffers are marked as no longer being modified, but may remain assigned.

SCR --- addr U,79 "s-c-r"

The address of a user variable containing the number of the screen most recently LISTed.

SCREEN ---

This word redirects the output text stream to the computer screen via >OUTPUT.

SEARCH addr1 L addr2 --- addr3 n

Searches for the string defined by address addr1, length L in the vocabulary pointed to by address addr2. Returns parameters as for FIND. See: FIND

SEMAPHORE --- M

A defining word executed in the form :

n SEMAPHORE <name>

A dictionary entry for <name> is created and eight bytes are put aside in the dictionary. See: SIGNAL WAIT and Sections 19 and 21.

SHOW n1 n2 ---

Prints all TRIADs necessary to cover the range of blocks from n1 to n2-1.

SIGN n --- 83

If n is negative, an ASCII "-" (minus sign) is appended to the pictured numeric output string. Typically used between <# and #>.

SIGNAL addr --- M

This word is used for syncronising Tasks. It takes the semaphore address addr, and increments the semaphore count. If the count was 0 then the task at the top of the waiting list is restarted. See: Section 19

SLEEP See: Section 19.

SMUDGE ---

Used during word definition to toggle the "smudge bit" in a definition's name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SOUND See: Section 5.3

SP! ---

A computer dependant procedure to initialise the parameter stack pointer from SP0.

SP0 --- addr U

A user variable containing the initial value of the parameter (or data) stack pointer.

SP@ --- addr 79 "s-p-fetch"

addr is the address of the top of the stack just before SP@ was executed.

SPACE --- M,79

Displays an ASCII space.

SPACES +n --- M,79

Displays +n ASCII spaces. Nothing is displayed if +n is zero.

SPAN --- addr U,83

The address of a user variable containing the count of characters actually received and stored by the last execution of EXPECT.

STATE --- addr U,79

The address of a user variable containing the compilation state. A non-zero content indicates compilation is occurring, but the value itself is system dependant. A Standard Program may not modify this variable.

STATUS --- addr

This is the address of a user vector which is executed at the end of INTERPRETation of a line. It may be revectored to print out other system parameters, or whatever. See: Section 17.

STOP See: Section 19. M

STREAM --- addr L

Uses BLK and >IN to return the address addr and length L of the input stream. If L = 0 input stream is exhausted.

SWAP 16b1 16b2 --- 16b2 16b1 79

The top two stack entries are exchanged.

TASK See: Section 19. M

TAB Col Line ---

This word will position the current cursor to column and line.

TEXT char --- M

Accept characters from the input stream, as for WORD, into PAD, blank-filling the remainder of PAD to 84 characters.

THAW addr --- M

Includes buffer at address addr into the free buffer list. See: Section 19.

THEN --- C,I,79
sys --- (compiling)

Used in the form :

```
flag IF ... ELSE ... THEN
```
or
```
flag IF ... THEN
```

THEN is the point where execution continues after ELSE, or IF when no ELSE is present. sys is balanced with its corresponding IF or ELSE.

TIB --- addr 83 "t-i-b"

The address of the text input buffer. This buffer is used to hold characters when the input stream is coming from the current input device. The minimum capacity of TIB is 80 characters.

TIMER --- addr M

Returns the address addr in zero page of the TIMER queue. See: Section 19.

TOGGLE addr b ---

Complement the contents of addr by the bit pattern b.

TOP --- addr U

A user variable containing the address of the latest disc buffer accessed.

TP --- addr

Returns the address addr in zero page which contains the address of the Task Record (Task Pointer) for the current Task. Value returned is 24 Hex. See: Section 19

TPROMPT See: Section 19.

TRIAD n ---

Prints the page (three blocks) containing Block n with the number of each block printed above it. The top block on the page will have a block number evenly divisible by three.

TRUE --- n

Returns a logical "true" . n = -1 (Non-zero).

TSK See: Section 19.

TSW --- addr U

A user variable containing the address of the queue containing the present task.

TYPE addr +n --- M,79

+n characters are displayed from memory beginning with the character at addr and continuing through consecutive addresses. Nothing is displayed if +n is zero.

