METACRAFTS

FOR
APPLE II

METACRAFTS FORTH
USER'S GUIDE
FOR
APPLE II COMPUTERS

COPYRIGHT NOTICE

144 Crewe Road, Shavington, Crewe CW2 5AJ, England

Printed by Johnsons of Nantwich Limited

ACKNOWLEDGEMENTS

We would like to acknowledge the work of the Forth Interest Group, P.O.Box 1105, San Carlos CA 94070., in particular the fig-FORTH model by William Ragsdale, and the FORTH-79 standard document by the standards team.

The transportable control structure primitives and the idea for OUTLINE are due to Kim Harris of Laxen & Harris Inc.

The macro assembler came from an article by William Ragsdale in FORTH Dimensions vol. III no.5.

The ARGUMENTS and RESULTS words are due to an article by Marc Perkel which appeared in FORTH Dimensions vol. III no.6.

CONTENTS

ACKNOWLEDGEMENTS

CONTENTS

INTRODUCTION 1

PART I INSTALLATION GUIDE

1 HARDWARE CONFIGURATION 3
2 LOADING & RUNNING FORTH 4
3 MAKING BACKUP COPIES OF THE SYSTEM DISC 6
4 PERIPHERAL CONFIGURATION 7
  4.1 Execution Vectors 7
  4.2 Input-Output Execution Vectors 7
  4.3 Operator Terminal 8
  4.4 Printer Output 9
  4.5 Mass Storage 11

PART II DEVELOPMENT TOOLS

1 COPY 15
  1.1 Activating COPY 15
  1.2 Running COPY 15
  1.3 Duplicating Blocks using Two Drives 16
  1.4 Duplicating Blocks using a Single Drive 17
2 DOCUMENT 18
  2.1 Activating DOCUMENT 18
  2.2 Printer Output 18
  2.3 Generating Source Listings 18
  2.4 Generating Source Screen Outlines 19
  2.5 Producing a Screen Index 20
  2.6 Application Generated Printer Output 20
3 DECODE AND DUMP 22
  3.1 Activating DECODE 22
  3.2 DECODE Output 22
  3.3 Activating DUMP 24
  3.4 DUMP Output 24
4 Screen Editor 25
  4.1 Activating the Editor 25
  4.2 Working with a 40 Column Display 25
  4.3 Command Format 26
  4.4 Cursor Moving Commands 26
  4.5 Changing Screens 27

  4.6 Editing Modes 27
  4.7 OVERWRITE Mode 28
  4.8 INSERT Mode 28
  4.9 DELETE Mode 29
  4.10 Search and Replace Feature 30
  4.11 Generating the Index Line 32
  4.12 Trial Load 32
  4.13 Termination and Recovery 34
5 DEBUG 36
  5.1 Activating DEBUG 36
  5.2 Loading the Application 36
  5.3 Starting the Test 37
  5.4 Performing the Test 37
    5.4.1 Single Word Stepping 38
    5.4.2 Data Validation 38
    5.4.3 EMIT and KEY 40
    5.4.5 Examining non-Stack Data 41
    5.4.6 Debugger Skip Function 41
  5.5 Errors 42
  5.6 Defining Words 42

PART III SYSTEM DESCRIPTION

1 WORD SETS 44
2 INTERNAL DETAILS 44.1
  2.1 Direct Threaded Code 44.1
  2.2 The Interpreter Pointer 45
  2.3 Stacks 46
  2.4 Dictionary Header 46
  2.5 Defining Words 48
  2.6 Memory Map 48
3 VOCABULARY MECHANISM 50
  3.1 Context stack and FIND 50
  3.2 Current Definitions Stack & CREATE 50
  3.3 Vocabulary Tree 51
  3.4 Compatibility with the 79-Standard 52
  3.5 Internal Organisation 52
4 EXECUTION VECTORS 54
5 OVERLAYS 55
  5.1 Creating Overlays 55
  5.2 Loading Overlays 56
  5.3 System Use of Overlays 57
  5.4 Building a Turnkey Disc 58
6 BUFFER AND HEAP MANAGEMENT 59
7 LANGUAGE CARD 60

8 LOCAL VARIABLES 61
9 6502 ASSEMBLER 62
  9.1 The Assembly Process 62
  9.2 Run-time and Assembly-time 62
  9.3 Security 62
  9.4 Op-codes 63
  9.5 Stack Addressing 64
  9.6 Return Stack 65
  9.7 FORTH Registers 66
  9.8 CPU Registers 67
  9.9 Setup and N 67
  9.10 Control Flow 68
  9.11 Looping 69
  9.11 Conditional Execution 69
  9.12 Nested Control Structures 70
  9.13 Returning Control 70
  9.14 Reset and BRK 71
10 STRINGS, GRAPHICS & GAME CONTROLLERS 72

PART IV GLOSSARY

1 DEFINITION OF TERMS 75
2 GLOSSARY NOTATION 84
3 STACK MANIPULATION 87
4 NUMBER COMPARISON 90
5 ARITHMETIC & LOGICAL 92
6 MEMORY 96
7 CONTROL STRUCTURES 99
8 TERMINAL INPUT-OUTPUT 107
9 NUMERIC CONVERSION 114
10 MASS STORAGE INPUT-OUTPUT 118
11 DEFINING WORDS 124
12 VOCABULARIES 130
13 COMPILER 133
14 MISCELLANEOUS 138
15 STRINGS 143
16 HEAP MANAGER 147
17 LOCAL VARIABLES 148
18 GRAPHICS AND GAME CONTROLLERS 149
19 ASSEMBLER 154

APPENDIX 1 ERROR MESSAGES 162

APPENDIX 2 ERROR REPORTS 165

APPENDIX 3 FURTHER READING 166

INTRODUCTION

Metacrafts FORTH is a high performance implementation of the FORTH language and operating system for use with Apple II/IIe computers. It is fully compatible with the 1979 definition of the language as documented by the FORTH standards team.

This manual is your guide to the installation and use of the system. It will not teach you how to write programs in FORTH. If you require a tutorial text then you should consult the reading list in the appendix, where you will find a comprehensive selection of titles suitable for novices.

The user guide is split into four parts. The first part covers all the information you need to install Metacrafts FORTH on your configuration. Part II tells you how to use the set of program development tools stored on the system disc. In Part III you will find a description of the system's internal structure and principle extensions. Finally, Part IV contains a glossary of the word sets supported by the system.

While every endeavour has been made to present the information contained in this manual in as clear a fashion as possible, there will no doubt be some ambiguities and errors of omission. Please contact us and tell us about these, or any other suggestions you feel might improve either the User's Guide or indeed the software itself.

PART I INSTALLATION GUIDE

You should read this part of the manual before attempting to use Metacrafts FORTH. In it you will find details of the hardware configuration required and instructions which tell you how to modify/enhance the operating system to deal with alternative configurations. You will also find instructions telling you how to load and run the system, and how to backup the system disc.

1. HARDWARE CONFIGURATION

To be able to use Metacrafts FORTH you will need the following equipment:

- An Apple II with Autostart ROM and 48K RAM, or an Apple IIe.
- At least one 16-sector Disk II disc drive.
- A black and white television receiver.

To obtain the best results, however, we strongly recommend the use of:

- At least two disc drives.
- A green phosphorous monitor to reduce eyestrain.
- An 80 Column Video Card.
- Colour display capability if colour graphics are to be used.
- A printer.

2. LOADING AND RUNNING FORTH

The floppy disc that you purchased together with this manual contains all the software that you require to be able to load the system. Simply insert the disc in the boot drive (see your DOS 3.3 Manual if you don't know what a 'boot drive' is), ensure that the display terminal is set up for displaying 40 columns (if you have a hard-switching 80 column card), and switch on the Apple II.

Approximately twelve seconds later you should hear a series of beeps from the speaker indicating that the load has been successful. The display screen at this point will be showing details of the system version, the disc serial number and the message "SYSTEM READY" followed by OK, signifying that FORTH is ready to accept input from the keyboard.

If, after making several attempts, you can't get the system to load as just described, try loading some other disc - for example, the DOS 3.3 disc you were given when you purchased the Disk II.

If that won't load either, then check all connections and the power supply. If you still have no success, contact your dealer. If, on the other hand, the problem seems to be confined to the FORTH disc, then contact your supplier, who will check the disc and replace it, free of charge, if it turns out to be faulty.

Once you've achieved SYSTEM READY you can start typing FORTH text directly on the keyboard. The system will make no attempt to execute the code that you type until you press the key marked "RETURN", which we refer to as <return> in the rest of the manual. The FORTH keyboard input routine allows you to type up to 80 characters before pressing <return>. If you try and exceed this limit, the Apple speaker will beep. You can cancel everything you've typed by pressing the CTRL and X keys together, instead of pressing <return>. Alternatively, by pressing the backspace key you can reposition the cursor at any of the characters you've typed, retype it and any following characters, and

reposition the cursor back at the end of the line using the forward space key (->) before pressing <return>.

When FORTH has finished executing the text you typed, it responds with the message "OK" and waits for more input. Actually, the word that outputs the OK response has been implemented as an execution vector (see Part III chapter 4) called .OK. This means that you can substitute your own version to output the stack contents for example, before the OK message.

(Note: the brackets [and], and the underscore character can be obtained by typing CTRL-V, shift -M and CTRL-O respectively.)

3. MAKING BACKUP COPIES OF THE SYSTEM DISC

The FORTH system disc is write-protected to prevent you from inadvertently destroying its contents. Nevertheless, it is good programming practice to make copies of all important discs, and your FORTH disc is no exception. For this reason we allow you to make copies of it for backup purposes, and no other. We respectfully ask you to help keep down the cost of high quality software by not abusing this privilege.

Before proceeding therefore, please read chapter 1 in part II of this guide which tells you how to make complete or partial copies of your floppy discs.

4. PERIPHERAL CONFIGURATION

Metacrafts FORTH is configured to use the Apple keyboard and 40 column display, a centronics printer and two Disk II drives. The purpose of this chapter is to tell you how alter the configuration, either to replace one or more of these devices, or attach additional ones. For example, you may want to send output to an 80 column display instead of the standard 40 column one, or perhaps you want to connect a graphics tablet. If so, you should read the rest of this chapter and also Part III chapter 5 which talks about the overlay system.

4.1 Execution Vectors

Because we have implemented the principle input-output words as execution vectors, it is really quite easy to modify the configuration once you know what you want to do. Execution vectors are discussed in detail in Part III chapter 4. Very briefly, they are like user variables in that they are used to store a value. The value of an execution vector is the compilation address of some word. Whereas the run-time action of a user variable causes the address of the user variable to be left on stack, the run-time action of an execution vector causes control to be passed to the compilation address assigned to the vector.

4.2 Input-Output Execution Vectors

All input-output activity ultimately passes through the one or other of the execution vectors below.

EMIT -- output a character.

KEY -- input a character.

PAGE -- continue output on a new page.

@CURSOR -- leave cursor position on stack.

!CURSOR -- set cursor position to value on stack.

CLREOL -- clear from cursor position to end of line.

CLREOP -- clear from cursor position to end of page.

R/W -- read/write a mass storage block.

FORMAT -- format a mass storage volume.

Each of these words is fully documented in the glossary in Part IV of this guide.

4.3 Operator Terminal

Terminal output is transmitted to the display screen via EMIT, keyboard input is accepted via KEY. EMIT is initialised to invoke (EMIT) which, in turn, calls the Apple monitor COUT routine. KEY is initialised to invoke (KEY) which calls the Apple monitor RDKEY routine.

COUT is itself a form of vector, because it passes control to the routine whose address is stored at CSW. Similarly, RDKEY passes control to the routine whose address is stored at KSW. During system initialisation, and following a reset, CSW and KSW are set up to point at monitor routines which handle 40 column output and Apple keyboard input.

It is possible to direct character input/output to an alternative device by storing the address of the device's driver code in CSW or KSW as appropriate. Usually the driver code will be the device controller firmware.

You can, of course, write your own character input/output routines and use IS to assign its entry point to EMIT/KEY.

In addition to EMIT and KEY, there are five other vectors which perform video display functions. @CURSOR and !CURSOR are used to manipulate the cursor and are initialised to call (@CURSOR) and (!CURSOR) respectively. PAGE is set to call (PAGE) and is used to clear the screen and position the cursor in the top left-hand corner. CLREOL and CLREOP, initialised to call (CLREOL) and (CLREOP) , are used to clear the

display from the cursor position to the end of line and end of page respectively. Each of these five functions is carried out by an Apple monitor routine.

Screens 56 and 140 on the system disc contain an example of words set up to drive a VIDEX VIDEOTERM compatible 80 column video card. If such a card exists in slot 3, it can be activated by invoking the word 80-COLS. Be warned, however, that if you press reset with such a card active, it will be deactivated by the Apple monitor. It should be reactivated by typing 80-COLS again.

4.4 Printer Output

All character output can be directed, via EMIT, to a printer by storing the address of the printer driver routine in CSW. If pre-processing of the character output stream is required, the compilation address of a FORTH pre-processor word should also be assigned to EMIT using IS.

Screens 106-110 on the system disc contain a pre-processor for output to a Centronics printer. This is used by the words in the DOCUMENT overlay described in Part II chapter 2. You can use this pre-processor in your own applications by making a copy of it on your application disc.

If your printer/paper is incompatible with the printer parameters in screen 106, then you should amend them using the editor, and remake the DOCUMENT overlay.

The preprocessor has the following capabilities:

Automatic Pagination

-- a form-feed is performed after L/P lines have been printed on a page.

Optional Page Headings and Numbering

-- turned on by the word TITLE" and off by the word NO-TITLE. The starting page number can be set by the word PAGE-NUMBER.

Linefeed Switch

-- controlled by the constant HAS-LF on screen 106. If HAS-LF is set to 1 the printer is assumed to output an automatic LF character after every CR character that is sent to it. If HAS-LF is set to 0 the pre-processor will send a LF character after every CR character that it receives.

Formfeed Switch

-- controlled by the constant HAS-FF on screen 106. If HAS-FF is set to 1 the printer is assumed to respond correctly if it is sent a form-feed character (ASCII 12 decimal). If HAS-FF is set to 0 the pre-processor will simulate the effect of a form-feed character by sending the appropriate number of LF characters to the printer. You can use PAGE to cause a form-feed if you use the printer pre-processor, because this reassigns PAGE to emit a form-feed character when the printer is selected by PRINTER-ON.

Page Width

-- controlled by the constant PW on screen 106. PRINTER-ON resets the value of C/T to the value of PW.

Sheet Size

-- controlled by the constant L/S on screen 106. This constant determines the physical size of a sheet of paper in terms of the maximum number of lines that will fit on it.

Page Size

-- controlled by the constant L/P on screen 106. This constant determines the maximum number of lines, inclusive of heading, that will be sent to the printer before a form-feed is automatically emitted by the pre-processor.

Printer Slot

-- controlled by the constant 'PRINTER on screen 106. this constant determines the number of the peripheral slot containing the printer interface card.

Setup String

-- controlled by the string SETUP on screen 106. This is the string which initialises the Centronics printer. It is initialised to Ctrl-I80N which switches off the display output and allows 80 character lines to be sent to the printer. The string is typed in hexadecimal, and the first number is the string length.

4.5 Mass Storage

Metacrafts FORTH is designed to drive up to 6 Disk II drives, each with a disc storage capacity of 140 blocks of 1024 bytes. The controllers for these drives must be inserted into contiguous peripheral slots, with the boot drive connected to the controller in the highest numbered slot. Drives are referenced by a number in the range 1-6, where 1 is the boot drive. The current drive for disc transfers is taken to be the value of the user variable DR. The number of drives connected to the system must be stored in the user variable #DRIVES. The value of #DRIVES is used by COPY, described in Part II chapter 1, to determine whether single drive disc-to-disc copying is necessary.

Disc transfers take place to and from the FORTH block buffers in 1024 byte units. Transfer requests are serviced by the execution vector R/W which is set to invoke the Disk II driver (R/W). If (R/W) detects a device/media error it calls ABORT" DISK II ERROR", which terminates execution of the current program.

Floppy disc formatting is carried out by the execution vector FORMAT which is set up to call the Disk II formatter (FORMAT), which formats the disc in the current drive (determined by DR).

For performance reasons Metacrafts FORTH does not use either the DOS 3.3 or UCSD Pascal sector interleaving sequence. This means that discs produced on these systems cannot be read by Metacrafts FORTH unless the sector interleaving table at location hexadecimal BFB8 is altered to correspond to the interleaving on the disc to be read.

It is possible to mix mass storage device types, but to do so it is necessary to define words to carry out the read, write and format actions required by FORTH. The rest of this section explains how to go about doing this.

If you look at the definition of R/W in the glossary in Part IV of this guide, you will see that it expects to find the internal address of the block buffer involved in the transfer on stack. The external address, which is the one returned by BLOCK and BUFFER, is obtained by adding 5 to the internal address. Given the internal address, the following word will return the block and drive numbers associated with the buffer:

```
: XFER-INFO ( addr --- block drive )
2+ DUP @ 7FFF AND ( block number)
SWAP 2+ C@        ( drive number)  ;
```

In other words, if you write your own R/W driver, you can obtain details of the transfer from the internal address as just described.

The only information required to be able to format a disc is the drive number, and this is contained in DR.

Once you've written driver words for your mass storage device you need to know how to link them into FORTH. Because you need at least one Disk II drive to be able to boot the system it is not sufficient to merely reassign R/W and FORMAT to point at your drivers because this would have the effect of isolating the Disk II. The solution is to write intermediate words that sort out which driver to call based on the drive number. The intermediate words are then assigned to R/W and FORMAT using IS.

For example, if you had a single Disk II drive as drive 1, and a 64K RAM card to simulate mass storage as drive 2, then the words defined below would be

sufficient to ensure that all activity is routed to the correct driver.

```
: <R/W>  ( addr opcode --- )
OVER 4 + C@           ( get drive number )
1 = IF (R/W) ELSE RAM-DISC THEN ;

: <FORMAT>
DR @ 1 = IF (FORMAT) ELSE FORMAT-RAM THEN ;

FIND <R/W> IS R/W

FIND <FORMAT> IS FORMAT
```

PART II DEVELOPMENT TOOLS

This part of the manual describes the purpose and operation of the software tools that have been supplied to help you develop FORTH applications more easily. Chief among these are the sophisticated screen editor and source level debugger.

1. COPY

COPY is used to copy mass storage blocks from one place to another, either across volumes, or within the same volume, using multiple drives, or a single drive.

1.1 Activating COPY

COPY is stored as a pre-compiled overlay on the system disc in block 30. Its source code is stored in blocks 94-98. To activate COPY simply type the word COPY and press <return>. If the overlay is already in store it will be entered. If it isn't the operating system will load it from the system disc which should be in drive 1. If some other disc is in drive 1 you will be requested to replace it with the system disc and press any key to start loading. When loading is complete you will be asked to reinsert your own disc if drive 1 was the current or only drive.

1.2 Running COPY

COPY asks the user for information about what is to be copied to where. The precise dialogue depends on whether or not you have a single drive system. COPY checks the value of #DRIVES, and if this has the value 1 it assumes you have a single drive system. If you have a single drive system you will need to set #DRIVES to 1 because Metacrafts FORTH is issued with #DRIVES set to 2. The following sequence of words will reset #DRIVES to the value 1:

```
1 #DRIVES !
```

You can avoid typing this every time you load FORTH by using the editor to update block 13, which is automatically loaded every time you boot the system (see Part III chapter 5.3).

If you happen to have a single Disk II drive and a single 8" drive, then you should also set #DRIVES to 1 if you want to duplicate either a Disk II disc or an 8" disc. If you want to transfer blocks from one type to the other, then you need to set #DRIVES to 2. From this you will probably have inferred that COPY will work with any mass storage type provided that you have set up R/W to handle the different types (see Part I chapter 4).

1.3 Duplicating Blocks Using Two Drives

Suppose you want to copy all or part of one disc onto another disc (throughout the rest of this chapter we use the term disc to mean any mass storage medium which your system supports). If the destination disc is blank, you must format it using FORMAT before you activate COPY. When you have done this, activate COPY and answer the questions as they appear. If you mistype a value you can use the backspace key to reposition and retype it. COPY expects you to press <return> at the end of each reply before it asks the next question.

The final question asks you either to press return when you are ready for copying to commence, or to press the escape key if you want to quit the COPY program. At this point you should insert the discs into the appropriate drives and then press <return>. If, for any reason, you decide not to go ahead with the copy, you should press the ESC key. This will get you out of the COPY program and return you to FORTH.

Copying will start as soon as you press <return> (COPY typically takes about one minute to duplicate the 140 blocks of a Disk II disc). When copying is complete a message to this effect will appear on the screen followed by a 'continue or quit' prompt. If you want to make further copies you should press <return> at this point, otherwise press the ESC key to be returned to FORTH.