U. u --- M,79 "u-dot"

u is displayed as an unsigned number in a free-field format.

U.R u +n --- M,83 "u-dot-r"

u is converted using the value of BASE and then displayed as an unsigned number right aligned in a field +n characters wide. If the number of characters required to display u is greater than +n, an error condition exists.

U< u1 u2 --- flag 83 "u-less-than"

flag is true if u1 is less than u2

UM* u1 u2 --- ud 83 "u-m-times"

ud is the unsigned product of u1 times u2. All values and arithmetic are unsigned.

UM/MOD ud u1 --- u2 u3 83 "u-m-divide-mod"

u2 is the remainder and u3 is the floor of the quotient after dividing ud by the divisor u1. All values and arithmetic are unsigned. An error condition results if the divisor is zero or if the quotient lies outside the range {0..65,535}, but no action is taken.

UNTIL flag --- C,I,79
sys --- (compiling)

Used in the form :

BEGIN ... flag UNTIL

Marks the end of a BEGIN-UNTIL loop which will terminate based on flag. If flag is true, the loop is terminated. If flag is false, execution continues to just after the corresponding BEGIN. sys is balanced with its corresponding BEGIN.

UNQUEUE See: Section 19.

UP --- addr U

A user variable containing the User Pointer, the address of the user variables.

UPDATE --- 79

The currently valid block buffer is marked as modified. Blocks marked as modified will subsequently be automatically transferred to mass storage should its memory buffer be needed for storage of a different block or upon execution of FLUSH or SAVE-BUFFERS.

USE ---

Open file for use as disc screens. I.e., USE NAME to use the file NAME as a block file for LISTing and LOADing etc. See: Section 5.7

USER +n --- M

A defining word executed in the form :

+n USER <name>

which creates a user variable <name>. +n is the offset within the user area where the value for <name> is stored. Execution of <name> leaves its absolute user area storage address.

VARIABLE --- M,79

A defining word executed in the form :

VARIABLE <name>

A dictionary entry for <name> is created and two bytes are ALLOTted in its parameter field. This parameter field is to be used for the contents of the variable. The application is responsible for initialising the contents of the variable which it creates. When <name> is later executed, the address of its parameter field is placed on the stack.

VECTOR ---

A defining word executed in the form:

n VECTOR <name>

A dictionary entry for <name> is created and two bytes are set aside in the dictionary. This word creates user vectors which can be altered at a later date. See: MAKE and Section 17.

VOC? addr1 --- addr2

Given the name field address addr1 of a word in the dictionary this word follows the links down the dictionary until it reaches the vocabulary definition and returns the name field address addr2 of the vocabulary name. Used by ORDER.

VOCABULARY --- M,83

A defining word executed in the form :

VOCABULARY <name>

A dictionary entry for <name> is created which specifies a new ordered list of word definitions. Subsequent execution of <name> replaces the first vocabulary in the search order with <name>. When <name> becomes the compilation vocabulary new definitions will be appended to <name>'s list. See : DEFINITIONS.

VLIST ---

List the word names of the CONTEXT vocabulary starting with the most recently defined.

VTABLE --- addr

Returns the address of the Vocabulary Table for the Task.

WAIT addr ---

This word takes the semaphore address addr and looks at the semaphore count. If the count = 0 the task is added to the waiting list of the semaphore and stops. If the count is equal to or greater than 1, the semaphore count is decremented and the task continues. See: SEMAPHORE SIGNAL and Section 19.

WAKE See: Section 19.

WHERE n blk ---

When an error occurs loading a block the position in the input stream, n, where the error occurred and the block number blk, being interpreted are left on the stack. WHERE takes these values and types out the line in the block where the error occurred and marks the position of the error with a ^ character (carat).

WHILE flag --- C,I,79
sys1 --- sys2 (compiling)

Used in the form :

BEGIN ... flag WHILE ... REPEAT

Selects conditional execution based on flag. When flag is true, execution continues to just after the WHILE through to the REPEAT which then continues execution back to just after the BEGIN. When flag is false, execution continues to just after the REPEAT, exiting the control structure. sys1 is balanced with its corresponding BEGIN. sys2 is balanced with its corresponding REPEAT. See : BEGIN

WIDTH --- addr

A user variable containing the maximum number of characters saved in the compilation of a definition's name. It must be between 1 and 31, with a default value of 31. The name character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.

WORD char --- addr M,83

Generates a counted string by non-destructively accepting characters from the input stream until the delimiting character char is encountered or the input stream is exhausted. Leading delimiters are ignored. The entire character string is stored in memory beginning at addr as a sequence of bytes. The string is followed by a blank which is not included in the count. The first byte of the string is the number of characters {0...255}. If the string is longer than 255 characters, the count is un-specified. If the input stream is already exhausted as WORD is called, then a zero length character string will result.

If the delimiter is not found the value of >IN is the size of the input stream. If the delimiter is found >IN is adjusted to indicate the offset to the character following the delimiter. #TIB is unmodified.

The counted string returned by WORD may reside in the "free" dictionary area at HERE or above. Note that the text interpreter may also use this area.

WORDS --- M

List the word names in the first vocabulary of the currently active search order.

XOR 16b1 16b2 --- 16b3 79 "x-or"

16b3 is the bit-by-bit exclusive-or of 16b1 with 16b2.

[--- I,79 "left-bracket"
--- (compiling)

Sets interpret state. The text from the input stream is subsequently interpreted. For typical usage see LITERAL. See :]

[´] --- addr C,I,M,83 "bracket-tick"
--- (compiling)

Used in the form :

[´] <name>

Compiles the compilation address addr of <name> as a literal. When the colon definition is later executed addr is left on the stack. An error condition exists if <name> is not found in the currently active search order. See : LITERAL

[COMPILE] --- C,I,M,79 "bracket-compile"

Used in the form :

[COMPILE] <name>

Forces compilation of the following word <name>. This allows compilation of an immediate word when it would otherwise have been executed.

\ ---

Ignore the rest of the line on which this word occurs, until the end-of-line character or CR occurs. Used to put comments on a line.

] --- 79 "right-bracket"

Sets compilation state. The text from the input stream is subsequently compiled. For typical usage see LITERAL. See : [

^ABORT ---

This is the default word of the user vector ABORT. See: ABORT.

^BLOCK ---

This is the default word of the user vector BLOCK. See: BLOCK.

^CR ---

This is the default word of the user vector CR. See: CR

^CREATE ---

This is the default word of the user vector CREATE. See: CREATE.

^ERROR ---

This is the default word of the user vector ERROR. See: ERROR.

^{FIND} ---

This is the default word of the user vector {FIND}. See: {FIND}.

^INTERPRET ---

This is the default word of the user vector INTERPRET. See: INTERPRET.

^KEY? ---

This is the default word of the user vector KEY?. See: KEY?

^LOAD n ---

This is the default word of the user vector LOAD. See: LOAD.

^MESSAGE ---

This is the default word of the user vector MESSAGE. See: MESSAGE.

^PARSE ---

This is the default word of the user vector PARSE. See: PARSE.

^REPORT ---

This is the default word of the user vector REPORT. See: REPORT.

^R/W ---

This is the default word of the user vector R/W. See: R/W

^STREAM ---

This is the default word of the user vector STREAM. See: STREAM.

{ASSIGN} See: Section 19. M

{DEASSIGN} See: Section 19. M

{FIND} addr 1 --- addr2 n

Performs the same function as FIND except that string is defined by the address and length on the stack. This is more flexible than needing a counted string as for FIND. See: FIND

{INSTALL} See: Section 19. M

{LEAVE} See: Section 19. M

{LOOP} See: Section 19. M

{QUEUE} See: Section 19. M

{SLEEP} See: Section 19. M

{UNQUEUE} See: Section 19. M

{WAIT} See: Section 19. M

{WORDS} addr ---

Takes an address addr of the top of a linked list of dictionary entries, in other words, the name field address of the last word in a vocabulary and prints the names of all the words in the linked list. Used by WORDS