Below is an example of a dialogue which would duplicate the disc in drive 1 on the disc in drive 2.

```
SOURCE DRIVE ? 1
FIRST BLOCK ? 1
LAST BLOCK ? 140
DESTINATION DRIVE ? 2
DESTINATION BLOCK ? 1
```

```
PRESS RETURN TO CONTINUE, ESC TO QUIT

COPY COMPLETE

PRESS RETURN TO CONTINUE, ESC TO QUIT
```

If you want to move a set of blocks from one part of a disc to another part of the same disc, simply set the destination drive to be the same as the source drive. COPY is intelligent enough to handle overlapped copying. For example, if you move blocks 1-10 to blocks 3-12 then, after copying, blocks 3-12 will contain the original contents of blocks 1-10 (which is what you would expect!).

1.4 Duplicating Blocks Using a Single Drive

Single drive copying can involve a certain amount of media swapping. COPY asks whether or not you want to be prompted to swap discs. If you respond in the negative to this prompt, copying will take place within the same disc. In other words, blocks will be moved from one part of the disc to another. If you respond positively to the prompt, COPY will ask you to swap source and destination discs as and when necessary. The number of disc swaps is a function of the number of blocks to be copied and the number of buffers available for COPY's use (usually about 27).

Below is a sample single disc COPY dialogue:

```
START BLOCK ? 10
LAST BLOCK ? 20
DESTINATION BLOCK ? 60
TYPE RETURN IF SWAPPING DISCS

PRESS RETURN TO CONTINUE, ESC TO QUIT
SWAP DISCS AND PRESS ANY KEY TO CONTINUE

COPY COMPLETE

PRESS RETURN TO CONTINE, ESC TO QUIT
```

2. DOCUMENT

DOCUMENT, as the name implies, helps with documentation of your applications. It can be used to produce paginated listings of source screens on your printer as well as summary and index listings either on the terminal or printer.

2.1 Activating DOCUMENT

DOCUMENT is stored as a pre-compiled overlay on the system disc in blocks 31-32. Its source code is stored in blocks 106-111 and 100-101. To activate DOCUMENT simply type the word DOCUMENT and press <return>. If the overlay isn't in store the operating system will load it from the system disc which should be in drive 1. If some other disc is in drive 1 you will be requested to replace it with the system disc and press any key to start loading. When loading is complete you will be asked to reinsert your own disc if drive 1 was the current or only drive. FORTH responds OK when DOCUMENT is ready for use.

2.2 Printer Output

If you have a Centronics compatible printer you can use the facilities provided by DOCUMENT to produce paginated printer output. If your printer is not Centronics compatible, or if your paper size is not 66 lines to the page, then please read Part I chapter 4.4 which explains how to adapt the system to deal with your printer.

2.3 Generating Source Listings

With DOCUMENT loaded you can use either PRINT to produce a printed listing of a sequence of screens, or PRINT1 to produce a listing of a single screen. You can also, if you wish, specify a title to be printed at the top of each page together with the page number. The word TITLE" can be used to specify the text to appear in the title and PAGE-NUMBER to specify the number of the first page (the default is 1). If TITLE" has been invoked, then titles will be printed on all output until the feature is turned off by NO-TITLE.

Refer to the glossary in Part IV of this guide to find out how to call these words.

2.4 Generating Source Screen Outlines

With DOCUMENT loaded you can use OUTLINE to produce a summary of a sequence of source screens, and 1OUTLINE to produce a summary of a single screen. A screen summary consists of those screen lines that have one of '(',':' or 'C' as first character. In other words, provided you adopt the convention of starting all comment, colon and code definitions in column one, you can use these words to produce listings summarising the contents of your code.

Below is an example of OUTLINE's output of part of the system disc.

```
SCREEN #133
===========
   ( GRAPHICS & GAMES                      03-JAN-83 KGL )
   CODE HLINE ( X Y --- ) ( DRAW LINE TO X Y )
   CODE DRAW  ( ADDR U --- ) ( DRAW SHAPE AT ADDR )
   ( ROTATED BY FACTOR U )
SCREEN #134
===========
   ( GRAPHICS & GAMES                      06-JAN-83 KGL )
   : C180 ( DEGREES --- COSINE*10000 COSINE 0-180 )
```

A printed listing of screen outlines can be generated by prefixing a sequence of OUTLINE/1OUTLINE calls with PRINTER-ON, and postfixing them with PRINTER-OFF. The above example, for instance, was produced by the following set of words:

```
PRINTER-ON 133 134 OUTLINE PRINTER-OFF
```

The printed output will be paginated with titles if these have been switched on.

If you press any key while OUTLINE is sending output to either a printer or the terminal, output will stop until you press another key. This feature is useful if you are trying to read output sent to the terminal.

Refer to the glossary in Part IV of this manual for details of how to call OUTLINE and 1OUTLINE.

2.5 Producing a Screen Index

Another useful feature of DOCUMENT is the word INDEX. This is used to produce a terminal or printer listing of the first lines of a given range of screens subject to the condition that the lines consist of comment. If a screen is found whose first line does not consist of comment, a question mark is output. If the index is sent to a printer it will be paginated with titles if these have been switched on.

Below is a sample listing of INDEX's output of part of the system disc:

```
32 ?
33( SYSTEM OVERLAY GENERATOR          23-FEB-83 KGL )
34( MISCLELLANEOUS GLOBALS            14-DEC-82 KGL )
35( VOCABULARY WORDS                  14-DEC-82 KGL )
```

As with OUTLINE, pressing any key temporarily stops and starts the output from INDEX.

2.6 Application Generated Printer Output

You can use the words PRINTER-ON and PRINTER-OFF in an application in order to direct the character output stream to the printer. The resultant output will be paginated automatically, although you can manually force a form-feed by means of PAGE. In addition, your applications can use TITLE", PAGE-NUMBER and NO-TITLE to further control the form of the printed output. Be warned, however, that the cursor control words @CURSOR and !CURSOR will have no effect on the printer.

The printer control words can be made available at compile time in one of two ways. First, you can simply insert a call to DOCUMENT at the start of your application, before any words have been added to the dictionary. Alternatively, you can compile the relevant source screens along with your application (see Part I chapter 4.4).

If you choose the first method and want to generate an overlay containing your application, don't forget to include the printer words in the overlay (either that, or remember to load DOCUMENT before loading your application).

3. DECODE AND DUMP

It is sometimes useful to be able to examine the code that the FORTH compiler has generated for a colon definition, particularly if you are developing new compiler words. DECODE will generate a symbolic terminal or printer listing of any colon definition. Similarly, during program testing, you might want to examine the contents of an area of store - the contents of a block buffer for instance. DUMP lets you display, on the terminal or printer, the contents of selected areas of store.

3.1 Activating DECODE

DECODE is loaded in source form from blocks 102-105. To activate it simply type the word DECODE followed by the name of the word you want to examine, and press <return>. If the overlay is already in store it will be entered. If it isn't the operating system will load it from the system disc which should be in drive 1. If some other disc is in drive 1 you will be requested to replace it with the system disc and press any key to start loading. When loading is complete you will be asked to reinsert your own disc if drive 1 was the current or only drive.

3.2 DECODE Output

If the word you want to examine is not a colon definition or simply not in a vocabulary on the context stack, you will be notified accordingly and returned to FORTH. Output from DECODE is sent to the terminal, and can be temporarily stopped and restarted by pressing any key on the keyboard. To obtain a paginated, titled printer listing of the output, load DOCUMENT before you start working on your application development, and use the printer words to reroute the output to the printer.

Below is a sample listing of a request to DECODE the word ".

```
DEFINITION OF " (IMMEDIATE)

3534  P   LIT 007F 127
3538  P   CLIT 22 34 "
353B  P:  ENCLOSE
353D  P:  ?NULL
353F  P   STATE
3541  P   @
3543  P   0BRANCH 0012 18
3547  P:  COMPILE 3516 13590
354B  P:  >HERE
354D  P   C@
354F  P   1+
3551  P:  ALLOT
3553  P   BRANCH 0004 4
3557  U:  >PAD
```

The first line of the output identifies the decoded word, in this case ", and indicates if it is an IMMEDIATE word. Each of the subsequent lines begins with the hexadecimal address of a cell in the dictionary which belongs to the definition being decoded. Each such cell contains the compilation address of the word whose name appears on the line containing the cell's address. The characters which follow the cell address consist of a P if the word is in the protected dictionary, or U if it is unprotected (a protected definition can't be forgotten by FORGET). This is followed by a : if the word is a colon definition. If any in-line data belonging to the word follows the cell address, then this is output after the word's name. In many cases the data is output in a variety of forms. The following list gives the form used for each of the words known to have in-line data:

Word name	In-line data format		
LIT	hex	decimal	
CLIT	hex	decimal	ASCII
BRANCH	hex	decimal	
0BRANCH	hex	decimal	
(LOOP)	hex	decimal	
(+LOOP)	hex	decimal	
(OF)	hex	decimal	
COMPILE	hex	decimal	
(.")	string		

(")	string
(ABORT")	string

DECODE is capable of decoding all of the colon definitions produced by the compiler words in the released system. If you develop a compiler word which generates in-line data, then you should modify the definition of DECODE on screen 105. If you don't, DECODE will behave unpredictably if you ask it to decode a definition containing the new word.

3.3 Activating DUMP

DUMP is loaded in source form from blocks 98-99 of the system disc. To activate DUMP simply type the address of the first byte to be dumped followed by the address of the last byte and then the word DUMP and press <return>. If the overlay is already in store it will be entered. If it isn't the operating system will load it from the system disc which should be in drive 1. If some other disc is in drive 1 you will be requested to replace it with the system disc and press any key to start loading. When loading is complete you will be asked to reinsert your own disc if drive 1 was the current or only drive.

3.4 DUMP Output

Each line consists of 16 bytes of store displayed first in hexadecimal, and then in character format. Non-printing characters are shown as an underscore character. The address of the first byte displayed on each line appears in hexadecimal at the beginning of the line followed by a / character.

If you use DUMP with a 40 column display, then 8 bytes will be output on each line instead of 16.

As with DECODE, the store dumper can be used in conjunction with DOCUMENT to send output to the printer.

4. Screen Editor

The editor lets you create and modify FORTH blocks containing source code and text data. The unit of display is the FORTH screen. Cursor movement and text manipulation is allowed anywhere within a screen. Movement of the display 'window' between screens and across mass storage devices is permitted.

4.1 Activating the Editor

The editor is stored as a pre-compiled overlay on the system disc in blocks 21-26. Its source code is stored in blocks 78-93. To activate the editor, type the number of the screen you want to look at (having first selected the appropriate mass storage device), followed by the word LOOK, and press <return>. If the overlay is already in store it will be entered. If it isn't the operating system will load it from the system disc which should be in drive 1. If some other disc is in drive 1 you will be requested to replace it with the system disc and press any key to start loading. When loading is complete you will be asked to reinsert your own disc if drive 1 was the current or only drive.

Eventually, the screen you specified will be displayed with the flashing cursor positioned over the first character of line 0. This is called the 'start' position.

If the screen you select for editing has never before been written to by the editor (a newly formatted disc for example), then the screen contents will consist solely of the character '@'. If, on the other hand, the block containing the screen has previously been used to store non-textual data, then the display will contain a totally random collection of characters. In either case the block must be prepared for editing by means of the delete mode 'S' command (see 4.9).

4.2 Working with a 40 Column Display

If you are working with the standard Apple 40 column display, then only the leftmost 37 characters of each of the 16 x 64 character lines will be displayed. However, the cursor control facilities described below enable you to scroll the screen to the left and to the right, thus affording you access to the complete FORTH screen.

We suggest that you use the leftmost 37 characters of each line for program code, and the remaining characters for comment. If you confine yourself to using this approach, you will find editing and program testing with the debugger is much simpler, particularly if you don't have a printer.

Of course, if you have an 80 column card, you will be able to see the whole screen without any problem.

4.3 Command Format

Editing actions are invoked by the user typing editor commands. Each command consists of a single character, usually a control character which is typed by simultaneously pressing the CTRL key and one other key. In the rest of this chapter control characters are shown typed in brackets. For example, (C) denotes CTRL-C and it is typed by pressing the CTRL and C keys simultaneously.

4.4 Cursor Moving Commands

There are nine cursor moving commands. They are:

Command	Action
->	move cursor one position to the right
<-	move cursor one position to the left
(Q)	move cursor vertically to the line above
(Z)	move cursor vertically to the line below
(S)	move cursor to the start position
(T)	move cursor 5 positions right (tab)
(B)	position on first non-blank character of the line
(E)	position after last non-blank character of the line
<return>	position on first non-blank character of the next line

If you have an Apple IIe you can use the vertical tab characters instead of (Q) and (Z).

The <return> command has additional mode dependant side-effects which are described later in this chapter.

Any attempt to move the cursor outside the 16 x 64 character screen display will be met by a beep from the speaker.

4.5 Changing Screen

The following commands let you move to another screen:

Command	Action
(P)	move to previous screen
(N)	move to next screen
(G)	go to a user specified screen

The first two of these commands should require no further explanation. When you type (G), however, the display will clear and the message

```
GO TO WHICH SCREEN?
```

will appear. If the screen you want to select is on the same disc, simply type its number and press <return>. If the screen is on another disc, and you have a single disc system, you should first insert the correct disc before carrying out the above instructions. If, however, you have two or more drives, you should insert the required disc in a free drive and postfix the new screen number with a fullstop followed by the drive number, and then press <return>. For example, suppose you are looking at screen 35 from the disc in drive 1 and you want to change to screen 60 from the disc in drive 2. To do this simply type 60.2 and press <return> in response to the above request.

4.6 Editing Modes

When the editor is first entered it is operating in OVERWRITE mode. You will see a message to this effect below line 15 of the screen on display. If you press the ESC key the editor switches to DELETE mode and the message below line 15 changes accordingly. Press the ESC key again, and you will find yourself in INSERT mode. To return to OVERWRITE mode simply press ESC one more time. In other words, repeatedly pressing the ESC key causes the editor to cycle through the three edit modes. The cursor and screen changing commands may be used in all three modes.

4.7 OVERWRITE mode

Overwrite mode is the normal mode for entering new program source into a screen. Simply type the required characters and they will appear on the display just as if you were using a normal typewriter. When you reach the end of a line the Apple will beep. To carry on on the next line, press <return> and carry on typing.

In common with normal typewriters the editor has a tab function which is activated by the (T) command. The tab positions are fixed and occur every five character positions. The other cursor moving commands can be used to position the cursor anywhere on the 16 screen lines.

Any characters typed in overwrite mode will replace the characters at the cursor position. One way of editing existing text is, therefore, to position the cursor on the first character to be changed and start typing replacement text.

4.8 INSERT mode

Often you will want to insert words between other words. You can do this by selecting INSERT mode, positioning the cursor on the character that is to follow the insertion, and typing the text to be inserted at that point. As you type you will find that the new characters appear at the cursor position and the original characters are displaced to the right to make room for them. As non-blank characters reach the end of the line they are lost.

The (T) and <return> commands have additional side effects when used in INSERT mode. The (T) command displaces all the text from the current cursor position to the right to the next tab position, leaving blanks in their place. The <return> command moves all the text between the cursor position and the end of the line down to the line below, justified to the far left of the screen. All the lines below the original line are displaced down a line to make room for the new line, and line 15 is lost.

If you position the cursor at the first character position of a line (other than line 15) and press <return>, the complete line will be moved to the line below leaving a blank line in its place.

4.9 DELETE mode

This mode is most often selected prior to inserting some replacement text. There are eight commands for use in delete mode. These are listed in the following table.

Command	Action
C	delete the character at the cursor position
W	delete all the characters from the current position up to, but excluding, the next blank.
L	delete the current line and move all following lines up to replace it. A blank line moves into line 15.
S	delete all the text in the current screen.
M	delete the current line, as in L, and push it onto the copy stack (see below).
D	push a duplicate copy of the current line onto the copy stack and move cursor to line below.
R	pop the line on top of the copy stack and place it on the line at the cursor position, displacing the line currently there, and all following lines, down one place.

The S command is normally used to prepare screens for editing. Before the screen is cleared you will be asked to confirm the request, the Apple will beep, and the message REALLY? will appear in place of the mode status message. If you really want to clear the entire screen, press <return>. If not, press any other key and the request will be cancelled.

The copy stack mentioned above is used to either move or duplicate lines of text within or between screens. The stack holds at most 16 lines of text. The push commands M and D can be interspersed with any other editor command (including R) apart from the (X) command described in 4.13.

The copy stack functions like any other stack: the last line pushed onto it by M or D will be the first line popped by R. The Apple speaker will beep if you attempt to overfill the stack or use R with an empty stack.

The <return> command also has side effects in DELETE mode. Pressing <return> causes the text between the cursor position and the end of the line to be deleted and the cursor to move to the first non-blank character on the following line.

4.10 Search and Replace Feature

The (F) command, applicable in any mode, turns on the search and replace feature with which you can get the editor to find, and optionally replace, a specified character string wherever it occurs in a given set of screens.

Typing (F) clears the screen and the prompt

```
FIND?
```

appears. You should respond by typing the character string to be found. You may type any sequence of characters up to a maximum of 63, excluding the character ^. Mistakes can be corrected by backspacing and retyping. Press <return> when you are ready.

Once you have specified the search string you will be requested to type a replacement string. This is the character string with which you want to replace instances of the search string. If you only want to find instances of the search string without making any alterations, you should press the ESC key followed by

the <return> key. Otherwise type the replacement string in the same way as you typed the search string. It too may be up to 63 characters long.

Finally, you will be asked to say where you want the search to terminate. The search always starts with the current screen, and you can choose which screen is to be the last one searched. You are allowed to choose a screen with a lower number than the current screen.

Press <return> to start the search.

Up to this point the screen will contain something like the following:

```
FIND? SOME SORT OF STRING
REPLACE WITH? ANOTHER SORT OF STRING
STOP AT SCREEN? 76
```

When an instance of the search string is found, the screen containing it is displayed with the cursor positioned on the first character of the instance. If you want to make a replacement, press <return> and you will see the instance disappear to be replaced by your replacement string.

If replacing the instance would cause non-blank characters to be lost at the end of the line, as might be the case if the replacement string is longer than the search string, then the speaker will beep and no replacement will be made.

If you choose not to replace the instance found, press any key other than <return> or ESC, and searching will continue to the next instance.

If you never specified a replacement string, then any key other than ESC causes searching to continue.

If you press the ESC key when an instance of the search string is found, the search terminates at that point.

When the terminating screen is reached the Apple beeps and the screen is displayed with the cursor at the start position.

4.11 Generating the Index Line

The index line is the first line of the screen, and is normally reserved for storing a comment. The DOCUMENT INDEX feature can be used to generate an index of a given set of screens consisting of the index lines.

The (I) command is intended to help you generate the index line quickly and easily.

The first time you use (I) after booting FORTH you will be asked to supply identification information to be inserted in the index line by this and every subsequent use of (I). The way this happens is that the mode status line disappears to be replaced by 13 dots with the cursor positioned over the first. You should now type up to 13 characters of identifying information. Mistakes can be corrected by using backspace and retyping. When you have typed the string, press <return>. The editor will write an index line consisting of the comment symbols and your identifier positioned right justified on the line. The cursor is left positioned after the open comment symbol so that you can type in screen specific information.

The next time you want to generate an index line just type (I) and the editor will generate one using your identifier and then wait for you to add the screen specific data.

We suggest that you adopt the FORTH practice of using the 13 character identifier to record the date and initials of the author of the screen. The format used on the system disc is typical in this respect.

4.12 Trial Load

Quite often, the first time you load a screen, FORTH will find something wrong with it. Usually it is something trivial such as a missing THEN, or an undefined word. To help you find and correct such mistakes quickly and easily the editor has its own load command.

Suppose you've just finished typing in a sequence of source screens, and you want to use the editor's loader to find out if they are OK. Invoke the loader with the (L) command and the screen will clear to be replaced by the prompt

```
LOAD WHICH SCREEN?
```

to which you should respond with number of the first screen to be loaded. If the screen is not on the current drive, you can append the drive number to the screen number separated from it by a fullstop (as with the (G) command described in 4.5). Mistakes can be corrected by backspacing and retyping the number.

Press <return> when you've typed the screen number and loading will begin. If no errors are found the editor will respond with a message and return you to FORTH. Up to this point the screen will look something like:

```
LOAD WHICH SCREEN? 35.2

LOADING COMPLETEok
```

If, on the other hand the compiler finds an error, then loading stops and an error message is output. A typical example of this is:

```
LOAD WHICH SCREEN? 35.2
THEN STRUCTURE?
PRESS ANY KEY TO CONTINUE
```

The first part of the message is a standard compiler error indicating the word that is wrong and the reason. When you obey the continue prompt the editor overlay is reloaded (which may involve swapping discs) and the screen containing the error is displayed with the cursor positioned just before the word which follows the incorrect word. You can now correct the error and try again. There is no need to worry about FORGETting anything already loaded during the previous attempt: the editor does that automatically.

4.13 Termination and Recovery

When you complete an editing session you can leave the editor by using the (C) command. You don't have to worry about saving your work on mass storage: the editor uses the standard FORTH block buffers and UPDATE's all modified blocks as editing proceeds. The (C) command calls SAVE-BUFFERS to tidy up at the end of the session.