20.1 Memory and System Usage

a/ Machine Memory Usage

Byte #		
00-1F		- Zero page workspace - used when calling certain O.S. routines. Words using this space should make sure they are not swapped out before finishing with it. (Use KEEP and GIVE)
20		- Not Used.
21		- Contains machine code for an indirect jump.
22-23	WR	- Working register, contains CFA of word currently being executed.
24-25	TP	- Task Pointer, address of Task Record for current task.
26	PR	- Priority, contains negative number of clock ticks before current task is swapped out. If -ve this value is incremented on each clock interrupt until it becomes 0 and then next time through the inner interpreter the task will be swapped out. If 0 or +ve the value is unchanged by clock interrupts. By setting bit 7 = 0, the task will not be swapped out until a PAUSE is executed. (This is how KEEP works).
27	PR	- MSB of Priority - Ignored.
28-29		- Address of User Pointer.
2A-2B	SP	- Data Stack Pointer for current task.
2C-2D	RP	- Return Stack Pointer for current task.
2E-2F	IP	- Instruction (or Interpretive) Pointer for current task.

(Continued)

30-3F WA - Workspace Area. May be used within a code definition as temporary working registers.

40-47 - Multi-tasking registers - used by system for manipulating Task Records in queues. Interrupts must be disabled when using these registers.

40-41 RA Record Address.
42-43 QA Queue Address.
44-45 TA Tail Address.
46-47 TR Temporary Register.

48-49 testv - contains the address of code to be executed each time through next. The default condition is that it points to code to look at the ´ESCAPE´ key.

4A-4B setv - contains address of code to be executed whenever a new task is swapped in. This may execute a word, e.g. to set up a screen window for each task as it is swapped in.

4C-4D WRv - contains address of code to be executed to execute next Forth word. It may be possible to change this code to remap addresses etc.

4E-4F nextv - contains address of inner interpreter code (NEXT). May be altered to change inner interpreter, i.e. to skip Escape Testing or task swapping to speed up the system.

50-55 TIMER queue
- Task Records on this queue have their delay registers decremented on each clock interrupt until 0 when the Task Record is put onto the READY queue.

50-51 NFA - address of queue name.
52-53 HEAD - address of 1st Task Record. (FF= empty queue)

54-55 TAIL - address of last Task Record. (= HEAD if empty)

(Continued)

56-5B READY queue
- tasks whose records are in this queue are ready to run.
56-57 NFA - address of queue name.
58-59 HEAD - address of 1st task. (FF= empty queue)
5A-5B TAIL - address of last task (= HEAD if empty)

5C-7F Not used.

80-FF O.S. zero page variables.

100-1FF Machine Stack
- rarely used except for errors which store messages at 100 onwards.

200-3FF O.S. use.

400-4FF Main System Task Page 1 Workspace.

400-44F User Variables
450-47F User Vectors
480-4AB Vocabulary Table
4AC-4FF Data (or Parameter) Stack

500-5FF Main System Task Page 2 Workspace.

500-54F TIB - Terminal Input Buffer.
550-55F Output workspace for numbers. (1C bytes below PAD)
560-5B3 PAD - scratchpad, used by EDITOR.
5B4-5FF Return Stack.

600-7BF Task Records (28 Tasks)

600-60F is the Main System Task Record.

7C0-7CF Memory Page Records
Each 2 byte field contains the Task Record Address of the task which is using the memory page associated with this field. (0 indicates memory page is un-assigned).

(Continued)

7D0-7D3 File Channel Numbers
- these contain the DFS file handle of the files associated with F0,F1,F2,F3 respectively.

7D4-7D7 File Counts
- these contain a count of the number of tasks using F0,F1,F2,F3 respectively.

(A File can only be closed if no tasks are using it.)

7D8-7DB Task Record Variables.

7D8-7D9 No. of tasks available (Default = 28)
7DA-7DB Start address of Task Records (Default = 600)

7DC-7DF Memory Paging Variables.

7DC-7DD Page Records Start Address (Default = 7D0)
7DE Lowest Page available.
7DF Highest Page available + 1

7E0-7E7 MEMORY semaphore
- may be used to wait for memory pages to become available.

7E0-7E1 NFA - Address of semaphore name (MEMORY)
7E2-7E3 HEAD - Address of Task Record of 1st task waiting.
7E4-7E5 TAIL - Address of Task Record of last task waiting.
7E6-7E7 COUNT - No. of memory pages available.

7E8-7EF BUFFER semaphore
- used by tasks to wait for disc buffers to become available.

7E8-7E9 NFA - Address of semaphore name (BUFFER)
7EA-7EB HEAD - Address of Task Record of 1st task waiting.
7EC-7ED TAIL - Address of Task Record of last task waiting.
7EE-7EF COUNTER - No. of disc buffers available.

(Continued)

7F0-7F1 Buffer Link
- top of linked list of buffers available.

7F2-7F3 LOMEM - Tested by ALLOT

7F4-7F5 HIMEM - Tested by MODE

7F6-7F9 Not used
Contains code used to call clock interrupt routine. (JSR FF00, JMP IRQIV)

800-(PAGE-1) O.S. and DFS and ROM use.

PAGE-(PAGE+C17) System starts up by allocating space for 3 disc buffers. The first is frozen and used as four memory pages for running tasks. The other two are used as disc buffers.

(PAGE+C18)-DP Dictionary.

-7FFF Display memory.

b/ Task Records

Each Task is described by a Task Record of 16 bytes.

Byte #

0-1 Link Field - used when a task is in a queue, it holds the address of the next Task Record in the queue.

FF indicates end of queue.
1 indicates task dormant.
0 indicates Task Record not in use.

2-3 Priority - negative number of clock ticks before a task is swapped out.

Main Task priority = -10 (~ 100 mS)
Background Task priority = -2 (~ 20 mS)

4-5 User Area - Address of memory holding User Variables, etc.

6-7 Stack Pointer - Address of the top of the Parameter Stack.

8-9 Return Stack Pointer - Address of the top of the Return Stack.

10-11 Instruction Pointer - Address containing the CFA of the next word to execute.

12-15 DELAY - Used when a Task is in the TIMER queue. Contains the number of clock ticks until the Task starts.

LSB in byte 12
MSB in byte 15

Bytes 2 to 11 are copied to the zero page registers when a Task is running.

Bytes 6 to 11 are copied back from the zero page registers when a Task is swapped out.