If, while you are editing, you make a mistake involving more than a couple of characters, you can use (X) in any mode to cancel the most recent actions. The (X) command returns the screen to the state it was in at the time of the last screen change, mode change or <return> command, whichever occurred last.

You cannot use (X) to recover text replaced by the (F) command.

(X) clears the copy stack, leaves the mode unchanged and positions the cursor at the first non-blank character of the line it was on when (X) was invoked.

You can return the current screen to the state it was in at the time you selected it by means of the (R) command, which is available in all modes. (R) requests confirmation of your request by beeping the speaker and displaying the message REALLY? on the mode line. Respond by pressing <return> if you want to recover the screen, otherwise press some other key if you want to cancel the request.

Finally, you can abandon what you are doing completely by simultanously pressing CTRL-SHIFT-P. You will be asked to confirm the request in the same way as the (R) command above, and you should respond accordingly. If you press <return>, then the editor calls EMPTY-BUFFERS and returns you to FORTH.

Note: if you press RESET while under control of the editor, the prompt "PRESS ANY KEY TO CONTINUE" will appear on the 40 column display. You should obey the prompt to get back to FORTH. If you are connected to an 80 column card, press <return> and then 80-COLS followed by <return>.

5.0 DEBUG

The debugger is used to test selected word definitions at the source code rather than the object or compiled code level. You can maintain tight control of program execution at all times, and work with both 40 and 80 column displays. The current version of the debugger cannot be used in conjunction with the graphics facilities unless you are fortunate enough to have two display units -- one connected to the Apple video output socket, and the other to an 80 column display card.

5.1 Activating DEBUG

The debugger is stored as a pre-compiled overlay on the system disc in blocks 27-29. Its source code is stored in blocks 78-79 and 112-120. Before the application to be tested can be loaded, DEBUG must be loaded. To do this you should type DEBUG and press <return>. If the overlay isn't already in store the operating system will load it from the system disc which should be in drive 1. If some other disc is in drive 1 you will be requested to replace it with the system disc and press any key to start loading. When loading is complete you will be asked to reinsert your own disc if drive 1 was the current or only drive. FORTH returns with the OK response when DEBUG is ready.

5.2 Loading the Application

The debugger has its own loader also called LOAD. This loader differs from FORTH's LOAD in the way that it compiles code into the dictionary.

Often, not all the words belonging to an application require testing in a given session -- some of them may already have been tested in an earlier session. These words should be loaded before the words to be tested (called the test set) using FORTH's own loader. Once this has been done, add the debugger vocabulary to the context by typing DEBUGGER and pressing <return>, and then load the test set using the version of LOAD in the debugger vocabulary.

For example, suppose screens 35-40 contain words that have already been tested, and that the test set is contained in screens 41-45. The complete load sequence would be:

```
DEBUG
35 LOAD
DEBUGGER
41 LOAD
```

This example assumes that screen 35 loads 36-40, and that screen 41 loads 42-45 either by using LOAD or -->.

5.3 Starting the Test

Before you start the test you must ensure that the disc containing the application screens is in the currently selected drive (the test set should always be contained on a single disc).

The debugger vocabulary contains the word TEST. A call to TEST should be inserted into the sequence of words used to call the application, immediately preceding the name of the word which starts the application. For example, suppose the test set loaded in the example in 5.2 is started by the word MIXER. Suppose also that MIXER expects to find the address of some data, CEMENT say, on the stack. Then, instead of typing

```
CEMENT MIXER
```

as normal, you should type

```
CEMENT TEST MIXER
```

and press <return> when you are ready to start testing.

As soon as execution of MIXER reaches a word which belongs to the test set, the screen containing its definition is displayed with the cursor positioned at the last character of the first word in the definition.

5.4 Performing the Test

The programmer should be on the lookout for two things during a test session. First that the flow of control through the application is correct, which is another way of saying that the words in the application are obeyed in the intended sequence. The second thing to be checked is that the data belonging to the program are initialised and updated as and when expected. There are several ways of using the debugger to do check these things.

5.4.1 Single Word Stepping

You can use the debugger to step through the component words of a definition, one word at a time. For example, suppose the definition of MIXER is:

```
    : MIXER
         INGREDIENTS GET BEGIN MIX READY UNTIL POUR ;
```

Suppose also that MIXER belongs to the test set, so that the debugger will stop and display the screen containing the above definition when it is called as in 5.3. The cursor will be positioned at the S of INGREDIENTS.

Pressing <return> at this point will cause one of two things to happen. If the definition of INGREDIENTS had been loaded as part of the test set, the debugger will move the cursor to the first word of its definition, changing the screen displayed if necessary. If it was not part of the test set, the cursor will be moved to the T of GET.

By repeatedly pressing <return> in this way you can step through the definitions of the words in the test set, and thereby validate the flow of control through the application. Eventually you will end up at the semicolon terminating the definition of the last word called in the test set, and from there pressing <return> would cause the debugger to return you to FORTH after first removing DEBUGGER from the context stack.

5.4.2 Data Validation

Clearly, just stepping through the code like this is

not too meaningful by itself. Ideally, the programmer needs to be able to see the values of his data changing as the cursor steps through the word definitions. The debugger partially achieves this ideal by displaying the data stack at the bottom of the display.

There are, in fact, two stack display lines: one bears the name STACK(H), and the other the name STACK(D). Both denote the values stored in the data stack: STACK(D) shows the value of each stack cell in decimal, and STACK(H) shows the same values in hexadecimal. As the cursor steps through the code, the debugger updates the stack display lines.

Above the STACK(H) line is a line headed DEPTH. This is followed by the decimal count of the number of 16 bit items currently stored on the data stack (the value returned by the word DEPTH in fact). The count may occasionally indicate that there are more items on the stack than are being displayed on the screen. This is due to the limited line length of the display terminal. In 40 column mode at most the top 5 cells and in 80 column mode at most the top 11 cells will be displayed. The cell on the top of stack is the rightmost value shown, and the word EMPTY appears when the stack is empty.

Underneath the stack display lines is a line bearing the legend

```
          LOOP INDEXES J I: 35 4
```

The two numbers after the colon are the current values of the loop index words J and I. The I value is only meaningful if the program is currently executing an iterative DO loop, and the J value is only meaningful if the DO loop nesting is two or more and the cursor is not in the outermost loop.

Below the index line is a line headed #WORDS. this is followed by a count of the number of words that the cursor has stepped over. It is useful for measuring performance.

5.4.3 EMIT and KEY

Because the debugger requires exclusive use of the display, there is a potential conflict if you want to test words that use EMIT or KEY, either directly or indirectly. To circumvent the problem the debugger intercepts all EMIT and KEY calls, and reroutes them to a special part of the display.

Whenever a word EMIT's a character the debugger will display it on the line of the display which begins with the word EMIT. As successive characters are output they are displayed one after another on this line until the end of the line is reached. At this point the debugger clears the line and the next character is output at the start of the EMIT display. All non-printing characters are displayed as the carat (^) character.

If a word requests input directly from KEY, the debugger will position the cursor on the first blank of the line which begins with the text "KEY:". As soon as you press a key its ASCII value will appear on the stack, and its external representation will be displayed at the cursor position. If KEY is called indirectly, from QUERY for example, the cursor will appear to remain motionless on the KEY line as you type successive characters up to the <return>. The characters themselves will be displayed on the EMIT line as they are typed.

5.4.5 Examining Non-stack Data

Most applications contain variable and array declarations, as well as other application specific data structures. You can examine the value of such data, and even modify it, at any time during the course of a testing session.

If you press the ESC key, the debugger clears the display, outputs the word ESCAPE and calls the text interpreter. In other words, you can now do anything you could do outside the debugger, apart from load an overlay. Typically you will want to use ? to examine the values of variables and array elements, and

possibly even modify some of them.

When you have finished, and you are ready to continue with the test, simply type the word OK and press <return>. The screen you left will reappear with the cursor positioned where it was when you pressed ESC.

5.4.5 Debugger Skip Function

It is sometimes useful to be able to tell the debugger that you are not interested in stepping through the sequence of words which follow. Consider the following definition:

```
: EXAMPLE
    START DO SOMETHING LOOP ;
```

Suppose you have been once round the loop, have stepped all the way through SOMETHING, and are satisfied that is working alright. Instead of stepping through SOMETHING on every subsequent iteration of the loop, you can use the skip feature to suppress stepping of, in this case, the body of SOMETHING.

The skip command is invoked by pressing S instead of <return>, and is normally used in conjunction with the editor commands ->, <-, (Q), (Z), (N) and (P) which are available when you use the debugger.

Suppose that the cursor is positioned on the G of SOMETHING, and you want use S to skip on to the call of LOOP. Simply use -> to position the cursor at the P of LOOP, and press S. The debugger will allow execution of the application to continue unimpeded until the word at the cursor position, LOOP in this case, is reached. While the intermediate words are being executed the display lines below line 15 of the screen will be updated after every 100 words executed. When execution reaches the word at the cursor the debugger stops the program and waits for you to do something. If you press S, execution restarts and continues until the word is once more encountered. Alternatively you could choose to return to step mode by pressing <return>, or examine data by pressing ESC.

If you press a key while execution is proceeding in skip mode, the debugger will stop the program immediately and display the screen containing the next word to be executed with the cursor positioned on its last character. You can then continue in step mode or select another word to skip to.

There are no constraints placed on where in the application you are allowed to position the cursor other than that it must be on the last character of the name of a word in a definition belonging to the test set, and that the definition will be executed at least once more. If any of these conditions is not satisfied, the debugger won't be able to stop the program.

5.5 Errors

The debugger is a tool for tracing through code at the source level. Total control is given to you, the programmer. Be warned, however, that the debugger makes no attempt to prevent your application from corrupting itself, or the system.

If you lose control of the system, pressing RESET should enable you to regain control, although you will also have to call 80-COLS as well if you have an 80 column card.

If your application calls QUIT, ABORT or ABORT", execution will stop, the Apple will beep and you will be returned to FORTH.

5.6 Defining Words

If you load any defining words as part of a test set, then be prepared to have them traced if they are invoked during loading of the test set.

PART III SYSTEM DESCRIPTION

This part of the manual describes the principle structural features and extensions of the system. The information will be useful to those who require a deeper knowledge of the internal features of the implementation.

1. WORD SETS

Metacrafts FORTH supports the 79 Standard word set as defined in the October 1980 document published by the FORTH standards team. The only deviations from the standard concern the word +LOOP, and the vocabulary mechanism. The +LOOP deviation is described in the Glossary in Part IV. The vocabulary mechanism that we have implemented is described in chapter 3 of this part of the manual.

In addition to the required word set Metacrafts FORTH supports the two standard extensions, namely the Double Number and Assembler word sets.

All of the words in the standard word sets are defined in the glossary, suitably annotated to distinguish them from non-standard words.

Besides the standard words the system contains many non-standard definitions. Some of these have been taken from the Reference set defined in the 79 Standard document, and some are out of the fig-FORTH model. The remainder are either our own homebrewed words, or have been taken from one or other of the many FORTH publications that are available.

The function and use of those word sets that are either our own, or whose description is not readily available in one or other of the books on FORTH, are described in the following chapters.

It is important to remember that FORTH is essentially a toolkit for applications development, and that different implementations usually support their own word sets for such things as string handling, editing and so on. If you want to write an application which must run on different implementations of the language, you must make sure that all your words derive from the words in the 79-standard, and take care to avoid words which deviate from the standard definition.

2. INTERNAL DETAILS

This chapter looks at some of the principle internal details of the system. The information contained here will be of value to anyone thinking of using the Assembler, and also to anyone who intends producing new dictionary structures. The chapter assumes that the reader is familiar with the underlying principles of FORTH.

2.1 Direct Threaded Code

Unlike most implementations of FORTH which are based on the concept of Indirect Threaded Code (ITC), Metacrafts FORTH is based on Direct Threaded Code (DTC). Although there are some ardent FORTH enthusiasts who argue that ITC is a fundamental property of the language, we are of the opinion that ITC, DTC etc., are only implementation techniques in just the same way that p-code is one way of implementing Pascal. This view is supported by the fact that the 79 Standard document does not require the implementer to use any particular technique.

In DTC compilers the compilation address that is compiled for each word in a definition is the address of the machine code which is called when the word is interpreted by the address interpreter. In ITC compilers the compilation address is the address of a cell containing the address of the machine code which is called. The diagram below shows the difference quite clearly.

```
                  D I R E C T   T H R E A D E D   C O D E

          DTC thread
          |           |                              +-----------+
          |-----------|                              |  header   |
          |           |                              |           |
          |-----------|                              |-----------|
          | comp.addr |------------------------->    | M/C code  |
          |-----------|                              |           |
          |           |                              |           |
```

INDIRECT THREADED CODE

```
 ITC thread
|           |                 +-----------+
|-----------|                 |  header   |
|           |                 |           |
|-----------|                 |-----------|
| comp.addr |---------------->| code addr |---+
|-----------|                 |-----------|   |
|           |                 | M/C code  |<--+
|-----------|                 |           |
```

In the DTC implementation of a machine code definition the machine code itself is stored at the compilation address, while at the compilation address in an ITC implementation you will find a pointer to that code (usually the address of the next cell as in the diagram).

In the DTC implementation of a high-level definition the compilation address contains a JSR call to the machine code routine responsible for the type of definition. The ITC implementation of this would store the address of the responsible code at the compilation address.

2.2 The Interpreter Pointer

The principle register in the FORTH machine is the Interpreter Pointer (IP for short). It is stored in page zero. The IP always contains the address of the dictionary cell containing the compilation address of the word that is currently executing. This is shown in the following diagram.

```
                  DTC thread         +-----------+
    I P          |           |       |  header   |
 +--------+      |-----------|       |-----------|
 |        |----->| comp.addr |------>| M/C code  |
 +--------+      |-----------|       |           |
                 |           |       |           |
                 |-----------|
                 |           |
```

A consequence of this arrangement is that if the word currently executing is a high level word of some sort, then the top of the return stack will contain IP's value at the time the word was called. In other words, it contains the compilation address of the caller. ITC systems generally increment the value of IP before a call is made.

Be warned that any code you write which depends on the value stored on the return stack may not function correctly on some implementations of the language.

2.3 Stacks

The FORTH return stack is held in the 6502 hardware stack in page one of main memory.

The data stack is stored in page zero of main memory. The bottom of the stack is at the high address end of the page, and the stack itself grows towards low memory. The low byte address of the top of stack cell is held in the 6502's X-register. The user variable S0 contains the address of the bottom of stack. S0 is the first user variable in the user variable area, which is memory page two.

2.4 Dictionary Header

In Metacrafts FORTH the dictionary header format differs slightly from the one usually found in systems based on the fig-FORTH model. The format used for high level words is:

```
                              Header Format
                            +--------------+
Link field address ------->|  link field  |
                            |--------------|
Name field address ------->|  name field  |
                            |              |
                            |--------------|
Code field address ------->|  code field  |
                            |--------------|
Parameter field address -->|  parameter   |
                            |    field     |
                            |              |
```

The standard word FIND returns the cfa (code field address) of a word, and ' (tick) returns the pfa (parameter field address) of a word. The non-standard words CFA, PFA, NFA and LFA can be used to move about the dictionary header in the manner shown below:

```
pfa <---PFA--- cfa ---LFA---> lfa ---NFA---> nfa
                ^              |
                |              |
                +-----CFA------+
```

Be warned that these words are not the same as their namesakes in the fig-FORTH model in which the pfa plays a greater role than it does in the 79-Standard. It is also important to realise that the pfa has no meaningful interpretation for the header of a machine code definition. This is a consequence of the use of DTC where the code is located at the cfa, whereas it is at the pfa in ITC based implementations.

The structure of the name field is the same as the one used in the fig-FORTH model. Namely:

```
              +--------------------------------+
      nfa--->| 1 | P | S | c   o   u   n   t |
              +--------------------------------+
1st.byte----->|                                |
              |--------------------------------|
              |                                |
              :                                :
              |                                |
              |--------------------------------|
last byte---->| 1 |                            |
              +--------------------------------+
```

The count byte and last name byte always have the high order bit set equal to one. The precedence bit P is set to one for IMMEDIATF words, and the smudge bit S is set to one during the compilation of the definition. The five bit count is the actual number of bytes that were in the name, and the maximum number of name bytes stored is determined by the user variable WIDTH.

2.5 Defining Words

If the run-time code of a defining word is written in FORTH (i.e. because it follows DOES>), then on entry to the run-time code the top of the data stack will contain the pfa of the defined word.

If, however, the run-time code is written in machine code (i.e. because it follows ;CODE), then the run-time code can compute the pfa of the defined word by popping the value on top of the 6502 hardware stack (return stack) and adding one to it. If the run-time code isn't interested in the pfa, it should, in any case, pop the return stack and discard the value.

2.6 Memory Map

The diagram below depicts the system's use of main memory.

```
C000 ------>  +------------------------------+
              |   Disk II driver routines    |
B7E8 ------>  |------------------------------|
              |      Block buffers (2)       |
TOP ------->  |------------------------------|
              |                              |
              :  Free space for Dictionary   |
              |                              |
PAD+256       |------------------------------|
              |     String Buffer Area       |
PAD ------->  |------------------------------|
              | WORD & Numeric output buffer |
DP -------->  |------------------------------|
              |                              |
              :    D I C T I O N A R Y       :
              |                              |
800 ------->  |------------------------------|
              | Text & Lores graphics page 1 |
              |------------------------------|
PAGE 3 ---->  |    Terminal Input Buffer     |
              |------------------------------|
PAGE 2 ---->  |       User Variables         |
              |------------------------------|
PAGE 1 ---->  |        Return Stack          |
              |------------------------------|
PAGE 0 ---->  |   Data Stack+ System Data    |
              +------------------------------+
```

3. VOCABULARY MECHANISM

Due to the restrictions of the 79-Standard vocabulary mechanism we have developed an alternative system for Metacrafts FORTH which, we believe, is more flexible.

3.1 Context Stack and FIND

When a vocabulary name is executed, the lfa of the latest word in the vocabulary is placed on top of a stack known as the "context stack". The top item on the stack can be accessed via CONTEXT.

The 79-Standard word FIND, which is called by the text interpreter, is used to search the context stack, starting with the top vocabulary.

The non-standard word PRUNE, if it finds the address of CONTEXT on stack, drops the top of the context stack.

During system load the context stack is initialised with the FORTH vocabulary, and it can be reinitialised by calling TRUNK.

3.2 Current Definitions Stack and CREATE

The 79-Standard word DEFINITIONS copies the top of the context stack onto the top of the "current stack". The top item on the top of the stack can be accessed via CURRENT.

New definitions are placed at the front of the vocabulary on top of the current stack by CREATE. CREATE uses ?UNIQUE to test if the name of the new word already exists in the vocabulary on top of the current stack. If it does, then the message:

```
<name> NOT UNIQUE
```

is output.

All vocabularies are 'sealed', which means that no vocabularies chain to any other vocabulary. In practical terms, the lfa of the oldest definition in a vocabulary contains zero.

New vocabulary words created by VOCABULARY are added to the vocabulary on top of the current stack.

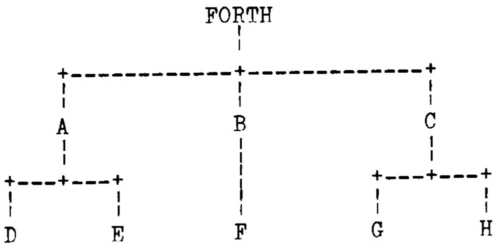
PRUNE, if it finds the address of CURRENT on stack, drops the top vocabulary off the current stack.

During system load the current stack is initialised with the FORTH vocabulary, and can be reinitialised by calling TRUNK.

The word DEFINED will drop the top item on both vocabulary stacks. It is intended to complement DEFINITIONS.

3.3 Vocabulary Tree

A useful way of viewing the structure of a set of vocabularies is as a tree with FORTH forming the trunk. Consider the following tree:



The definitions of the vocabularies A,B and C are part of FORTH. D and E are part of A. F is part of B. G and H are part of C.

Access to the contents of the vocabularies in such a tree is controlled by the context stack. For example, before a word in D can be executed, D must be on the context stack. But D itself cannot be executed until A is on the context stack, and so on. Furthermore, no new word can be added to a vocabulary until that vocabulary has been put on top of the current stack, and then it must first of all have been added to the context.

The context stack can, thus, be thought of as a way of dynamically controlling the search paths through the tree.

3.4 Compatibility with the 79-Standard

Provided that only one vocabulary, other than FORTH, is held on the context stack at any one time, the search order used by FIND is compatible with the 79-Standard.

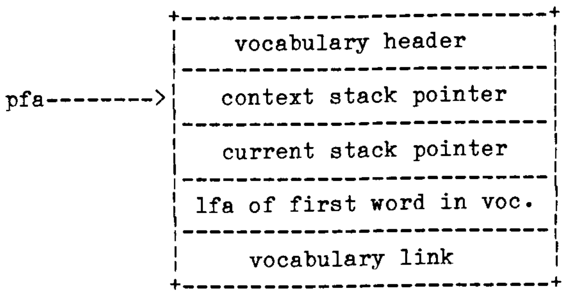
If DEFINITIONS is always used to add the vocabulary on the top of the context stack to the current stack, then creation of new words is also comaptible with the 79-Standard.

Furthermore, if these rules are adhered to, the values of CURRENT and CONTEXT are Standard-compatible.

3.5 Internal Organisation

Both vocabulary stacks are actually implemented as linked lists which chain through the parameter fields of the vocabularies on stack. CONTEXT points at the pfa of the vocabulary on top of the context stack, and CURRENT points at the pfa of the vocabulary on top of the current stack.

Four items of data are maintained at the pfa of each vocabulary, as shown in the following diagram:



If the vocabulary in the diagram is in the context stack, then the 'context stack pointer' will contain the pfa of the vocabulary below it on the stack, or zero if it is at the bottom of the stack. Similarly, if it is also on the current stack, then 'current stack pointer' will contain the pfa of the vocabulary below it on the stack, or zero if it is at the bottom of the stack. The 'lfa of first word in voc' is the head of the chain of definitions in the vocabulary. It is zero if the vocabulary is empty, otherwise it points at the latest definition. Finally, the 'vocabulary' is the address of the corresponding word in the vocabulary definition defined before it, or zero if FORTH.

One consequence of the implementation of the stacks as linked lists is that no vocabulary can appear more than once on the same stack. If you try and break this rule, the system will loop the first time it tries to search through all the vocabularies in the stack.

4. EXECUTION VECTORS

Execution vectors are a special form of user variable whose values are always the cfa of some word in the dictionary. Executing the name of an execution vector caused the word at the cfa assigned to it to be executed.

Several execution vectors have been used in Metacrafts FORTH, primarily to allow users to reconfigure the input/output words built into the system. See Part I chapter 4 for further details of these.

You can define your own execution vectors with the word EXECUTE:, and you can assign the cfa of a word to a vector by means of IS. Remember that IS always expects a cfa on the stack, so when it is used in a colon definition you need to write something like the following:

```
: AWORD  .... [ FIND <name> ] LITERAL IS AVECTOR .. ;
```

The word WAS leaves the current value of an execution vector on stack.

5. OVERLAYS

Metacrafts FORTH supports a simple dictionary overlay mechanism. This is an effective way of reducing the time taken to load applications by making a copy on disc of part of the dictionary. The editor and debugger are examples of the use of overlays by the system itself

5.1 Creating Overlays

Suppose you have written a large application which you want to save on disc as a dictionary overlay. First you must ensure that the first dictionary definition in the overlay, called the load point, is a double number variable. The first word in the editor, for example, is:

```
2VARIABLE OVERLAY
```

Next estimate how many mass storage blocks will be required to hold the overlay. The following word definition computes this value:

```
: OVLSIZE ( +++ #BLOCKS )
    HERE FIND LFA - ( #BYTES IN OVERLAY)
    B/BUF /MOD      ( COMPUTE #BLOCKS + OVERFLOW)
    SWAP DUP >R     ( SAVE OVERFLOW )
    960 /           ( ?ROOM FOR VOC DATA)
    R> 0>           ( IS OVERFLOW > ZERO)
    IF 1+ THEN      ( BLOCK FOR OVERFLOW)
    +               ( BLOCKS REQUIRED)
    SPACE .         ( PRINT IN OUT) ;
```

If you type OVLSIZE followed by the load point name, the number of blocks required will be displayed. The strange value 960 is necessary because the last block in the overlay contains (6 x N + 4) bytes of additional data, where N is the number of vocabulary words in the dictionary. If more than 10 vocabularies have been declared you should reduce the constant 960 by 6 for each additional one.

Once you have worked out how many blocks are required, find a contiguous set of spare blocks on the disc

which is large enough to hold the overlay.

Suppose your load point is called TASK and that OVLSIZE has computed that you need 10 blocks. Suppose further that block 35 of the selected disc is the first of a sequence of at least 10 free blocks. Then, if you type:

```
35 SAVE TASK
```

FORTH will save your application on disc starting at block 35 (your disc must be in the currently selected drive). When it has finished it will output the message:

```
45 IS THE LAST BLOCKok
```

to confirm that 10 blocks were necessary.

SAVE always saves the part of the dictionary that exists between the load point and the value returned by HERE. In addition, it saves the value of DP and information from the pfa of each vocabulary in the total dictionary.

5.2 Loading Overlays

The word RESTORE is used to load overlays from mass storage. Suppose you want to load the application which was SAVE'd in section 5.1. RESTORE expects to find the start and end block numbers of the overlay on stack, and so the call would look like:

```
35 45 RESTORE
```

The call will place the overlay back at the same location from which it was SAVE'd. It will also restore the value of DP, and the information saved for each vocabulary. Finally, it resets the vocabulary stacks by calling TRUNK.

It is important to ensure that the contents of the dictionary preceding the load point at RESTORE time are the same as they were at SAVE time, otherwise dictionary corruption will ensue, and the results will be unpredictable.

5.3 System use of Overlays

When you boot Metacrafts FORTH, the nucleus of the system is loaded from the first 3 tracks of the system disc. This contains the Disk II driver routines and a precompiled dictionary containing enough words to support compilation of the rest of the system.

Once it has been loaded the nucleus is entered and, after performing some initialisation, it executes a LOAD command for block 13. If you examine block 13 you will see that it contains a RESTORE request. This loads the BOOT3 overlay which contains the rest of the dictionary. You will find the code belonging to BOOT3 in blocks 34-77 and 138-139 of the system disc.

Blocks 138 and 139 contain the definitions of the words which load the development tools described in Part II.

You can, of course, modify or otherwise enhance any of the code on the system disc (other than the nucleus), and generate new overlays. You will have to do this, for example, if you change any of the printer constants in block 106.

Before making any changes it is vitally important that you make backup copies of the system disc, and observe the following rules:

1. If you increase/decrease the number of blocks required for a system overlay, then you will have to modify the overlay loader definition and remake the system overlays.

2. If you modify the code in BOOT3 such as to increase/decrease its dictionary requirement, then you will have to remake both the system and your own overlays.

The code in block 33 will automatically remake the system if it is loaded as follows:

```
0 FENCE !   FORGET BOOT3   33 LOAD
```

5.4 Building a Turnkey Disc

When your application is working you can generate what is known as a turnkey disc for it. This is a disc which boots FORTH and the application, and enters it automatically. The key to this lies in the fact that the nucleus doesn't care what is in block 13, it simply loads it.

This means that you can produce, using COPY, a disc which contains blocks 1-12 of the system disc and an overlay containing your application stored somewhere beyond block 13. In addition you you must provide code in block 13 which will load and enter the application. Suppose you have saved your application in blocks 14-60, and the entry point is called WORD-PROCESSOR. The following code is sufficient to load and enter it:

```
14 60 RESTORE
FIND WORD-PROCESSOR
EXECUTE
```

All applications should re-vector QUIT so that any errors are dealt with by the application. If this is not done, someone using the application may find him or herself suddenly talking to FORTH!.

6. BUFFER & HEAP MANAGEMENT

There are two special word sets provided for performing memory management tasks. The buffer manager is part of BOOT3 and exists in code form in blocks 53 and 54. It is used by COPY. The heap manager is stored in source form in blocks 121-124. It is not used by any part of the system.

The system normally runs with two block buffers which are located in high memory, just below the Disk II drivers. Some applications (COPY is an example) may operate more efficiently using more than two buffers. The buffer manager words have, therefore, been providedto enable applications to control the number of block buffers in the system.

The heap manager provides an alternative form of dynamic memory management. It provides functions similar to the NEW and DISPOSE functions found in Pascal. The programmer uses ALLOT-HEAP to allocate a block of memory for use as a heap. The word -HEAP is used to obtain space on the heap, and +HEAP is used to return space to the heap. A 'first-fit' algorithm is used to manage the heap, and a garbage collection algorithm collects and recombines unused space if no block of the requested size can be found. The heap manager aborts with the message HEAP FULL if it is unable to satisfy a request for store.

It is important to make sure that you allocate enough space to the heap, otherwise frequent calls to the garbage collector will have an adverse effect on performance. One way of being able to tell if the heap is large enough is to place a call to BELL before the call to COMPACT on line 11 of screen 123. Every time the garbage collector is called the Apple speaker will beep.

7. LANGUAGE CARD

If your system has a memory extension that is compatible with the Apple Language Card, then you can use the extra memory space for storing the heap, or as a mass storage device, or for any other purpose you discover. The system will not use the space automatically for dictionary storage. You can use it for this purpose if necessary, by arranging to reset DP to point at an address beyond CFFF, and TOP to an address below FFFA.

If you write assembler routines which call the Apple monitor, then you must precede the monitor call by a call to LCOFF, and follow it by a call to LCON (see the Assembler vocabulary). This is necessary because the system runs with the language card permanently selected if it is present. In the current version of the system LCON always selects bank 2 of the language card.

8. LOCAL VARIABLES

Occasionally, when writing a definition, the amount of juggling of the data stack can become complex and unwieldy. The ARGUMENTS and RESULTS words were developed for this purpose.

Suppose you want to write a word which computes the volume, surface area and edge length of a box given the length, width and height. Without using local variables, this can be quite a tricky problem. The solution using local variables is shown below.

The phrase "3 ARGUMENTS" assigns the names of local variables S1 to S9 to the top nine stack positions, with S1-S3 returning the values of the 3 stack values that were there on entry to BOX. S4-S9 are zero-filled and the stack pointer is set to S9.

S1 to S9 act as local variables returning their contents, not their addresses. To write to one you precede its name with the word TO. For example, 5 TO S4 stores a 5 in S4. The word +TO is used to increment the value of a local variable. For example, 5 +TO S4 will add 5 to the value of S4.

After all the calculating is done, the phrase "3 RESULTS" leaves 3 results on the stack relative to the stack position when ARGUMENTS was called. All intermediate stack values are lost.

```
: BOX 3 ARGUMENTS ( height S1, length S2, width S3)
  S1 S2 S3 * * TO S4 ( volume)
  S1 S2 * 2*  S2 S3 * 2*  S1 S3 * 2*  + + TO S5 ( area)
  S1 4 *  S2 4 *  S3 4 *  + + TO S3  ( edge)
  S5 TO S2
  S4 TO S1
  3 RESULTS   ( volume S1, area S2, edge length S3) ;
```

9. 6502 ASSEMBLER

Metacrafts FORTH is provided with a machine language assembler to create execution procedures that would be time inefficient if written as colon definitions. It is intended that structured code be written for clarity of expression. Functions may be written first in FORTH, tested, and then rewritten in machine code, using the assembler, with the minimum of restructuring.

9.1 The Assembly Process

Code assembly just consists of interpreting with the ASSEMBLER vocabulary on top of the context stack. During assembly of CODE definitions, FORTH continues interpretation of each word found in the input stream. Assembly does not takes place in compile mode as in colon definitions. During assembly, the word being assembled is "smudged" until END-CODE which only "unsmudges" it if the data stack integrity has not been destroyed.

9.2 Run-time & Assembly-time

One must be careful to understand at what time a particular word definition executes. During assembly, each assembler word (such as # or JMP,) in the input stream is executed. Its function at that time is called "assembling" or "assembly-time". This function typically involves such things as op-code generation, address calculation, mode selection etc.

The later execution of the generated code is called "run-time". This distinction is particularly important with the conditional words. At assembly-time each such word (i.e. IF, BEGIN, etc.) itself runs to produce machine code which will later execute at what is labelled "run-time" when its named code definition is executed.

9.3 Security

Numerous checks for user errors are made within the assembler:

-- All items placed on the stack during assembly must be removed before END-CODE.

-- Control structures must be properly nested and paired.

-- Address modes and operands must be allowable for the op-code.
If an error occurs during assembly, END-CODE never gets executed, which means that the word remains smudged to prevent its use in later definitions.

9.4 Op-codes

The bulk of the assembler vocabulary consists of dictionary entries for the op-codes. The following op-codes have no operands:

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

When any of these are executed, the corresponding 6502 op-code byte is assembled into the dictionary.

The following op-codes usually take an operand:

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

If an operand is specified it must already be on the stack. An address mode may also be specified. If none is given, the op-code uses page zero or absolute addressing. The address modes are determined by:

Symbol	Mode	Operand
.A	accumulator	none
#	immediate	8 bits only
,X	indexed X	page zero or absolute
,Y	indexed Y	page zero or absolute
X)	indexed indirect X	page zero only
)Y	indirect indexed Y	page zero only
)	indirect	absolute only
none	memory	zero page or absolute

Here are some examples of FORTH versus conventional assembler. Note that the operand comes first, followed by any mode modifier, and then the op-code mnemonic (note: the comma is also part of the mnemonic). This arrangement makes the best use of the stack at assembly time. Also each FORTH word is set off from its neighbour by blanks, as is required for all FORTH source text.

```
          .A ROL,    ROL A
         1 # LDY,    LDY #1
     DATA ,X STA,    STA DATA,X
     DATA ,Y CMP,    CMP DATA,Y
        6 X) ADC,    ADC (06,X)
    POINT )Y STA,    STA (POINT),Y
    VECTOR ) JMP,    JMP (VECTOR)
```

The words DATA and VECTOR specify machine addresses. In the case of "6 X) ADC," the operand memory address $0006 was given directly. This is occasionally done if the usage of a value doesn't justify devoting the dictionary space to a symbolic value.

9.5 Stack Addressing

The data stack is located in page zero, and is usually addressed by "z-page,X". The stack starts near the end of the page and grows towards low memory. On entry to a code definition the data stack pointer will be in the X index register, and the definition is responsible for maintaining its integrity. Incrementing X by two removes an item on the stack, while decrementing X by two makes room for another item on the stack.

Sixteen-bit values are placed on the stack according to the 6502 convention of storing the low byte at the low address, and the high byte at the high address. This technique allows the indexed indirect X address mode to be used with an address held on the stack.

Because the last and last-but-one items placed on the stack are the most frequently accessed, the support words BOT for Bottom and SEC for Second are provided. Using

```
          BOT LDA, assembles LDA (0,X) and
          SEC ADC, assembles ADC (2,X)
```

Here is a pictorial example of the data stack in page zero.

```
                    |           |
                    |           |
                    |-----------|
                    | SEC high  |
                    |--       --|
                    | SEC low   |
                    |-----------|
                    | BOT high  |
                    |--       --|
                    | BOT low   |<-- X offset above 0
                    +-----------+
```

The following code or's together four bytes on the stack:

```
     BOT LDA,   BOT 1+ ORA,   SEC ORA,   SEC 1+ ORA,
```

To obtain the 14th. byte on the stack, use BOT 13 + LDA,

9.6 Return Stack

The return stack is located in the 6502 hardware stack in page 1. It starts at $01FE and grows towards $0100. No lower bound is checked as page 1 has sufficient capacity for the majority of applications.

The 6502 S register points to the next free byte below the last item stored on the return stack. The byte order is the same as on the data stack.

The PHA, and PLA, op-codes can be used to store and retrieve bytes on the return stack. To operate on an arbitrary byte, the method to be used is:

1) Save X in XSAVE.

2) Execute TSX, to get the S register in X.

3) Use RP) to address the low byte of the last item stored on the stack. Offset this to address older items on the stack.

4) Restore X from XSAVE.

The following example non-destructively tests that the second item on the return stack is zero.

```
CODE IS-IT ( ZERO?)
     XSAVE STX,   TSX,    ( steps 1,2)
     RP) 2+ LDA,  RP) 3 + ORA,  ( or 2nd item's bytes)
     0= IF, INY, THEN, ( bump Y to 1 if item zero)
     TYA,  PHA,   ( save boolean)
     XSAVE LDX,   ( restore X, step 4)
     PUSH JMP,    ( exit to push boolean on data stack)
END-CODE
```

9.7 FORTH Registers

Several FORTH registers are available only at the assembly level and have been given names that return their memory addresses. These are:

IP -- address of the Interpreter Pointer which contains the address of the cell being interpreted by the address interpreter.

IR -- address of the indirect register which is sometimes used to hold addresses for the indirect jump instruction.

XSAVE -- address of the register for temporary storage of the X register.

UP -- address of the register which contains the address of the first byte of the user area.

N -- address of a utility area in zero page from N-1 to N+7.

9.8 CPU Registers

When the address interpreter passes control to a CODE definition certain conventions must be maintained:

1) The Y register is zero, and may be freely used.

2) The X register contains the data stack offset, and must not be corrupted. CODE definitions can add/remove data stack items if they modify X accordingly.

3) The S register contains the return stack pointer.

4. The accumulator may be freely used. Its value on entry is undefined.

5. The processor is in binary mode and must be returned in that mode.

9.9 SETUP & N

When absolute memory registers are required, use the N-area in page zero. These registers may be used as pointers for indexed/indirect addressing, or for temporary values. It is very important to note that many FORTH words use the N-area, in particular the arithmetic routines involving multiplication and division.

It is often necessary to move stack values to the N-area. CMOVE and <CMOVE are typical examples of this. The routine SETUP has been provided for this purpose. Upon entering SETUP the accumulator specifies the number of 16-bit stack values to be moved to the N-area. At most 4 items may be moved. The call to move 3 items is:

```
3 # LDA,   SETUP JSR,
```

The effect on the stack and on N of this call is:

```
      Stack before    N after     Stack after
           H                           H
           G                   BOT-->  G
          ---
           F            F
           E            E
          ---          ---
           D            D
   SEC-->  C            C
          ---          ---
           B            B
   BOT-->  A            A   <--N
```

9.10 Control Flow

FORTH discards the usual convention of instruction labels in preference to using structured programming techniques.

First, each FORTH word name is permanently included in the dictionary. This allows procedures to be located and executed by name at any time, as well as be compiled within other definitions. Be warned, however, that if you assemble the address of a colon definition within a CODE definition, the run-time effect is undefined.

Secondly, within a code definition, branching is performed using control structures similar to the ones used in colon definitions. The major difference concerns the run-time control of branching in a code definition. In a colon definition flow of control is determined by flags stored on the data stack. In assembler definitions control flow is determined by the processor status bits. The programmer must indicate which bit to test just before a conditional branching word.

The conditional specifiers are:

specifier	meaning	processor status
CS	test carry set	C=1
0<	test byte less than zero	N=1
0=	equal to zero	Z=1
CS NOT	test carry clear	C=0
0>=	test positive	N=0
0= NOT	test not equal to zero	Z=0

The overflow status bit is so rarely used that it has not been included. If it is required, add 50 CONSTANT VS to screen 62. This tests for overflow being set.

9.11 Looping

A conditional loop is formed by placing the instructions to be repeated between BEGIN, and UNTIL,. Consider:

```
6 # LDA,   N STA,
BEGIN,   PORT DEC,
     N DEC,
0= UNTIL,
```

An unconditional loop is formed by placing the instructions to be executed between BEGIN, and AGAIN,.

Consider:

```
6 # LDA,   N STA,
BEGIN,
     N DEC, 0= IF, NEXT JMP, THEN,
     PORT DEC,
AGAIN,
```

In this example, the counter is decremented before PORT is decremented, which means that the branch to NEXT will be made after 5, and not 6, decrements of PORT. AGAIN, unconditionally passes control to BEGIN,. Because there are no local labels, the only way of leaving an unconditional loop is to branch to an external address such as NEXT.

9.11 Conditional Execution

Paths of execution may be chosen in CODE definitions in the same way as in colon definitions. In this case, however, the branch decision is based on the value of a bit in the processor status register.

```
PORT LDA,  0=
IF, ( CODE IF PORT=0) THEN,( continue) ...
```

In this example, the accumulator is loaded from PORT. The zero flag is tested and, if it is a 1, the code following IF, is executed. Otherwise the code following THEN, is executed.

The ELSE, word allows the programmer to specify a false branch in a conditional structure.

```
PORT LDA, 0= IF, ( TRUE BRANCH) ELSE, ( FALSE BRANCH)
THEN,
```

In this example, the value of PORT will cause one of the two branches to be selected, before continuing execution after THEN,.

9.12 Nested Control Structures

Control structures may be nested according to the usual conventions of structured programming. That is, each control structure begun must be terminated before a preceding control structure which has not already been terminated. An ELSE, must pair with the immediately preceding IF,.

9.13 Returning Control to the Address Interpreter

There are a variety of ways of returning to the address interpreter, depending on the stack action to be performed first. These functions are already in the nucleus, and they all pass control to the address interpreter at NEXT when they have finished:

POP	remove one 16-bit stack value
POP2	remove two 16-bit stack values
PUSH	add a 16-bit value to the stack

PUT write a 16-bit value over the value at BOT

PUSHYA add a 16-bit value to the stack, with Y as the high byte, and the accumulator as the low byte.

The convention for PUSH and PUT is:

1. push the low byte onto the machine stack

2. leave the high byte in the accumulator

3. jump to PUSH/PUT.

A branch to NEXT should be made if no juggling of the stack is necessary.