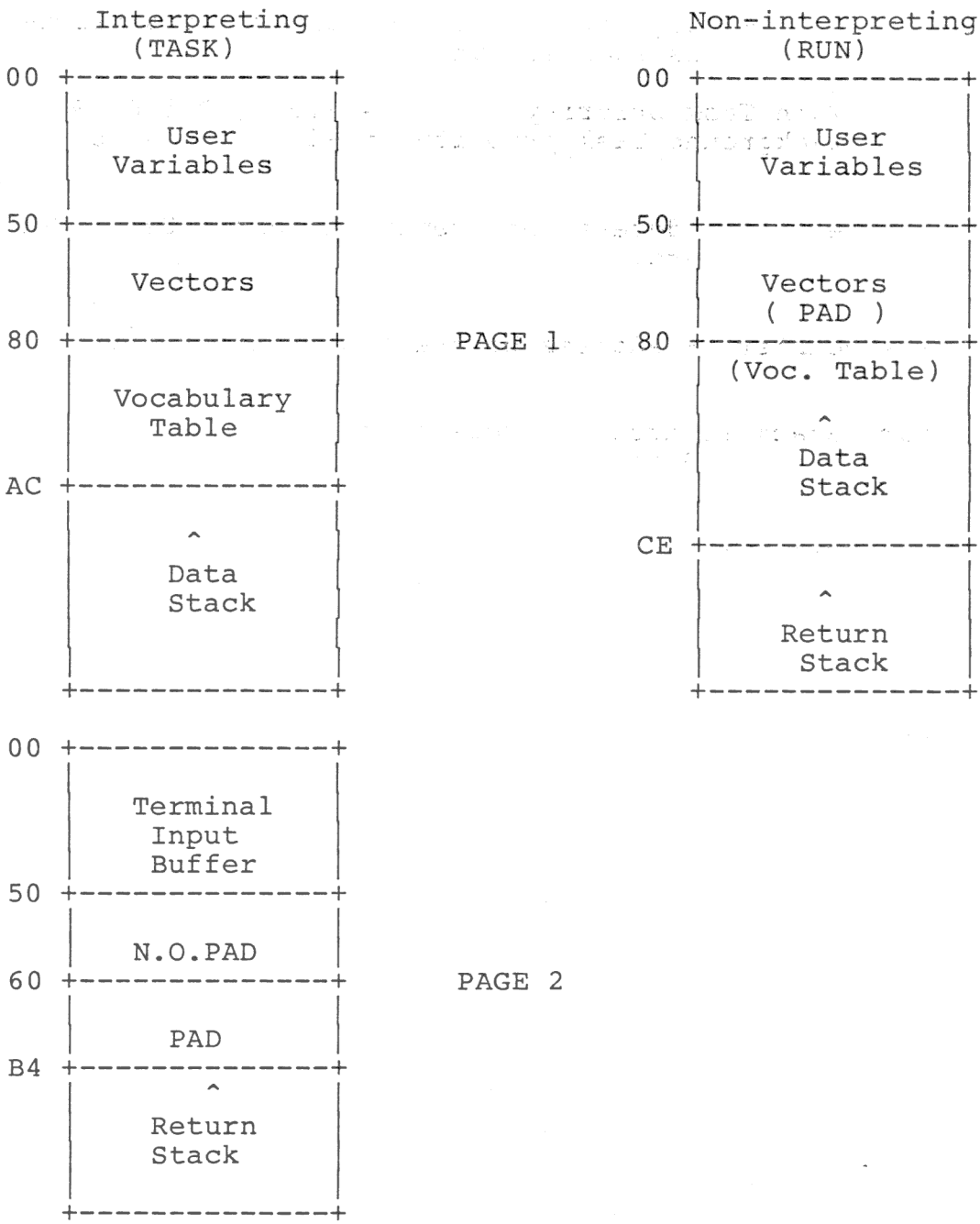
c/ Task Memory

Every Task, if it uses high-level Forth words needs memory to run. There are two types of high-level Task.

1/ Interpreting - these allow interpretation of the input stream but need 2 memory pages to run.

2/ Non-interpreting - these will run previously compiled words and need only 1 memory page.

The memory page usage of the two types of Task is as follows :



d/ User Variables

Byte (relative to UP)		[Default Value]
0-1	UP	- User pointer, contains address of User Variables
2-3	VP	- Vector pointer, contains address of User Vectors
4-5		- Address of PAD
6-7		- Address of Vocabulary Table
8-9	SP0	- Initial value of Data Stack Pointer
A-B	RP0	- Initial value of Return Stack Pointer
C-D	TIB	- Address of Terminal Input Buffer
E-F	DP	- Dictionary Pointer
10-11	FENCE	- Top of protected dictionary
12-13	ID	- NFA of Task name
14-15	TSW	- Task Status Word, contains address of queue containing task
16-17	FILE	- Address of byte containing DFS file handle for current block file
18-19	BASE	- Numeric I/O base [0A]
1A-1B	EOL	- Character used to denote end-of-line [ASCII 5E, ^]
1C-1D	BS	- Character used to denote backspace [ASCII 7F]