9.14 Reset and BRK

The monitor reset and BRK vectors are set up to cause FORTH to perform a warm start. This means that the system is restarted from ABORT without reinitialising system data. An assembler routine can alter these vectors, which are in page 3, as and when necessary. If the monitor is called, care must be taken to remember to turn off a language card if one is present.

10. STRINGS, GRAPHICS & GAME CONTROLLERS

Metacrafts FORTH provides a powerful set of words for processing character strings. Some of these are based on the UCSD extensions to Pascal. In our system a character string consists of a byte containing the length of the string (excluding the length byte), followed by the characters forming the string.

Strings are ordered according to the following rules. A string s0 is said to precede a string s1 if:

1) string s0 is shorter than s1 and each character in s0 is equal to the corresponding character in s1, or

2) string s0 is shorter than s1, and the first character in s0 that does not match the corresponding character in s1 precedes it in the ordering of ASCII characters.

The longest string consists of 255 characters, and the shortest consists of zero characters. The first character in a string has index 1, the second 2, and so on.

The string word set is contained in BOOT3, and its source code is contained in screens 65-73 on the system disc.

In addition to the string extensions, words are provided for handling low and high resolution graphics, and the Apple game controllers. The low resolution graphics routines use the primary screen. The secondary screen is occupied by the bottom of the dictionary. The high resolution routines select the secondary screen. The primary screen is occupied by the top of the nucleus. You have a choice of using the Apple BASIC high resolution graphics routines including the same shape table facilities (the structure of shape tables is the same as that described in the Applesoft manual). There are also a set of turtle graphics routines if you feel like using turtle coordinates.

If you choose to use Cartesian coordinates, you can select your own coordinate scale to map onto pixel coordinates. Remember that pixel coordinate 0,0 is in the top left-hand corner, and that coordinate 279,191 is in the bottom right corner. Suppose you want to set the origin (0,0) in the bottom left corner and (100,100) in the top right corner. To do this call DISPLAY-SCALE as follows:

```
0 100 100 0 DISPLAY-SCALE
```

This informs the graphics system that x-pixel 0 is your x value 0, that x-pixel 279 is your x value 100, that y-pixel 0 is your y value 100, and that y-pixel 191 is your y value 0.

In your program you work in terms of your coordinate system, and use XY-DISPLAY and DISPLAY-XY to convert to and from the pixel coordinates, whichare expected by the graphics routines.

You can store the low/high resolution screens on disc by moving their contents to block buffers obtained from **BUFFER**, and which you then UPDATE.

The graphics word set is stored in source form in screens 125-137 on the system disc. The first word on screen 125, called GRAPHICS, is provided to help you make an overlay from the definitions. Note that the dictionary pointer has been moved to prevent code being compiled into the hires screen.

All the words supporting strings and graphics are fully described in the glossary.

PART IV GLOSSARY

This part of the manual consists largely of the glossary of words in the Metacrafts FORTH system organised according to function. There is also a dictionary of FORTH terms to help remove any misunderstandings about the terminology used in the manual.

1. DEFINITION OF TERMS

Numerous technical terms are used in the definition of FORTH. Those terms that are peculiar to FORTH itself are defined in this list.

address,byte

An unsigned number that locates an 8-bit byte in a standard FORTH address space over (0..65,535). It is a native machine address. Address arithmetic is modulo 65,536 without overflow.

address,compilation

The numerical value equivalent to a FORTH word definition, which is compiled for that definition. The address interpreter uses this value to locate the machine code corresponding to each definition.

address,code field

See compilation address.

address,link field

The address of the field of a word definition used to store the internal vocabulary pointer.

address,name field

The address of the first byte of a word's name in the dictionary.

address,native machine

The natural address representation of the host computer.

address,parameter field

The address of the first byte of memory associated with a word definition for the storage of compilation addresses in a colon definition and data in variable/constant definitions. Machine code word definitions do not have a parameter field address.

arithmetic,integer

All arithmetic operations are performed using signed 16 or 32 bit two's complement integer arithmetic, unless otherwise stated.

block

The unit of data from mass storage, referenced by block number. A block contains 1024 bytes of data. The mapping from block number to physical storage address is device and implementation dependent.

block buffer

A memory area where a mass storage block is maintained. Metacrafts FORTH lets the user control the number of block buffers used.

byte

An assembly of 8 bits. In reference to memory it is the storage capacity of 8 bits.

cell

A 16-bit memory location. The n-th cell contains the 2n-th and (2n+1)-th bytes of the FORTH address space. The order of bytes (byte sex) is unspecified.

character

A 7-bit number which represents an external character. The ASCII character set is considered standard and is used by this implementation. When used in a field of more than 7 bits, the higher order bits are zero.

compilation

The action of accepting text words from an input stream and placing corresponding compilation addresses in a new dictionary entry.

defining word

A word that, when executed, creates a new dictionary entry. The new word name is taken from the input stream. If the input stream is exhausted before the new name is available, an error condition exists. Common defining words are:

CONSTANT VARIABLE CREATE

definition

See 'word definition'.

dictionary

A structure of word definitions in computer memory. The dictionary entries are organized into vocabularies, and are locatable by name. The dictionary is extensible, growing toward high memory.

equivalent execution

For the execution of a standard program (a program written using only words taken from a 79-Standard standard word set), a set of non time-dependent inputs will produce the same non time-dependent outputs on any 79-Standard FORTH system with sufficient resources to execute the program. Only standard source code is transportable.

error condition

An exceptional condition which requires action by the system other than the expected function. A complete list of the error conditions detected by Metacrafts FORTH appears elsewhere in this reference manual.

false

A zero number representing the false condition flag.

flag

A number that may have two logical states, zero and non-zero. These are named 'true' = non-zero, and 'false' = zero. Standard word definitions leave 1 for true and 0 for false.

glossary

A set of word definitions given in natural language describing the action of the computer when it executes the word.

immediate word

A word defined to automatically execute when encountered during compilation. Typical immediate words are IF, DO, DOES>.

input stream

A sequence of characters available to the system, for processing by the text interpreter. The input stream is usually either the computer terminal or mass storage. >IN and BLK specify the input stream. Words using or altering >IN and BLK are responsible for maintaining and restoring control of the input stream.

interpreter, address

The set of word definitions which interprets sequences of FORTH compilation addresses by executing the word definition for each one.

interpreter,text

The set of word definitions that repeatedly accepts a word name from the input stream, locates the corresponding dictionary entry, and starts the address interpreter to execute it. Text in the input stream interpreted as a number leaves the corresponding value on the data stack. When in the compile mode, the addresses of FORTH words are compiled into the dictionary for later interpretation by the address interpreter. In this case, numbers are compiled, to be placed on the data stack when later interpreted. Numbers are accepted unsigned or negatively signed, according to the current number base specified by BASE.

load

The acceptance of text from a mass storage device, and execution of the dictionary definitions of the words encountered. This is the general method for compilation of new word definitions into the dictionary.

mass storage

Data is read from mass storage in the form of 1024 byte blocks. This data is held in block buffers. When indicated as UPDATEd (modified), data will be ultimately written to mass storage.

number

When stored in memory the byte order of a number is unspecified. When represented on the data stack, the higher 16 bits, with sign, of a double number are most accessible. When in memory, the higher 16-bits are at the lower address and storage extends over four bytes toward high memory. The byte order within each 16-bit field is unspecified.

The following number types and ranges may be specified:

type	range	minimum field
bit	0..1	1
character	0..127	7
byte	0..255	8
number	-32768..32767	16
positive number	0..32767	16
unsigned number	0..65535	16
double number	-2147483648.. 2147483647	32
positive double no.	0..2147483647	32
unsigned double no.	0..4294967295	32

output,pictured

The use of numeric output primitives, which convert numerical values into text strings. The operators are used in a sequence which resembles a symbolic picture of the desired text format. Conversion proceeds low digit to high, from high memory to low.

return

The means of terminating text from the input stream. Conventionally 2 null (ASCII 0) characters indicate end of text in the input stream. These characters are left in response to the actuation of the <return> key of the operator's terminal, and upon termination of a block transfer from mass storage.

screen

Textual data arranged for editing. By convention a screen consists of 16 lines (numbered 0 to 15) of 64 characters each. Screens usually contain program source text, but may be used to view/modify mass storage data. The first character of a screen occupies the first byte of a mass storage block, and is the beginning point for text interpretation. The screen editor is normally used to create and maintain screens of source code. A left-right scrolling technique makes it possible for all 64 characters of each line to be viewed at terminals with a screen width of less than 67 characters.

source definition

The text consisting of word names suitable for execution by the text interpreter. Such text is usually arranged in screens and maintained on a mass storage device.

stack,data

A last in, first out list consisting of 16-bit binary values. The stack is primarily used to hold intermediate values during execution of word definitions. Stack values may represent numbers, characters, addresses, boolean values, etc.

When the term 'stack' is used, it implies the data stack.

stack,return

A last in, first out list which contains the machine addresses of word definitions whose execution has not been completed by the address interpreter. As a word definition passes control to another definition, the caller's compilation address is placed on the return stack enabling the address interpreter to resume execution of the 'calling' word when the called word 'returns' to its caller.

The return stack may be used with great caution within a definition for the temporary storage of data.

string

A sequence of bytes containing ASCII characters, located in memory either by an initial byte address and a character count, or the address of a byte containing the character count, followed by the characters themselves.

transportability

This term indicates that equivalent execution results when a program is executed on other than the system on which it was created. See 'equivalent execution'.

true

A non-zero value represents the true value flag. Any non-zero value will be accepted by a standard word as true. All words in Metacrafts FORTH return 1 when leaving a true flag.

user area

An area in memory which is used for the storage of user variables and execution vectors.

user variables

These are a set of variables containing system

information. The user can add his own variables to this set. If the dictionary is held in ROM memory then system variables must be held elsewhere. This 'elsewhere' is the user area in RAM memory. The dictionary entry for a user variable consists of a constant index to the variable's value in the user area.

vocabulary

A list of word definitions ordered according to time of definition. The latest definition appears at the end of the list. Vocabularies may be used to reduce dictionary search time and to separate logically different word sets from each other.

word

A sequence of characters terminated by at least one blank or return. Words are usually obtained from the input stream.

word definition

A named FORTH execution procedure compiled or assembled into the dictionary. Its execution may be defined in terms of machine code generated by the assembler, or as a sequence of compilation addresses generated by the compiler.

word name

The name of a word definition. Names are distinguished by their length and up to a maximum of 31 characters. A name may not contain an ASCII null, blank or return character.

word set

A group of word definitions related by common characteristics and frequently maintained in a separate vocabulary.

The set of word definitions forming the 79-Standard is known as the "required word set". The assembler and double number words are "extension word sets", and the "reference word set" contains formerly standardized words, and words that are candidates for standardization.

2. GLOSSARY NOTATION

This section introduces the notation used to define words in the gloassaries.

Stack Notation

The first line of each definition describes, in brackets, the effect that execution of the definition has upon the stack. The notation used is one of:

(before --- after)

(before +++ after)

In this notation 'before' denotes a list of stack values denoting the arguments before execution, and 'after' denotes the list of results left after execution. In all cases the top of stack (most accessible entry) is shown to the right.

The second notation is identical in meaning to the first: however, the +++ is used to indicate the fact that the word reads information from the input stream (VARIABLE is an example of such a word).

Attributes

Capitalized symbols following the stack specification indicate attributes of the word being defined:

S	The word belongs to the 79-Standard required word set
E	The word belongs to a standard extension word set.
R	The word belongs to the reference set.
C	The word may only be used in a colon definition.
I	The word is IMMEDIATE.
U	The word is a user variable.
V	The word is an execution vector.

Capitalization

Word names are written in capital letters. The text interpreter will interpret lower-case word names as upper-case. Thus 'dup' will be interpreted as 'DUP'. Note that a word may be given a name containing lower-case letters, but the address interpreter will still treat these as upper-case during disctionary searches.

Pronunciation

The natural language pronunciation of FORTH word names appears as part of the glossary when the correct pronunciation is not intuitively obvious.

Stack Parameters

Unless otherwise stated, all references to numbers apply to 16-bit signed integers.

The following abbreviations are used to denote stack values:

addr

A value denoting the address of a byte within the FORTH memory space.

byte

A value representing an 8-bit byte. It occupies a 16-bit stack cell.

char

A value representing a 7-bit ASCII character. It occupies a 16-bit stack cell.

d

A 32-bit signed double number occupying 2 16-bit stack cells. The most significant 16-bits, with sign, is most accessible on stack.

ud

An unsigned double number as in d.

flag

A numerical value interpreted as having one of 2 logical (boolean) states. Flags occupy a single stack cell.

n

A 16-bit signed integer. It is often used to denote a stack cell of any type. This is particularly true of the stack manipulating words.

u

A 16-bit unsigned number.

s

The address of a character string containing the string length in its first byte.

Other abbreviations are used as and when necessary. The meaning of such abbreviations should be clear from the context, and may be assumed to denote a 16-bit stack cell unless otherwise specified.

3. STACK MANIPULATION

DROP (n ---) S

Remove and discard the top stack cell.

MDROP (nm .... n1 m ---)

Remove and discard the m stack cells below the top cell.

DUP (n --- n n) S

Duplicate the top stack cell.

SWAP (n1 n0 --- n0 n1) S

Exchange the top two stack cells.

OVER (n1 n0 --- n1 n0 n1) S

Duplicate the second stack cell.

ROT (n2 n1 n0 --- n1 n0 n2) S

Rotate the top 3 stack cells. In other words, move the third stack element to the top of stack. "rote"

-ROT (n2 n1 n0 --- n0 n2 n1)

This is the same as ROT, but in the opposite direction. In other words, move the top stack element to the third position. "minus-rote"

PICK (nm ... n1 m --- nm ... n1 nm) S

Leave a copy of the m-th stack cell in place of the value m. The value of m must be greater than 0. If m is less than or equal to zero, or exceeds the depth of the stack the outcome is undefined.

ROLL (nm ... n1 m --- nm+1 ... n1 nm) S

Move the m-th stack cell to top of stack in place of m. The value of m must be greater than 0. If m is less than or equal to zero, or exceeds the depth of the stack the outcome is undefined.

?DUP (n --- n (n)) S

Duplicate the top stack cell only if it is non zero. "query-dup"

DEPTH (--- u) S

Leave the number of cells in the data stack before DEPTH was executed.

>R (n ---) S,C

Transfer the top stack cell to the return stack. Note that every >R must be balanced by a R> in the same control structure nesting level of a colon definition. >R must be used in compile mode only, otherwise its effect is undefined. "to-r"

R> (--- n) S,C

Transfer the cell on the top of the return stack to the data stack. R> must only be used in compile mode otherwise its effect is undefined. "r-from"

R@ (--- n) S,C

Copy the cell on the top of the return stack to the top of the data stack. R@ must only be used in compile mode otherwise its effect is undefined. "r-fetch"

2DUP (d --- d d) E

Duplicate the double number on the top of stack.

2DROP (d ---) E

Remove and discard the double number on the top of stack.

2SWAP (d1 d0 --- d0 d1) E

Exchange the two double numbers on the top of stack.

2OVER (d1 d0 --- d1 d0 d1) E

Leave a copy of the second double number on the top of stack.

2ROT (d2 d1 d0 --- d1 d0 d2) E

Rotate the 3 double numbers on the top of stack. In other words, move the third double number to the top of stack.

4. NUMBER COMPARISON

< (n1 n0 --- flag) S

Flag true if n1 less than n0. "less-than"

= (n1 n0 --- flag) S

Flag true if n1 equals n0. "equals"

> (n1 n0 --- flag) S

Flag true if n1 greater than n0. "greater-than"

0< (n --- flag) S

Flag true if n less than zero. "zero-less"

0= (n --- flag) S

Flag true if n equals zero. "zero-equals"

0> (n --- flag) S

Flag true if n greater than zero. "zero-greater"

D< (d1 d0 --- flag) S

Flag true if d1 less than d0. "d-less-than"

DU< (ud1 ud0 --- flag) E

Flag true if ud1 less than ud0. "d-u-less"

D= (d1 d0 --- flag) E

Flag true if d1 equals d0. "d-equal"

D0= (d --- flag) E

Flag true if d less than zero. "d-zero-equals"

U< (u1 u0 --- flag) S

Flag true if u1 less than u0. "u-less-than"

NOT (flag1 --- flag2) S

Invert the logical value of flag1 to give flag2. This is identical to 0=.

<> (n1 n0 --- flag) R

Flag true if n1 not equal to n0.

>= (n1 n0 --- flag)

Flag true if n1 greater than or equal to n0. "greater-or-equal"

<= (n1 n0 --- flag)

Flag true if n1 less then or equal to n0. "equal-or-less"

0<= (n --- flag)

Flag true if n less than or equal to zero. "equal-or-less-than-zero"

0>= (n --- flag)

Flag true if n greater then or equal to zero. "greater-or-equal-to-zero"

0<> (n --- flag)

Flag true if n non zero. "non-zero"

5. ARITHMETIC AND LOGICAL

+ (n1 n0 --- n2) S

Leave n2, the arithmetic sum of n1 and n0. "plus"

D+ (d1 d0 --- d2) S

Leave d2, the arithmetic sum of d1 and d2. "d-plus"

- (n1 n0 --- n2) S

Subtract n0 from n1 and leave the difference n2. "minus"

1+ (n --- n+1) S

Increment n by one. "one-plus"

1- (n --- n-1) S

Decrement n by one. "one-minus"

2+ (n --- n+2) S

Increment n by two. "two-plus"

2- (n --- n-2) S

Decrement n by two. "two-minus"

2* (n --- 2*n) R

Multiply n by two (3 times faster than n 2 *). "two-times"

2/ (n --- n/2) R

Divide n by two (6 times faster than n 2 /). "two-divide"

64* (n --- 64*n)

Multiply n by 64 (using shifts). "sixtyfour-times"

`*    ( n1 n0 --- n2 )    S`

Leave n2, the arithmetic product of n0 and n1. "times"

`/    ( n1 n0 --- n2 )    S`

Divide n1 by n0 and leave the quotient n2. n2 is rounded toward zero. "divide"

`MOD    ( n1 n0 --- n2 )    S`

Divide n1 by n0 leaving the remainder n2 which has the same sign as n1. "mod"

`/MOD    ( n1 n0 --- n3 n2 )    S`

Divide n1 by n0 and leave the remainder n3 and quotient n2. n3 has the same sign as n1. "divide-mod"

`*/MOD    ( n2 n1 n0 --- n4 n3 )    S`

Multiply n2 by n1, divide the result by n0 and leave the remainder n4 and quotient n3. The product of n1 and n2 is maintained as an intermediate 32-bit value for greater precision than the otherwise equivalent sequence

```
n2 n1 * n0 /
```

The remainder has the same sign as the intermediate product of n1 and n2. "times-divide-mod"

`*/    ( n2 n1 n0 --- n3 )    S`

Multiply n2 by n1, divide the result by n0 and leave the quotient n3. An intermediate 32-bit product is used as for */MOD. "times-divide"

`U*    ( u1 u0 --- ud2 )    S`

Perform an unsigned multiplication of u0 and u1, leaving the unsigned double number product ud2. "u-times"

`U/MOD    ( ud1 u0 --- u3 u2 )    S`

Perform the unsigned division of double number ud1 by u0, leaving the remainder u3 and quotient u2. "u-divide-mod"

`MAX    ( n1 n0 --- n2 )    S`

Leave the greater of the two numbers n0 and n1. "max"

`MIN    ( n1 n0 --- n2 )    S`

Leave the lesser of the two numbers n0 and n1. "min"

`ABS    ( n1 --- n2 )    S`

Leave n2, the absolute value of n1. "absolute"

`NEGATE    ( n --- -n )    S`

Leave the 2's complement of a number.

`DNEGATE    ( d --- -d )    S`

Leave the 2's complement of a double number.

`AND    ( n1 n0 --- n2 )    S`

Leave the bitwise logical 'and' of n1 and n0.

`OR    ( n1 n0 --- n2 )    S`

Leave the bitwise logical 'or' of n1 and n0.

`XOR    ( n1 n0 --- n2 )    S`

Leave the bitwise 'exclusive-or' of n1 and n2.

`M/MOD    ( ud1 u0 --- u4 ud3 )`

A mixed magnitude operator which leaves a double length quotient ud3 and single length remainder u4, from a double length dividend ud1 and single length divisor u0. "m-divide-mod"

M/ (**d** n0 --- n2 n1)

A mixed magnitude operator which leaves the remainder n2 and quotient n1 from a double number dividend d, and single length divisor n0. "m-divide"

M* (n1 n0 --- d)

A mixed magnitude operator which leaves the double number signed product of n1 and n0. "m-times"

U/ (ud u0 --- **u1**)

Leave the unsigned quotient u1 from the double length dividend ud and single length divisor u0. "u-divide"

DABS (d0 --- d1) E

Leave the absolute value of the double number d0. "d-abs"

DMIN (d1 d0 --- d2) E

Leave the larger of the two double numbers d0 and d1. "d-min"

DMAX (d1 d0 --- d2) E

Leave the lesser of the two double numbers d0 and d1. "d-max"

D- (d1 d0 --- d2) E

Subtract d0 from d1 and leave the difference d2. "d-minus"

6. MEMORY

@ (addr --- n) S

Leave on the stack the number contained at addr. "fetch"

! (n addr ---) S

Store the number n at addr. "store"

2@ (addr --- d) E

Leave on the stack, as a double number, the contents of the four consecutive bytes beginning at addr. "two-fetch"

2! (d addr ---) E

Store d, as a double number, in four consecutive bytes beginning at addr. "two-store"

C@ (addr --- byte) S
Leave on stack the contents of the byte at addr (with higher bits zero, in a 16-bit field). "c-fetch"

C! (n addr ---) S

Store the least significant byte of n at addr. "c-store"

+! (n addr ---) S

Add n to the 16-bit value at addr using the same convention as +. "plus-store"

+C! (n addr ---)

Add the least significant byte of n to the byte value at addr using the same convention as +. "plus-c-store"

1+! (addr ---) R

Increment the 16-bit value at addr by one.

"one-plus-store"

1-! (addr ---) R

Decrement the 16-bit value at addr by one. "one-minus-store"

? (addr ---) S

Display the number at addr in the same format as dot. "question-mark"

MOVE (addr1 addr2 n ---) S

Move the specified quantity n of 16-bit cells beginning at addr1 to memory beginning at addr2. The cell at addr1 is moved first. If n is negative or zero nothing is moved.

CMOVE (addr1 addr2 n ---) S

Move n bytes beginning at addr1 to addr2. The contents of addr1 is moved first proceeding to high memory. If n is negative or zero nothing is moved. "c-move"

<CMOVE (addr1 addr2 n ---) R

Identical to CMOVE but beginning with the byte at high memory, and proceeding toward the byte at addr1. "reverse-c-move"

FILL (addr n byte ---) S

Fill memory beginning at addr with a sequence of n copies of byte. If n is negative or zero nothing happens.

ERASE (addr n ---) R

Fill memory beginning at addr with a sequence of n zero valued bytes. If n is negative or zero nothing happens.

BLANKS (addr n ---) R

Fill memory beginning at addr with a sequence of n

ASCII blank (decimal 32) characters. If n is negative or zero nothing happens.

7. CONTROL STRUCTURES

DO (n1 n0 ---) S,I,C

Use in a colon definition in the form:

DO . . . LOOP or
DO . . . +LOOP

Begin a loop which will terminate based on control parameters. The loop index begins at n0, and terminates based on the limit n1. At LOOP or +LOOP, the index is modified by a positive or negative value. Loops beginning with DO may be nested up to a level determined by the space left in the return stack. Under normal circumstances this should be more than adequate.

The limit n1 and initial value of the index n0 are signed numbers in the range (-32768..32767).

See the definitions of LOOP and +LOOP.

LOOP S,I,C

Increment the DO loop index by one, terminating the loop if the new index is equal to or greater than the limit. Otherwise return control to the corresponding DO.

+LOOP (n ---) S,I,C

Add the signed increment n to the loop index using the convention for +, and compare the result to the limit. Return control to the corresponding DO until the new index is greater than or equal to the limit (n>0), or until it is less than or equal to the limit (n<0).

(Note: Metacrafts FORTH differs from the standard at this point. The standard requires that +LOOP be terminated when the index is strictly less than the limit when n<0.)

I (--- n) S,C

Copy the current value of the index of the immediately enclosing DO loop onto the data stack. Although this word may only be used within a loop (and thus, by default, only within a colon definition), no compile time check is made that this is the case. The effect of using the word under invalid conditions is undefined.

J (--- n) S,C

Copy the current value of the index of the DO loop enclosing the immediately enclosing loop onto the data stack. The conditions applying to the use of J are the same as for I, with the additional proviso that the result is undefined if the loop nesting is less than two.

EXIT S,C

Terminates execution of the current colon definition. EXIT may not be used within a DO loop. The effect of using EXIT under invalid conditions is undefined.

LEAVE S,C

Force termination of a DO loop at the next LOOP or +LOOP by setting the loop limit equal to the current value of the loop index. The index itself remains unchanged, and execution proceeds normally until the loop terminating word is reached. The effect of using LEAVE outside a loop is undefined.

IF (flag ---) S,I,C

Used in a colon definition in the one of the forms:-

IF . . . ELSE . . . THEN or
IF . . . THEN

If flag is true, the words between IF and ELSE (first case) or IF and THEN (second case) are executed. Thereafter execution continues with the words following THEN.

If flag is false, the words between IF and ELSE (first case) or IF and THEN (second case) are skipped, and execution continues with the words following ELSE (first case) or THEN (second case).

IF control structures can be nested

ELSE S,I,C

Executes at the end of the sequence of words executed after a true IF. It causes execution to continue with the words following the corresponding THEN.

THEN S,I,C

Marks the end of a conditional sequence of words controlled by an IF word. It is the point at which execution continues after execution of the conditional sequence.

BEGIN S,I,C

Used in a colon definition in one of the forms:

BEGIN . . . UNTIL or
BEGIN . . . WHILE . . . REPEAT or
BEGIN . . . AGAIN

BEGIN marks the start of a word sequence for conditional repetitive execution. A BEGIN-UNTIL loop will be repeated until the flag expected by UNTIL is true. A BEGIN-WHILE-REPEAT loop will be executed until the flag expected by WHILE is false, and then execution continues with the word following REPEAT.

Conditional loops can be nested.

UNTIL (flag ---) S,I,C

Marks the end of a BEGIN-UNTIL loop which is terminated by UNTIL if flag is true, otherwise control returns to the first word following the corresponding BEGIN.

WHILE (flag ---) S,C,I

Marks the conditional control point in a BEGIN-WHILE-REPEAT loop. If flag is true the body of the loop between WHILE and REPEAT is executed, otherwise the loop terminates and control passes to the word following the corresponding REPEAT.

REPEAT S,C,I

Marks the end of a BEGIN-WHILE-REPEAT loop. It is executed as the last word in the body of the loop and it always passes control to the word following the corresponding BEGIN.

AGAIN R,I,C

Marks the end of a BEGIN-AGAIN loop. It always passes control to the word following the corresponding BEGIN. The only way to exit from a BEGIN-AGAIN loop is by executing one of EXIT, QUIT, ABORT or ABORT". Note that leaving a BEGIN-AGAIN loop also means leaving the current colon definition!

CASE (n --- n) I,C

Used in a colon definition in the form:

CASE...OF...ENDOF...OF...ENDOF...DEFAULT...END-CASE

CASE marks the beginning of a multiple choice control structure. The value n is used within the CASE control structure to determine which, if any, of the alternative word sequences to execute.

CASE structures can be nested within one another.

OF (n1 n0 --- (n1)) I,C

Marks the start of one of the alternative choices in a CASE structure. The value n1 is assumed to be the value on stack on entry to the CASE structure. OF compares it with the value n0 and, if the two are equal it removes both values and passes control to the next word. If n0 is not equal to n1, n0 is removed from the stack and control passes to the word following the corresponding ENDOF.

ENDOF I,C

Marks the end of one of the alternative choices in a CASE structure. It is executed as the last word of such an alternative and it passes control to the word following the corresponding END-CASE.

DEFAULT (n --- n) I,C

It is possible to arrange that the last alternative in a CASE structure is always executed if non of its predecessors has been selected for execution. DEFAULT marks the start of this so-called default case, and END-CASE marks the end of it. The value n is the value that was on stack at the start of the CASE structure. On completion of the default action control passes to the word following the corresponding END-CASE.

DEFAULT can be omitted if no default action is required.

END-CASE ((n) ---) I,C

Marks the end of a CASE control structure. In the absence of DEFAULT it is the word to be executed when, during run time, non of the CASE alternatives has been selected for execution. In that case, the value n on stack when the case construct was entered is removed and discarded.

EXECUTE (addr ---) S

Execute the word whose compilation address is on the stack.

<BRANCH I,C

Unconditional backward branch. At compile time it leaves the compilation address of the word BRANCH in the definition being compiled. At execution time an unconditional branch is made to the relative branch address left by <RESOLVE. Must be used in conjunction with <MARK and <RESOLVE.

<MARK (--- addr) C

Leaves the destination address of a backward branch on stack. Must be used at compile time in conjunction with <RESOLVE and one of <BRANCH or ?<BRANCH.

<RESOLVE (addr ---) C

Used at compile time at the source of a backward branch after either <BRANCH or ?<BRANCH. It expects to find the destination address of the branch operation on stack after being left there by a previous <MARK. It plants a relative branch address in the definition currently being compiled.

?<BRANCH (flag ---) I,C

Conditional backward branch. At compile time it leaves the compilation address of the word OBRANCH in the definition being compiled. At execution time a flag on stack is examined to see if a branch should be taken or not. If the flag is true, control passes to the relative branch address left by <RESOLVE, otherwise execution continues at the point following the branch.

>BRANCH I,C

Unconditional forward branch. Action same as for <BRANCH. Must be used in conjunction with >MARK and >RESOLVE.

>MARK (--- addr) C

Marks the source of a forward branch. Used either after >BRANCH or ?>BRANCH. It compiles a hole into the dictionary for the relative branch address which is subsequently placed there by >RESOLVE. It leaves the address ofthe hole on stack.

>RESOLVE (addr ---) C

Used at compile time at the destination of a forward branch. It computes a relative branch address to the current point in the definition and places this in the hole whose address has been left on stack by >MARK.

?>BRANCH (flag ---) I,C

Conditional forward branch. At compile time it leaves the compilation address of the word OBRANCH in the definition being compiled. At execution time a flag on stack is examined to see if a branch should be taken or not. If the flag is true control passes to the relative branch address left by >RESOLVE, otherwise execution continues at the point following the branch.

?PAIRS (n1 n0 ---)

Issues the error message STRUCTURE? if n0 does not equal n1, and aborts execution. It is used to test for matching control construct words.

(OF) (n1 n0 ---)

This word is the run-time handler for OF in a CASE construct. It compares the case value n1 with an alternative n0 and takes appropriate action. See OF.

(DO) (n1 n0 ---)

This word is the run-time handler for DO. It initialises the loop control parameters using n1 and n0. See DO.

(+LOOP) (n ---)

This word is the run-time handler for +LOOP. It increments th loop index by n and tests for termination. See +LOOP.

(LOOP)

This word is the run-time handler for LOOP. It increments the loop index by one and tests for termination. See LOOP.

BRANCH

This word is the run-time handler for performing

unconditional branches. The 16-bit value following it in the dictionary (the relative branch address) is added to the current interpreter pointer to arrive at a new value for the interpreter pointer. Execution continues at the new position.

OBRANCH (flag ---)

This word is the run-time handler for performing conditional branches. If flag is true then the 16-bit value following it in the dictionary (the relative branch address) is added to the current interpreter pointer to arrive at a new execution address. If flag is false then the interpreter pointer is advanced over the relative branch address.

8. TERMINAL INPUT-OUTPUT

CR S

Cause a carriage-return and line-feed to occur at the current output device. "c-r"

?CR (n ---)

Cause a carriage-return and line-feed to occur at the terminal if less than n+2 character positions remain on the current line, otherwise output a blank character. "query-c-r"

Note that this word will not function correctly if the terminal does not support cursor addressing.

EMIT (char ---) S,V

An execution vector which transmits char to the current output device. The system initialises EMIT to use (EMIT) as output handler.

(EMIT) (char ---)

Transmit char to the terminal device. "bracket-emit"

SPACE

Transmit an ASCII blank to the current output device.

SPACES (n ---) S

Transmit n blanks to the current output device. Nothing happens if n is less than or equal to zero.

TYPE (addr n ---) S

Transmit n characters beginning at addr to the current output device. Nothing happens if n is less than or equal to zero.

COUNT (addr --- addr+1 n) S

Leave the address addr+1 and the character count of a text string beginning at addr. The first byte at addr must contain the string length n, which must be in the range (0..255).

-TRAILING (addr n1 --- addr n2) S

Adjust the character count n1 of text at addr to exclude trailing blanks, i.e., the characters at addr+n2 to addr+n1-1 are blanks. If n1 is zero or negative, nothing happens. "dash-trailing"

KEY (--- char) S,V

An execution vector which leaves the ASCII value of the next available character from the current input device. The system initialises KEY to use the input handler (KEY).

(KEY) (--- char)

Accepts the next available char from the terminal device. "bracket-key"

EXPECT (addr n ---) S

Transfer characters from the input device to memory beginning at addr, upward, until a "return" character or n characters has been read. Nothing happens if n is less than or equal to zero. Two null characters are appended to the input text, so that an input buffer of n+2 characters must be available at addr.

QUERY S

Accept input of up to 80 characters, or until a "return" character, from the input device into the terminal input buffer whose address is stored in TIB. Both WORD and ENCLOSE can be used to accept text from this buffer by setting >IN and BLK to zero.

WORD (char --- addr) S

Receive characters from the input stream according to the delimiter char and place the characters in a string beginning at addr+1. The character count is placed in the byte position at addr. An arror condition results if char is ASCII null or if the count exceeds 255. Initial occurrences of char in the input stream are ignored. If char appears in the input stream as a terminating character, it is appended to the string but not included in the count. If the input stream is exhausted before char is encountered as a terminating character, the terminating character null is appended instead of char. A zero length will result if the input stream is exhausted when WORD is called.

Metacrafts implementation of WORD always leaves the input string at HERE.

ENCLOSE (char1 char2 --- addr len)

Scan the input stream from the character position specified by >IN. Skip instances of char1 and leave addr, the address of the first character not equal to char1, on stack. Continue scanning and begin counting characters scanned until either an instance of char2 is found or input is exhausted. Leave len, the count of scanned characters on stack. This is the length of the ENCLOSE'd string. An error condition results if the length of the ENCLOSE'd string exceeds 255 characters. If end of input is detected before the scan of leading delimiters (char1) is complete, len is set to zero. The length of the ENCLOSE'd string will be zero if end of input or a trailing delimiter (char2) is found immediately following the leading delimiter scan. >IN is advanced to point to the first character following the last character scanned unless an error is detected.

CHARS (char n ---)

Transmit n copies of char to the output device. Nothing happens if n is less than or equal to zero.

C/T (--- addr) U

Leave the address of a cell that contains the line length of the operator's terminal device. "c-slash-t"

TIB (--- addr) U

Leave the address of a cell that contains the address of the first byte of the terminal input buffer.

>IN (--- addr) U

Leave the address of a cell that contains the current character offset within the input stream. When BLK contains zero, the input stream is taken from the terminal input buffer. "to-in"

BELL R

Transmit a bell character (ASCII 7) to the current output device.

PAGE R,V

An execution vector that causes the terminal screen to be cleared and the cursor to be moved to the first character position on the top line. If the output device is a printer, output is moved to top of form. The system initialises PAGE to use (PAGE) to carry out its function.

(PAGE)

Causes the 40-column Apple display to be cleared and the cursor to be moved to the first character position on the top line.

<PAGE>

Causes an 80-column VIDEOTERM controlled display to be cleared and the cursor to be moved to the first character position on the top line.

@CURSOR (--- row col) V

An execution vector that leaves the current cursor position on stack. The system initialises @CURSOR to use (@CURSOR) to get the cursor row and column positions.

(@CURSOR) (--- row col)

Reads the current cursor position when a 40-column Apple display is being used.

<@CURSOR> (--- row col)

Reads the current cursor position when an 80-column VIDEOTERM display controller is being used.

!CURSOR (row col ---) V

An execution vector that resets the cursor position to the new row and col positions on stack. The system initialises !CURSOR to use (!CURSOR) to reposition the cursor. No check is made to ensure that the new position is valid.

(!CURSOR) (row col ---)

Repositions the cursor on a 40-column Apple display to row and col.

<!CURSOR> (row col ---)

Repositions the cursor on an 80-column VIDEOTERM controlled display to row and col.

.OK V

An execution vector that the text interpreter invokes when it has finished processing the terminal input buffer contents. The system initialises .OK to use (.OK).

(.OK)

Output the string OK on a 40-column Apple display.

<.OK>

Output the string ok on an 80-column VIDEOTERM controlled display.

FLASH

Select flashing character mode for the Apple display.

INVERSE

Select inverse character mode for the Apple display.

NORMAL

Select normal character mode for the Apple display.

?TERMINAL (--- flag)

Leave a true flag if a key on the terminal keyboard has been pressed, otherwise leave false. Note that this word does not read the character from the keyboard.

PAUSE

Halt execution until a key on the keyboard is pressed.

?WAIT

Check if a key on the keyboard has been pressed. If so PAUSE, otherwise do nothing.

80-COLS

Switch on a VIDEOTERM display controller in slot 3, and reset the terminal execution vectors PAGE, @CURSOR, !CURSOR and .OK.

CLREOL V

An execution vector that clears the current display line from the cursor position.

(CLREOL)

Initial value of CLREOL. Use with a 40 column display.

<CLREOL>

Assign to CLREOL when using an 80-column videoterm card.

CLREOP V

An execution vector that clears the display screen from the cursor position.

(CLREOP)

Initial value of CLREOP. Use with a 40 column display.

<CLREOP>

Assign to CLREOP when using an 80-column videoterm card.

9. NUMERIC CONVERSION

BASE (--- addr) S,U

Leave the address of a cell containing the current input-output numeric conversion base. The value must be in the range (2..70).

DECIMAL S

Set the input-output numeric conversion base to ten.

HEX R

Set the input-output numeric conversion base to sixteen.

. (n ---) S

Display n converted according to BASE in a free-field format with one trailing blank. Leading zeros are suppressed and a minus sign appears before a negative number. "dot"

U. (u ---) S

Display u converted according to BASE as an unsigned number in a free-field format with one trailing blank. "u-dot"

.R (n1 n0 ---) R

Display n1 right aligned and converted according to BASE in a field of n0 characters. Leading zeros are suppressed, and a minus sign appears before a negative number. If n0 is less than one or less than the minimum field width necessary to display the complete number, the number will be displayed with no leading blanks. "dot-r"

D. (d ---)

Display a signed double number converted according to BASE in a free-field format with one trailing blank. Leading zeros are suppressed, and a minus sign appears before a negative number. "d-dot"

`D.R          ( d n --- )`

Display a double number right aligned and converted according to BASE in a field of n characters. Leading zeros are suppressed, and a minus sign appears before a negative number. If n0 is less than one or less than the minimum field width necessary to display the complete number, the number will be displayed with no leading blanks. "d-dot-r"

`<#                                   S`

Initialize pictured numeric output. The words:

`<#   #   #S   HOLD   SIGN   #>`

can be used to specify the conversion of a double-precision number into an ASCII character string converted according to BASE. "less-sharp"

`#            ( ud1 --- ud2 )          S`

Generate from an unsigned double number ud1, the next significant digit as an ASCII character which is placed in an output string. Successive invokations of # generate digits of the number moving from low to high significance. The result ud2 is the quotient after division by BASE. # must only be used between <# and #>. The effect of using it elsewhere is undefined. "sharp"

`#S           ( ud --- 0 0 )           S`

Convert all digits of an unsigned double number ud according to the convention for # until the quotient left by # is zero. A single zero is added to the output string if the number was initially zero. As with #, #S must only be used between <# and #>. "sharp-s"

`#>           ( ud --- addr n )        S`

End pictured numeric output conversion. Drop ud, leaving the text address and character count of the output string suitable for output by TYPE.

"sharp-greater"

`SIGN         ( n --- )                S`

Insert a minus sign (ASCII -) into the pictured numeric output string if n is negative.

`HOLD         ( char --- )             S`

Insert char into the pictured numeric output string. As with #, it must only be used between <# and #>.

`HLD          ( --- addr )             U`

Leave the address of a cell that contains the address of the last character added to the pictured numeric output string.
Its value is only meaningful between calls of <# and #>. "h-l-d"

`NUMBER       ( addr --- d )           R,V`

An execution vector which performs numeric input conversion. The system initialises it to use (NUMBER) to perform the conversion. See (NUMBER) for further details.

`(NUMBER)     ( addr --- d )`

Convert the text string at addr to a signed 32-bit integer using the current value of BASE. If conversion is not possible an error condition exists. If a decimal point is encountered in the text, its position is left in DPL.

`DPL          ( --- addr )             U,R`

Leave the address of a cell which contains the number of digits encountered to the right of a decimal point during conversion of the last number input. If no decimal point was encountered its value is -1. "d-p-l"

`DIGIT        ( char n0 --- (n1) flag )`

Convert the ASCII character char to its binary equivalent n1 using base n0. If the conversion using base n0 is possible leave a true flag, otherwise leave only a false flag.

CONVERT (d1 addr1 --- d2 addr2) S

Convert to the equivalent stack number the text beginning at addr1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. addr2 is the address of the first non-convertible character.

S->D (n --- d)

Convert the 16-bit number n into its signed double length equivalent.

10. MASS STORAGE INPUT/OUTPUT

LIST (n ---) S

List the ASCII symbolic contents of block n on the current output device, setting SCR to n. The value of n is assumed to be a valid block number on the currently selected mass storage device. The contents of the block are displayed as 16 rows of 64 characters. Line truncation takes place if the current output device is the 40-column Apple display. The display is cleared before the screen is listed.