1E-1F	EOR	- Character used to denote end-of-record [ASCII 0D, CR]
20-21	WIDTH	- Maximum number of characters included in a word's Name Field [1F]
22-23	#TIB	- Maximum length of Terminal Input Buffer [50]
24-25	LL	- Output line length [50]
26-27		- Assembler address mode [2]
28-29	TOP	- Address of latest disc buffer accessed
2A-2B	BLK	- Block number of block being interpreted
2C-2D	SCR	- Block number of screen being edited
2E-2F	OUT	- Count of the number of characters sent to the output stream
30-31	DPL	- Position of decimal point in last number input
32-33	HLD	- Used in numeric output. Position of next character
34-35	SPAN	- Number of characters received in input stream
36-37	>IN	- Character offset into input stream
38-39	AWORD	- Address of last string parsed in input stream
3A-3B	LWORD	- Length of last string parsed in input stream
3C-3D	STATE	- If equal to zero ... executing, otherwise compiling
3E-3F	CSP	- Used during word definition to check stack pointer

e/ User Vectors

Byte (relative to VP)		Default Values [Interpreting / Non-interpreting]
0-1	ERROR	[^ERROR / ^ERROR]
2-3	REPORT	[^REPORT / ^REPORT]
4-5	MESSAGE	[^MESSAGE / ^MESSAGE]
6-7	ABORT	[DIE (^ABORT) / DIE]
8-9	PROMPT	[TPROMPT (CR) / TPROMPT]
A-B	KEY?	[0 (^KEY?) / 0]
C-D	KEY	[KYBD> (INPUT>) / KYBD>]
E-F	EMIT	[>OUTPUT / >OUTPUT]
10-11	CR	[^CR / ^CR]
12-13	PAGE	[CR / CR]
14-15	R/W	[^R/W / ^R/W]
16-17	BLOCK	[^BLOCK / ^BLOCK]
18-19	{FIND}	[^{FIND} / ^{FIND}]
1A-1B	INTERPRET	[^INTERPRET / ^INTERPRET]
1C-1D	STREAM	[^STREAM / ^STREAM]
1E-1F	PARSE	[^PARSE / ^PARSE]
20-21	CREATE	[^CREATE / ^CREATE]
22-23	LOAD	[^LOAD / ^LOAD]
24-25	STATUS	[.OK / .OK]

f/ Disc Buffers

These are held as 1032 bytes

Bytes

0 - 1023 Are the data.

1024 - 1025 Flag to indicate whether buffer is assigned (i.e. contains data) or not.

0 = unassigned.

When assigned this address contains the Task Record Address of the task that assigned it.

1026 - 1027 Link field.

Free buffers are kept in a linked list pointed to by location 07F0. This field points to the link field in the next buffer in the list.

0 = end of list.

1028 - 1029 File handle of Disc File in use for this buffer. The most significant bit is set if buffer has been UPDATEed.

1030 - 1031 Block number.

When a task uses a buffer, it is removed from the list, so that other tasks cannot get at it.

See: ?BUFF BLK? FETCH >DSK FREEZE THAW +BUFS BUFFERS for more information about the Disc Buffers.

Blocks are identified not only by the block number but also by the file to which it belongs. (Selected by USE).

FILE contains a pointer to the file handle of the disc file currently in use by the task.

Therefore FILE @ C@ . gives the file handle.

Up to four files may be used as block files. They are selected by using F0 F1 F2 or F3. These change FILE to point to the appropriate file handle. At the same time a count is kept of the number of tasks using a particular file so that a task cannot close a file which another task is using.

All tasks start up using F0.

FILE @ 4+ C@ . gives the count of the tasks using the file.

g/ SEMAPHORE Allocations.

Using the word SEMAPHORE to create semaphore words creates an eight byte dictionary entry. The use of the bytes is as follows :

Bytes

0 - 1 Point to the name of the semaphore. Used by DISPLAY to show which queue task is in.

2 - 3 Pointer to task record at the head of linked list of tasks waiting for the semaphore.

4 - 5 Pointer to the task record at the tail of linked list of tasks waiting for semaphore.

6 - 7 Semaphore Count.