?LOADING

Abort if the current input stream is the terminal.

BLK (--- addr) U

Leave the address of a variable containing the number of the block currently being interpreted. Zero means that input is being accepted from the terminal input buffer.

LOAD (n ---) S

Begin interpretation of block n by making it the current input stream after first preserving the values of >IN and BLK for the current input stream. If interpretation is not terminated explicitly, it will be terminated when the input stream is exhausted. Control then returns to the input stream containing LOAD, determined by the preserved values of >IN and BLK.

SCR (--- addr) S,U

Leave the address of a cell containing the number of the screen most recently listed. "s-c-r"

BLOCK (n --- addr) S

Leave the address of the first byte of a buffer containing block n. If the block is not already in a memory buffer, it is transferred from the currently selected mass storage device into whichever memory buffer has been least recently accessed. If the block occupying that buffer has been UPDATE'd, it is rewritten onto the correct mass storage device before block n is read into the buffer. If correct mass storage read/write is not possible, an error condition exists. Subsequent calls of BLOCK with different values of n will eventually re-use the block buffer at addr. For this reason, any later access to block n should be via another call of BLOCK.

UPDATE S

Mark the block most recently referenced through BUFFER or BLOCK as modified. The block will subsequently be automatically transferred to mass storage should its memory buffer be needed for storage of a different block, or upon execution of SAVE-BUFFERS.

BUFFER (n --- addr) S

Leave the address of the first byte of the least recently accessed block buffer, and mark this buffer as allocated to block n on the currently selected mass storage device without actually transferring the block from the device. If the previous contents of the buffer have been marked as modified, it is written to mass storage. If correct writing to mass storage is not possible, an error condition exists. No check is made to see if one of the other buffers already contains block n.

SAVE-BUFFERS S

Write all blocks that have been UPDATE'd to mass storage. An error condition exists if successful writing of all modified blocks is not possible.

EMPTY-BUFFERS S

Mark all block buffers as empty and do not write UPDATE'd blocks to mass storage.

--> R,I

Continue interpretation of the next block in sequence. It can be used within a colon definition that crosses a block boundary.

SAVE (n +++)

Read the next word name from the input stream, locate its definition in the dictionary and save it and all following definitions on the currently selected mass storage device starting at block n. Use as many contiguous blocks as it takes to store the definitions. Display the number of the last block allocated.

RESTORE (n0 n1 ---)

Transfer the word definitions saved by a single SAVE command in blocks n0 to n1 back to the point in the dictionary from which they originate. All definitions currently in the dictionary before this point must be identical to those there at the time the restored definitions were saved. All definitions following this point are overwritten. All vocabularies are restored to the state they were in at the time the dictionary was saved, and the vocabulary stacks are reset by TRUNK. n0 is the number of the first block used to save the definitions, and n1 is the number of the last block reported by SAVE.

FORMAT V

An execution vector that formats mass storage media. It is set up to call (FORMAT).

(FORMAT)

Format the diskette in the currently selected Disk II drive. This word should be used with great care to avoid destroying the contents of important diskettes.

R/W (addr n ---) V

An execution vector that either reads (n=1) or writes (n=2) a block to mass storage. 'addr' points at the system control information of the block buffer to be used in the transfer. The system initialises R/W to use (R/W) as handler for mass storage devices.

(R/W) (addr n ---)

Read (n=1) a mass storage block or write (n=2) a mass storage block using the block buffer at addr+5, and buffer control information at addr. If a transfer is unsuccessful an error condition exists.

+BUFFERS (n ---)

Increase the number of block buffers by n. An error condition exists if insufficient memory is available for the buffers. If n is zero or negative the effect is undefined.

-BUFFERS (n ---)

Reduce the number of block buffers by n. An error condition exists if the number of remaining buffers is less than or equal to two. The contents of the released buffers is lost. If n is zero or negative the outcome is undefined.

+1BUFFER

Increase the number of block buffers by one. No check is made to ensure that sufficient memory is available. See ?BUFFER.

-1BUFFER

Decrease the number of block buffers by one. An error condition exists if the number of remaining buffers is less than or equal to two.

?BUFFER (--- flag)

Leave a true flag if sufficient store is available for increasing the number of block buffers by one.

USE (--- addr) U

Leave the address of a cell containing the address of the control information of the next block buffer to be allocated by BUFFER.

PREV (--- addr) U

Leave the address of a cell containing the address of the control information of the last block buffer allocated by BUFFER.

C/BUF (--- n)

A constant whose value is the number of bytes in a block buffer.

C/L (--- n)

A constant whose value is the number of characters in a standard screen line.

#DRIVES (--- addr) U

Leave the address of a cell containing the number of mass storage devices connected to the system. The default value is two.

DR (--- addr) U

Leave the address of a cell containing the internal device number of the currently selected mass storage device.

DR1

Select mass storage drive 1.

DR2

Select mass storage device 2.

.LINE (n1 n0 ---)

Transmit line n1 of screen n0 to the current output device after removing trailing blanks. The line is truncated to the width of the output device if necessary. The effect is undefined if n1 is not a valid line number in the range (0..15).

(LINE)   (n1 n0 --- addr n2)

Leave the byte address of the first character of line n1 of screen n0, together with the maximum number n2 of characters that can be sent to the current output device without the line overflowing.

(LIST)   (n ---)

Identical to LIST with the exception that it does not issue a PAGE request.

11. DEFINING WORDS

:   S

A defining word used in the form:

: <name> . . . ;

Create a dictionary entry for <name> in the vocabulary on top of the CURRENT vocabulary stack, and switch on compile mode. Any word thus defined is known as a "colon definition". The compilation addresses of subsequent words from the input stream which are not immediate words are stored in the dictionary as part of the definition of <name>, to be executed whenever <name> is executed. Immediate words are executed as encountered. Compilation of <name> is terminated by ; .

If a word used in the definition of <name> is not found after searching all the vocabularies in the CONTEXT stack, beginning with the vocabulary on the top of the stack, conversion and compilation of a literal is attempted with regard to the current number base. If that fails, an error condition exists.

If <name> already exists in the vocabulary on top of the CURRENT stack, then the warning message:

<name> NOT UNIQUE

is output to the operator's display. "colon"

;   S,I,C

Terminate a colon-definition and stop compilation. If compiling from mass storage and the input stream is exhausted before encountering ;, an error condition exists. If the data stack does not have the same depth that it had when compilation of the definition began, an error condition exists. (See also -->) "semi-colon"

VARIABLE   S

A defining word executed in the form:

VARIABLE <name>

to create a dictionary entry for <name> and allot two bytes of storage in the parameter field. The application is responsible for initializing the stored value of the variable. When <name> is later executed it will leave the address of the first of the two bytes on the stack.

2VARIABLE E

A defining word executed in the form:

2VARIABLE <name>

to create a dictionary entry for <name> and allot 4 bytes of storage in the parameter field. The application is responsible for initializing the stored value of the variable. When <name> is later executed it will leave the address of the first of the four bytes on the stack. "two-variable"

VARIABLE$ (n +++)

A defining word executed in the form:

n VARIABLE$ <name>

to create a dictionary entry for <name> and allot storagefor a string of up to n characters in the parameter field. The application is responsible for initializing the stored value of the variable. When <name> is later executed it will leave the address of the string on the stack. "variable-dollar"

CONSTANT (n +++) S

A defining word executed in the form:

n CONSTANT <name>

to create a dictionary entry for <name>, leaving n in its parameter field. When <name> is later executed n will be left on the data stack.

2CONSTANT (d +++) E

A defining word executed in the form:

d 2CONSTANT <name>

to create a dictionary entry for <name>, leaving d in its parameter field. When <name> is later executed d will be left on the data stack. "two-constant"

CONSTANT$ (addr +++)

A defining word executed in the form:

addr CONSTANT$ <name>

to create a dictionary entry for <name>, leaving the string located at addr in its parameter field. When <name> is later executed the string will be moved to PAD, and the address of PAD left on the stack. "constant-dollar"

ARRAY (n0 n1 +++)

A defining word executed in the form:

n0 n1 ARRAY <name>

to create a dictionary entry for <name>, and allot sufficient storage for an array of (n1-n0+1) cells in the parameter field. The application is responsible for initializing the elements of the array.

<name> is executed in the form:

n <name>

to leave the address of the array element with index n on the stack. No array bound checking is performed.

CARRAY (n0 n1 +++)

A defining word executed in the form:

n0 n1 CARRAY <name>

to create a dictionary entry for <name>, and allot storage for an array of (n1-n0+1) bytes in the parameter field. The application is responsible for initializing the elements of the array.

<name> is executed in the form:

n <name>

to leave the address of the array element with index n on the stack. No array bound checking is performed.

ARRAY$ (n0 n1 n2 +++)

A defining word executed in the form:

n0 n1 n2 ARRAY$ <name>

to create a dictionary entry for <name>, and allot storage for an array of n1-n0+1 strings of n2 characters each in the parameter field. The application is responsible for initialising the elements of the array. <name> is executed in the form:

n <name>

to leave the address of the array element with index n on the stack. No array bound checking is performed. "array-dollar"

EXECUTE: (n +++)

A defining word executed in the form:

n EXECUTE: <name>

to create a dictionary entry for an execution vector with the name <name> and leave n in its parameter field. When <name> is later executed the effect is to

execute the word whose compilation address is stored in cell number n in the user area. See also IS, WAS and USER. "execute-colon"

USER (n +++) R

A defining word used in the form:

n USER <name>

which creates a user variable <name>. The value n is the cell offset within the user area where the value for <name> is stored. Execution of <name> leaves the address of the cell allocated in the user area. The value of n must be in the range (0..127). Note that cells (0..37) have already been allocated a purpose within Metacrafts FORTH, and that cells (38..63) are reserved for future expansion.

CREATE S,V

An execution vector and defining word used in the form:

CREATE <name>

to create a dictionary entry for <name>, without allocating any space for a parameter field. ?STORE is invoked to ensure that at least 512 bytes of dictionary space are available. The new definition is attached to the front of the list of words belonging to the vocabulary on top of the CURRENT vocabulary stack. The warning message:

<name> NOT UNIQUE

is sent to the operator's terminal if <name> already belongs to this vocabulary. In this case, the new definition supersedes all previous definitions.

When <name> is subsequently executed, the address of the first byte of <name>'s eventual parameter field is left on stack.

(CREATE)

A defining word used to initialize the execution vector CREATE. See CREATE for further details. "bracket-create"

DOES> S,I,C

Define the run-time action of a word created by a high-level defining word. Used in the form:

: <name> . . . CREATE . . . DOES> . . . ;

marks the termination of the defining part of the defining word <name> and the beginning of the run-time action of words later defined by <name>. On execution of a word <namex> defined by <name>, the sequence of words between DOES> and ; will be executed, with the address of <namex>'s parameter field on the stack. "does"

VOCABULARY S

A defining word executed in the form:

VOCABULARY <name>

to create, in the vocabulary on top of the CURRENT vocabulary stack, a dictionary entry for <name> which specifies a new ordered list of word definitions. Subsequent execution of <name> will cause the vocabulary which <name> denotes to be pushed onto the top of the CONTEXT vocabulary stack.

Note that if the CONTEXT stack only contains FORTH at the bottom, and some other application vocabulary on top of it, then Metacrafts' vocabulary structure is functionally equivalent to the requirements of the 79-Standard.

---

12. VOCABULARIES

(FIND) (addr1 addr0 --- addr)

Search the vocabulary chain beginning with the word whose lfa is addr0 until the word whose name is at addr1 is found. Leave the lfa of the definition whose name is the same as the one given. If no match is found, leave zero. A match exists if the lengths of the two names are the same and the characters stored in the definition are the same as the characters beginning the given name. Lower case characters are treated as upper case. "bracket-find"

-FIND (addr --- (addr) flag)

Search the vocabulary on top of the context stack until the definition whose name matches the name at addr is found. If a match is found, then leave the cfa of the matching word and a false flag, otherwise just leave a true flag. "minus-find"

?UNIQUE (+++ addr)

Search the vocabulary on the top of the current stack until a definition is found whose name appears next in the input stream. If a match is found, leave the cfa of the definition on the stack, otherwise leave a zero. "query-unique"

CONTEXT (--- addr) S,U

Leave the address of a variable specifying the vocabulary on top of the context stack. This is the starting point for dictionary searches during interpretation of the input stream.

CURRENT (--- addr) S,U

Leave the address of a variable specifying the vocabulary on top of the current stack. New definitions will be added to this vocabulary.

DEFINITIONS S

Add the vocabulary referenced by CONTEXT to the top of the current stack and change CURRENT to reference it. Subsequent definitions will be created in the vocabulary that is on top of the current stack.

DEFINED

Remove the top vocabulary from the context and current stacks, and reset CONTEXT and CURRENT to reference the new top of stack vocabularies.

FIND (+++ addr) S

Leave the cfa of the word whose name follows in the input stream. If the name cannot be found after a search of all the vocabularies in the context stack, beginning with the one referenced by CONTEXT, then leave zero.

FORGET S

Delete from the dictionary the definition of the word whose name follows in the input stream, together with any words defined since this word was added. If any vocabulary words are forgotten as a result of forgetting the word, then reset the current and context stacks using TRUNK. An error condition exists if the word is either not in a vocabulary on the context stack, or is located in the protected dictionary below FENCE.

Note: the 79-Standard requires that the word to be forgotten must belong to the current vocabulary. Metacrafts FORTH does not check this.

FORTH S,I

The name of the primary vocabulary. It is used to initialise the current and context vocabulary stacks during startup, and whenever ABORT or TRUNK are called.

PRUNE (addr ---)

Remove the vocabulary at the top of the stack referenced by the contents of addr, and reset addr to reference the new top of stack. addr should be either CONTEXT or CURRENT.

TRUNK

Clear both vocabulary stacks and reset them to contain FORTH. Reset CONTEXT and CURRENT to reference the new tops of the stacks.

VLIST R

Display the names of the words contained in the vocabulary on top of the context stack.

VOC-LINK (addr ---) U

Leave the address of a variable containing the address of a field in the most recent vocabulary definition. All vocabulary definitions are linked by these fields to enable FORGET, SAVE and RESTORE to function correctly.

' (+++ addr) S,I

Used in the form

' <name>

If executing, leave the pfa of the next word accepted from the input stream. If compiling, compile this address as a literal. Later execution will place the value on the stack. An error condition exists if <name> is not found in a vocabulary on the context stack. "tick"

13. COMPILER

!CSP

Save the stack position in CSP. Used as part of the compiler's error checking mechanism. "store-c-s-p"

?CSP

Abort with the message DEFINITION INCOMPLETE if the value of CSP is not the same as the stack position. "query-c-s-p"

CSP (--- addr) U

Leave the address of a variable used as a temporary store for the stack position during compilation.

, (n ---) S

Allot two bytes in the dictionary and store n there. "comma"

-TRAVERSE (addr0 --- addr1)

Scan towards low memory beginning with the byte at addr0-1 and stop when a byte is found with its high order bit set to 1. Leave the address of this byte on stack. "minus-traverse"

TRAVERSE (addr0 --- addr1)

Scan towards high memory beginning with the byte at addr0+1 and stop when a byte is found with its high order bit set to 1. Leave the address of this byte on stack.

;S R

This is the run time word compiled for ; at the end of a colon definition. It can also be used to stop interpretation of a screen. "semi-s"

?COMP

Abort compilation with the message COMPILATION ONLY if not in compile mode. "query-comp"

?EXEC

Abort compilation with the message EXECUTION ONLY if in compile mode. "query-exec"

ALLOT (n ---) S

Add n bytes to the parameter field of the most recently defined word. Can also be used to allocate n bytes of dictionary space at HERE.

ASCII C,I,R

Use in the form:

ASCII <char>

Get the next word <char> from the input stream and compile it as a character literal in the current definition. Abort with the message NOT ASCII CHAR if <char> consists of more than a single character. Leave the character on stack at run-time.

C, (char ---) R

Allot a single byte in the dictionary and store char there. "c-comma"

CFA (lfa --- cfa)

Convert a word's lfa into its cfa. "c-f-a"

LFA (cfa --- lfa)

Convert a word's cfa into its lfa. "l-f-a"

NFA (lfa --- nfa)

Convert a word's lfa into its nfa.

PFA (cfa --- pfa)

Convert a word's cfa into its pfa. Does not apply to CODE definitions. "p-f-a"

CLIT (--- char)

Load the literal byte at IP+2 onto the stack and increment IP so that interpretation continues beyond the literal. "c-lit"

CLITERAL (char ---) I

If compiling, then compile the stack value char as a character literal to be left on the stack at run-time by CLIT.

DLITERAL (d ---) I

If compiling, then compile the stack value d as a double number literal which will be loaded onto the stack at run-time.

DP (--- addr) U

Leave the address of a variable which contains the address of the next free memory byte above the dictionary. "d-p"

FENCE (--- addr) U

Leave the address of a variable which contains a dictionary address below which words cannot be removed by FORGET. FENCE is initialised by the code in block 13.

HERE (--- addr) S

Leave the address of the next available byte in the dictionary.

ID. (nfa ---)

Display the name at nfa. Note that it is incorrect to use COUNT and TYPE to display the contents of the name field because of the effect of width and the presence of non-count bits in the length byte. "id-dot"

IMMEDIATE S

Mark the most recently defined word as one which will be

executed when encountered in a definition at compile time rather than compiled.

LATEST (--- lfa)

Leave the lfa of the most recently defined word.

LIT (--- n)

Load the 16-bits at IP+2 onto the stack and increment IP so that interpretation can continue beyond the literal. The byte at IP+2 becomes the low-byte on stack.

LITERAL (n ---) S,I

If compiling, then compile the stack value n as a 16-bit literal which will be loaded onto the stack at run-time by LIT.

RECURSE C,I

Compile the cfa of the definition which contains the call. Use for compiling recursive procedure calls.

SMUDGE

Toggle the smudge bit in the name field of the latest definition. When the smudge bit is set, the word will be ignored during dictionary searches.

STATE (--- addr) U

Leave the address of a variable which contains the current compilation state: zero means the compiler is switched off, hexadecimal C0 means it is switched on.

TOGGLE (addr byte ---)

Complement the byte at addr with the byte on stack.

TOP (--- addr) U

Leave the address of a variable which contains the address of the top of the memory space available for dictionary use.

WIDTH (--- addr) U

Leave the address of a variable containing the maximum number of characters to be saved in the name field of new definitions. It must have a value in the range 1-31. The default value set up by the system is 31.

[I,S

Switch off the compiler and start executing text in the input stream. "left-bracket"

[COMPILE] S,I,C

Used in a colon definition in the form

[COMPILE] <name>

Force compilation of the following word called <name>. This allows compilation of an IMMEDIATE word which would otherwise have been executed. "bracket-compile"

] S

Switch on the compiler and start compiling the text in the input stream. "right-bracket"

14. MISCELLANEOUS

(S,I

Used in the form

(this is a comment)

Accept and ignore comment characters from the input stream, until the next right parenthesis. The left parenthesis is a FORTH word, and so must be followed by at least one space. It may be freely used while executing or compiling text. If more than 255 characters are read, or the input stream is exhausted before a right parenthesis is reached, an error condition exists. Note that the right paranthesis is not a FORTH word, it is simply a delimiter. "paren"

(ABORT") (flag ---)

Output the string at IP+2 and call ABORT if flag is true, otherwise advance IP past the string. "bracket-abort-quote"

(IS) (addr1 addr0 ---)

Store addr1, which must be a cfa, in the execution vector indexed by the contents of addr0. "bracket-is"

(QUIT)

Clear the return stack, switch off the compiler, select the terminal as input stream and pass control to the text interpreter. "bracket-quit"

(WAS) (addr ---)

Leave on stack the cfa which is the value of the execution vector whose index is held in addr. "bracket-was"

79-STANDARD S

Do nothing.

<SYSVOL (--- n flag)

Leave the number n of the currently selected drive on stack, and select the boot drive. Check to see if the system disc is in the boot drive and leave a false flag if it is. If not, ask the operator to load it, PAUSE, and leave a true flag on stack.

?CONTINUE (--- flag)

Ask the operator to press return is he wants to the application to continue, or to press ESC if he wants to quit. Wait for one of the two keys to be pressed, and leave a true flag if he presses return, false otherwise. "query-return"

?STACK

Abort with the message STACK FULL if the stack pointer lies beyond its bottom limit (hex 4A), and abort with the message STACK EMPTY if the stack pointer lies beyond its upper limit (hex FC) or if the cell at FC is found to be non-zero. Called by INTERPRET after each interpreted word. "query-stack"

?STORE

Abort with the message DICTIONARY FULL if the distance between the contents of DP and TOP is less than 512 bytes. The reason that the gap is so large is that incoming words are stored at HERE, and can be up to 256 bytes long (including count byte), and also PAD is 256 bytes beyond HERE and must be large enough to accomodate the longest string. "query-store"

ABORT S

Clear the data and return stacks, set the number base to 10, reset video mode to normal, call TRUNK and pass control to QUIT.