Semaphores are used by two words : SIGNAL and WAIT

See: SIGNAL WAIT and Section 19

h/ Error Messages

Message #		
0	Not used	
1	Stack?	- The data stack was empty. Stack overflow is not tested
2	Input?	- Data from the O.S. input stream was marked as in error
3	No Room	- An attempt was made to allot space which was not available
4	Redefinition	- A word has been redefined (Warning only)
5	Arguments?	- Not enough items on the stack for the word to execute properly
6	Undefined	- The word was not found in the search stream
7	No String	- The input stream was exhausted when being parsed for a string
8	Compile Only	- This word may only be compiled
9	Execute Only	- This word may only be executed
10	Loading Only	- This word may only be executing whilst input is from disc blocks
11	Structure?	- Program control structure (IF...THEN, DO...LOOP etc) not completed or nested properly
12	Definition?	- Whilst defining a word the stack has changed size. This may indicate an error
13	In Use	- Block file cannot be closed as it is being used by another task
14	Block?	- An illegal block number was specified - usually trying to update or load block 0
15	File?	- File was not found

16	Bad Mode	- Not enough room above HIMEM to allow mode
17	Escape	- "Escape" key has been pressed
18	Table Full	- Vocabulary Table has no more room
19	Abort"	- Error Message supplied by user
20	Vector?	- A value has not been assigned to a vector
21	No Block File	- Block file has not been opened for use
22	R/W Error	- DFS has reported a R/W error when using disc block
23	Line?	- An illegal line number was specified. (Only 0-15 is allowed)
24	Address Mode?	- An unknown assembler address mode was specified
25	Range?	- Assembler branch instruction cannot be used or offset is greater than 127
26	Protected	- Word is in protected part of dictionary
27	Not in Current	- Word was not found in the current Vocabulary
28	No Records	- No more Task Records are available
29	No Memory	- No more memory pages are available
30	FS?	- Tape or ROM filing system in use, but channel number specified

22.1 Stack Display Utility

As FORTH uses a stack to hold various parameters, it is very useful to be able to see what is on the stack. The following short routine does this and can be installed very easily.

```
: .S      DEPTH  IF   SP@ 1 - SP0 @ 2 - CR
                          ." Stack Base > "
                     DO I @ U. ( . ) -2 +LOOP
                  THEN  CR ;

       MAKE PROMPT .S      <return>
```

The "." in brackets can be substituted for U. if required. This program will always display in the main task's current number base.

Those with very early versions of Multi-FORTH 83 will need to execute SP! when first using this program and also after an error from the keyboard.

Error Messages

Message Number		
0	Not Found	- The word being searched for was not found in the current vocabulary
1	Stack?	- The data stack was empty. Stack overflow is not tested
2	Input?	- Data from the O.S. input stream was marked as in error
3	No Room	- An attempt was made to allot space which was not available
4	Redefinition	- A word has been redefined (Warning only)
5	Arguments?	- Not enough items on the stack for the word to execute properly
6	Undefined	- The word was not found in the search stream
7	No String	- The input stream was exhausted when being parsed for a string
8	Compile Only	- This word may only be compiled
9	Execute Only	- This word may only be executed
10	Loading Only	- This word may only be executing whilst input is from disc blocks
11	Structure?	- Program control structure (IF...THEN, DO...LOOP etc) not completed or nested properly
12	Definition?	- Whilst defining a word the stack has changed size. This may indicate an error
13	In Use	- Block file cannot be closed as it is being used by another task
14	Block?	- An illegal block number was specified - usually trying to update or load block 0
15	File?	- File was not found
16	Bad Mode	- Not enough room above HIMEM to allow mode
17	Escape	- "Escape" key has been pressed
18	Table Full	- Vocabulary Table has no more room
19	Abort"	- Error Message supplied by user
20	Vector?	- A value has not been assigned to a vector
21	No Block File	- Block file has not been opened for use
22	R/W Error	- DFS has reported a R/W error when using disc block
23	Line?	- An illegal line number was specified. (Only 0-15 is allowed)
24	Address Mode?	- An unknown assembler address mode was specified
25	Range?	- Assembler branch instruction cannot be used or offset is greater than 127
26	Protected	- Word is in protected part of dictionary
27	Not in Current	- Word was not found in the current Vocabulary
28	No Records	- No more Task Records are available
29	No Memory	- No more memory pages are available
30	FS?	- Tape or ROM filing system in use, but channel number specified