ABORT" (flag +++) C,R,I

Use in a colon definition in the form

ABORT" this is an error message"

Compile the text up to the terminating " as a string literal in the current definition. At run-time use (ABORT") to output the message if the flag is true. "abort-quote"

BL (--- char) R

Leave the ASCII code for blank space on stack. "blank"

BOOT

Cold start the system by reloading from the boot drive.

BOOT3

A double number variable used to mark the start of the main system overlay. Its value is of no significance once the overlay has been loaded.

CSW (--- addr)

Leave the address of the monitor's character output vector. "c-s-w"

FALSE (--- flag)

Leave the value which denotes logical false.

INTERPRET R

Begin interpretation at the character indexed by >IN relative to the block whose number is contained in BLK, continuing until the input stream is exhausted. If BLK contains zero, interpret characters from the terminal input buffer. The end of the input stream is marked by two bytes containing zero.

IS (addr +++) I

Use in the form

IS <name>

If compiling, compile the pfa of <name> as a literal followed by a call to (IS), otherwise put the pfa of <name> on stack and call (IS). Use to assign a value to the execution vector called <name>.

KSW (--- addr)

Leave the address of the monitor's character input vector. "k-s-w"

PAD (--- addr) S

Leave the address of a scratch area used to hold character strings for intermediate processing. It is located above the end of the dictionary. If less than 256 bytes are available between PAD and the top of the dictionary, abort with the message DICTIONARY FULL.

QUIT S,V

An execution vector which is initialised with the address of (QUIT).

RP!

Initialise the return stack pointer. "r-p-store"

SO (--- addr) U

Leave the address of a variable which contains the address of the base of the data stack. "s-zero"

SERIAL# (--- n)

Leave the serial number of the system disc. "serial number"

SP!

Initialise the data stack pointer with the value of SO. "s-p-store"

SP@ (--- n) R

Leave the value of the data stack pointer. "s-p-fetch

SYSVOL> (n flag ---)

Use after a call of <SYSVOL. Select drive n and ask the operator to reinsert his disc if flag is true and the boot drive is currently selected.

TRUE (--- flag)

Leave the numerical value of a true flag.

UO (--- addr)

Leave the address of the user area. "u-zero"

WAS (+++ addr) I

Use in the form

WAS <name>

If compiling, compile the pfa of <name> followed by a call to (WAS), otherwise load the pfa on the stack and call (WAS) directly. Use WAS to leave the current value of the execution vector <name> on stack.

15.STRINGS

!$ (s addr ---)

Store string s at addr. "store-dollar"

![]$ (char s n ---)

Overwrite the nth character of s with n. "store-bracket-dollar"

" I

Use in the form

" this is a string"

If compiling, compile a string literal to be loaded at HERE when the definition is executed. If executing, copy the string to PAD. Abort if the string exceeds 255 characters, or if the input stream is exhausted before the terminator " is reached. "quote"

$+C (s char --- s1)

Append the character char on the end of s. The length of s is not checked. "dollar-plus-c"

$->C (s --- char)

Leave the first character of s. Use to convert single character strings to a character on stack. "dollar-to-c"

(") (--- s)

Copy the string at IP+2 to memory at HERE and advance IP past the string. "bracket-quote"

(.")

Output the string at IP+2 and advance IP past the string. "bracket-dot-quote"

+$ (s1 s0 --- s1)

Append s0 to the end of s1. No check made on the resultant string length. "plus-dollar"

." I

Use in the form

." this is a message"

If compiling, compile a string literal which will be output by (.") when the definition is executed. If executing, output the string directly. Abort if the string is more than 255 characters long, or if the input stream is exhausted before the terminating " is reached. "dot-quote"

<$ (s1 s0 --- flag)

Leave a true flag if s1 precedes s0, otherwise leave a false flag. "less-dollar"

<=$ (s1 s0 --- flag)

Leave a true flag if s1 precedes, or equals, s0, otherwise leave a false flag. "less-equals-dollar"

=$ (s1 s0 --- flag)

Leave a true flag if s1 equals s0, otherwise leave a false flag. "equals-dollar"

>$ (s1 s0 --- flag)

Leave a true flag if s0 precedes s1, otherwise leave a false flag. "greater-dollar"

>HERE (addr count --- s)

Leave a string of count characters at HERE copied from addr. Don't check the value of count. "to-here"

>PAD (addr count --- s)

Leave a string of count characters at PAD copied from addr. Don't check the value of count. "to-pad"

@$ (addr --- s)

Fetch the string at addr to PAD. Leave the value of PAD. "fetch-dollar"

@[]$ (s n --- char)

Leave the nth character of s. "fetch-bracket-dollar"

C->$ (char --- s)

Convert char to a single character string and leave it at PAD. "c-to-dollar"

C? (addr n char --- addr1 n1 or 0)

Scan the n characters beginning at addr until a match with char is found. Leave the address of the matching character and the count of characters not scanned, inclusive of the matching one. Leave zero if no match was found or n is zero. "c-query"

DELETE$ (s n1 n0 --- s)

Delete n0 characters from s beginning with the n1-th. "delete-dollar"

IN$ (char1 char2 --- s)

Call ENCLOSE with char1 and char2 as delimiters and leave the enclosed characters at HERE. "in-dollar"

INSERT$ (s1 n s0 --- s1)

Insert s0 in s1 immediately preceding the nth character. If the value of n is one greater than the length of s, append s0 on the end of s1. Don't check the length of the resultant string. "insert-dollar"

LEN$ (s --- n)

Leave the length of s. "len-dollar"

MATCH$ (addr1 addr0 n --- n)

Compare the n characters at addr1 and addr0. Leave a negative value if the n characters at addr1 precede those at addr2, zero if they are equal, and a positive non-zero value if those at addr0 precede the ones at addr1. "match-dollar"

NEXT$ (s1 s0 --- s1 s0 n or 0)

Search s1 beginning with the character after the last point reached by a previous call of NEXT$ or SCAN$. Stop at the first instance of the string s0 and leave the index of the first matching character of s1 as well as the original strings. If no match is found, just leave a zero. "next-dollar"

SUB$ (s n1 n0 --- s)

Reduce the string s to the n0 characters beginning with the n1-th. "sub-dollar"

TYPE$ (s ---)

Output the string s. "type-dollar"

SCAN$ (s1 s0 --- s1 s0 n or 0)

Search s1 beginning with the first character and stop at the first instance of s0. Leave the index of the first matching character of s1 as well as the original strings. If no match is found, just leave zero. "scan-dollar"

16. HEAP MANAGER

-HEAP (n --- addr)

Leave the address of an area of at least n bytes of heap space. "**Minus-heap**"

+HEAP (addr ---)

Give back the area of heap at addr to the heap. The value of addr must have been obtained from a prior call of -HEAP. "plus-heap"

>HEAP (addr len --- s)

Leave the len characters at addr on the heap at s in string form. "to-heap"

ALLOT-HEAP (addr n ---)

Create a heap of n bytes at addr.

CLEAR-HEAP
Re-initialise the heap losing anything currently held on it.

17. LOCAL VARIABLES

+TO

Arrange that the next time a local variable is executed the number on stack is added to it. "plus-to"

ARGUMENTS (n ---)

Create space on stack for 9 local variables assigning the first n to the n items already on stack.

RESULTS (n ---)

Drop 9-n stack items to leave the first n local variables as results.

S1 to S9

If the local variable call precedes a call of +TO or TO, leave the value of the local variable. If it is the first call after TO, then store the number on stack in the local variable. If it is the first call after +TO, add the number on stack to the local variable.

TO

Arrange that the next call of a local variable causes the number on stack to be stored at the variable.

18. GRAPHICS & GAME CONTROLLERS

ASPECT-RATIO (n0 --- n1)

Multiply n0 by the ratio width/height measured in pixels. Use to provide square aspect ratios for critical graphics figures.

BKGND (n ---)

Clear the hires page to colour n.

BUTTON (n --- flag)

Leave a true flag if button n is being pressed, otherwise leave false.

CHOOSE (n0 --- n1)

Leave a random number between 1 and n0-1. Assume that the random number seed has been set by a call to RANDOMISE.

CLRTOP

Clear the top 40 rows of the lores screen.

COLOUR (n ---)

Select lores colour n.

COS (n --- n1)

Leave the cosine of angle n correct to 4 decimal places and scaled by 10000. The angle is measured in degrees and can be positive or negative.

SIN (n --- n1)

Leave the sine of angle n correct to 4 decimal places and scaled by 10000. The angle is measured in degrees and can be positive or negative.

DISPLAY-SCALE (xmin xmax ymin ymax ---)

Select the minimum and maximum values for the x and y

coordinates. Avoids the need to work in terms of pixels.

DISPLAY-XY (x y --- x1 y1)

Convert the pixel coordinates x,y to external coordinates using the display scale set up by DISPLAY-SCALE.

DRAW (addr u ---)

Draw a shape on the hires screen using the shape table at addr. Rotate the shape by u.

GRAPHICS

A double word variable placed at the start of the graphics screens. It can be used as the load point if an overlay is created from the graphics code.

HCLR

Clear the hires screen to black.

HCOLOR (n ---)

Select hires colour n.

HEADING (--- n)

Leave the turtle's current heading.

HFIND (--- x y)

Leave the current hires pixel coordinates.

HIRES

Select the hires screen, leave the pen up, and select HWHITE1 as the pen color.

HLIN (y xl xr ---)

Draw a horizontal line on row y of the lores screen, from xl on the left to xr on the right.

HLINE (x y ---)

Draw a line from the current pixel location to coordinates x,y on the hires screen.

HPLOT (x y ---)

Plot a point at x,y on the hires screen.

HPOSN (x y ---)

Set the current hires pixel position to x,y.

HSCALE (n ---)

Set the scaling factor for DRAW to n.

LORES

Select mixed text & lores graphics. Clear the top 40 rows and set the colour to white.

MOVEBY (n ---)

Move the turtle forward by n pixels on the current heading. Draw the turtle's path if the pen is down.

MOVETO (x y ---)

Move the turtle to coordinates x,y. Draw the turtle's path if the pen is down. Don't alter the turtle's heading.

NOTE (n1 n0 ---)

Sound the Apple's speaker with frequency n1 for duration n0.

PADDLE (n0 --- n1)

Leave the current value of paddle n0.

PEEK (addr --- n)

Leave the value of the byte at addr.

POKE (addr ---)

Write a 1 to the byte at addr.

PEN-UP

Lift the turtle's pen.

PEN-DOWN

Lower the turtle's pen.

PLOT (x y ---)

Plot a block on the lores screen at x,y.

RANDOMISE

Initialise the random number seed.

SCRN (x y --- n)

Leave the colour of the block at x,y on the lores screen.

TEXT

Select the text screen.

TURN (n ---)

Turn the turtle by n degrees. If n is negative turn clockwise. If n is positive,turn anti-clockwise.

TURNTO (n ---)

Set the turtle's heading to n.

VLIN (x yb yt ---)

Draw a vertical line in column x on the lores screen, from yb at the bottom to yt at the top.

XDRAW (addr u ---)

Delete the shape at the current coordinates using the shape table at addr and rotation u.

XY-DISPLAY (x y --- x1 y1)

Convert the external coordinates x,y to pixel coordinates using the display scale set up by DISPLAY-SCALE.

19. ASSEMBLER

(*) (--- addr)

Leave the address of the nucleus routine that is called by U*. Call with a JSR instruction with the stack set up for U*. "bracket-u-star"

(*/MOD) (--- addr)

Leave the address of the nucleus routine that is called by */MOD. Call with a JSR instruction with the stack set up for a call to */MOD. "bracket-star-slash-mod"

(;CODE)

Compiled by ;CODE.

(ABS) (--- addr)

Leave the address of the nucleus routine called by ABS. Enter with a JSR instruction and the stack set up for a call to ABS. "bracket-abs"

(DABS) (--- addr)

Leave the address of the nucleus routine called by DABS. Enter with a JSR instruction and the stack set up for a call to DABS. "bracket-d-abs"

(DNEGATE) (--- addr)

Leave the address of the nucleus routine called by DNEGATE. Enter with a JSR instruction and the stack set up for a call to DNEGATE. "bracket-d-negate"

(M*) (--- addr)

Leave the address of the nucleus routine called by M*. Enter with a JSR instruction and the stack set up for a call to M*. "bracket-m-star"

(M/) (--- addr)

Leave the address of the nucleus routine called by M/. Enter with a JSR instruction and the stack set up for a call to M/. "bracket-m-slash"

(NEGATE)   (--- addr)

Leave the address of the nucleus routine called by NEGATE. Enter with a JSR instruction and the stack set up for a call to NEGATE. "bracket-negate"

(S->D)   (--- addr)

Leave the address of a routine that converts the top two numbers as follows:

(n1 n0 --- d1 n0)

In other words, it converts the second stack number to a double number. Enter with a JSR instruction and the stack set up as just described. "bracket-s-to-d"

(U/MOD)   (--- addr)

Leave the address of the nucleus routine called by U/MOD. Enter with a JSR instruction and the stack set up for a call to U/MOD. "bracket-u-slash-mod"

)

Specify indirect absolute addressing mode for the next op-code generated. Only applies to the JMP, instruction.

#

Specify immediate addressing mode for the next op-code generated.

)Y

Specify indirect-indexed addressing mode for the next op-code generated.

,X

Specify indexed-X addressing mode for the next op-code generated.

,Y

Specify indexed-Y addressing mode for the next op-code generated.

.A

Specify accumulator addressing mode for the next op-code generated.

0<

Specify that the op-code generated for the following conditional will cause a branch if processor status N=0.

0=

Specify that the op-code generated for the following conditional will cause a branch if processor status Z=0.

0>=

Specify that the op-code generated for the following conditional will cause a branch if processor status N=1.

;CODE   E

Used to conclude a colon definition in the form:

: <name> CREATE ... ;CODE assembly code END-CODE

Stop compilation of the defining word <name> and start assembling the code between ;CODE and END-CODE.

When <name> later executes in the form

<name> <namex>

the definition <namex> will be created with its execution procedure given by the machine code following ;CODE. That is, when <namex> is executed, the address interpreter jumps to the code follwing ;CODE in <name>. When this happens the address on the top of the hardware stack will point to the byte preceding <namex>'s parameter field.

AGAIN,

Use in a CODE definition in the form:

BEGIN, ... AGAIN,

At run-time it marks the end of an unconditional loop. Control is returned to BEGIN,

ASSEMBLER E

Add the assembler vocabulary to the top of the context stack.

BEGIN,

Use in a CODE definition in one of the forms:

BEGIN, ... AGAIN,

BEGIN, ... UNTIL,

At run-time it marks the beginning of a repeated sequence of instructions. At the end of the loop, control returns to the point marked by BEGIN,. No code is assembled for it.

BOT

Addresses the bottom of the data stack by selecting the ,X addressing mode and leaving a zero on stack which can be modified if required to another byte offset in the data stack. For example, BOT 1+ LDA, loads the high byte from the bottom of stack cell.

CODE E

A defining word used in the form

CODE <name> .... END-CODE

Add the assembler vocabulary to the top of the context stack and create a dictionary entry for <name>. When <name> is later executed the machine code assembled beginning at the cfa will be entered.

CS

Specify that the op-code generated for the following conditional will cause a branch to be taken if the processor status C=0.

ELSE,

Use in a CODE definition in the form

cc IF, true part ELSE, false part THEN,

At run-time if the condition code is false, branch to the false part following ELSE,.

END-CODE E

Check that no garbage has been left on the stack and unsmudge the latest word in the dictionary if it is ok. Remove ASSEMBLER from the context stack.

IF,

Use in a code definition in the form

cc IF, true part ELSE, false part THEN,

or

cc IF, true part THEN,

At run-time the true part is selected if the condition code is true, otherwise the other branch is taken.

IP (--- addr)

Leave the address of the interpreter pointer for assembly into the next instruction.

IR (--- addr)

Leave the address of the indirect register for assembly into the next instruction.

LCOFF (--- addr)

Leave the address of a routine which will switch off the language card. Enter with a JSR instruction.

LCON (--- addr)

Leave the address of a routine which will switch on the language card with bank 2 selected. Enter with a JSR instruction.

N (--- addr)

Leave the address of the N-area for assembly into the next instruction.

NOT

Reverse the condition code that will be used to generate the next branch instruction.

NEXT (--- addr)

Leave the address of the FORTH address interpreter. All CODE definitions must exit directly to NEXT, or indirectly via routines such as POP.

POP (--- addr)

Leave the address of the nucleus routine that removes the top stack item before going to NEXT.

POP2 (--- addr)

Leave the address of the nucleus routine that removes the top 2 stack items before going to NEXT.

PUSH (--- addr)

Leave the address of the nucleus routine that adds the

accumulator as high-byte and the bottom machine stack byte as low-byte to the data stack before going to NEXT.

PUT (--- addr)

Leave the address of the nucleus routine that writes the accumulator as high-byte and the bottom of the machine stack byte as low byte over the existing bottom of data stack item.

RP)

A shorthand way of addressing the bottom byte of the return stack. It selects the ,X addressing mode and leaves $101 as offset. This can be modified if other bytes are to be accessed. Before operating on the return stack the X register must be saved in XSAVE.

SEC

Identical to BOT, except that 2 is left on the stack so that the second stack item will be accessed at run-time.

SETUP (--- addr)

Leave the address of the nucleus routine that loads the N-Area with addresses from the data stack. See the assembler description in Part II.

THEN,

Use in a CODE definition in the form

cc IF, true part THEN,

or

cc IF, true part ELSE, false part THEN,

It marks the point from which execution should continue after completion of the conditional code.

UNTIL,

Use in a CODE definition in the form

BEGIN, ... cc UNTIL,

At run-time a branch back to BEGIN, will take place unless the condition code is true.

UP (--- addr)

Leave the address of the location containing the address of the user area for assembly into the next instruction.

X)

Specify indexed indirect X addressing mode for the next op-code generated.

XSAVE (--- addr)

Leave the address of a location to be used for saving the X register for assembly into the next instruction.

APPENDIX 1 ERROR MESSAGES

STACK EMPTY

The program has taken too many items off the stack. Usually caused by doing too many DROPs. Use ?STACK in words to find this problem, and remove it after testing.

STACK FULL

The program has put too many items on the stack. Usually happens when a word in a loop leaves an item on the stack. Use ?STACK in loops to find this problem, and remove it after testing.

EXECUTION ONLY

The word is not allowed in colon definitions.

COMPILATION ONLY

The word is only allowed in colon definitions.

DEFINITION INCOMPLETE

The stack depth has changed during compilation of a colon definition, indicating some form of incompleteness.

DISC II ERROR

Usually the result of trying to access a block out of range. Could be a corrupt disc.

LOADING ONLY

The word is only allowed in a mass storage block. --> is the only instance of this in the system.

?

The word can't be found in any vocabulary on the context stack.

NAME MISSING

Create has not found the name of the new word before the end of the input stream.

NO TERMINATOR

Means that the end of the input stream has been reached before a terminating character was found.

STR > 255 CHARS

A string of more than 255 characters has been found in the input stream.

NULL DELIM

The parameter to WORD is a null character, and that isn't allowed by the 79 Standard.

DICTIONARY FULL

This means that there are less than 512 bytes left, so you'd better FORGET something!

STRUCTURE

Means that pairing of control structures is incorrect. e.g. too many IFs.

NOT ASCII CHARACTER

ASCII generates this if you don't follow it with a single character.

2 BUFFERS MINIMUM

The system requires a minimum of 2 block buffers. This message is produced by the buffer words if you try and free too many buffers.

NOT ENOUGH STORE

You've tried to create more block buffers than the system can support.

HAS INCORRECT ADDRESS MODE

This means you've used an assembler op-code illegally.

IS A PROTECTED DEFINITION

You've tried to FORGET a word that is in the protected part of the dictionary.

STRING TOO LONG

You've specified a search/replacement string of more than 63 characters.

HEAP FULL

The heap manager can't find enough space to satisfy your request. Check your program isn't looping, or use a bigger heap.

APPENDIX 2 ERROR REPORTS

When you've purchased your Metacrafts FORTH system, complete and return the registration card. This entitles you to use our error reporting service.

If you encounter a problem, and are unable to find a satisfactory solution, then send us an error report. We will endeavour to find a solution for you and let you know how to proceed.

Please make sure you let us have the following details when you send in an error report:

1. Serial number of your disc.

2. Hardware configuration.

3. A **complete** listing of your program, or a disc containing the source screens.

4. A **complete** description of the problem indicating **what** happens, **when** it happens, and **where** it happens.

5. Your name, address and a telephone number.

Send error reports to us at the address in the manual, and label them "ERROR REPORT".

APPENDIX 3 FURTHER READING

Starting FORTH by Leo Brodie.
published by Prentice Hall.

This is definitely the best introduction to FORTH that is available. Full of examples and exercises (with answers) . Uses the FORTH Inc. dialect of FORTH which differs slightly from the standard. Differences are marked.

FORTH PROGRAMMING by Leo J. Scanlon.
published by Howard W. Sams & Co., Inc.

Also an introduction to FORTH, but more formal than Starting FORTH. No exercises, but still worth reading.

FORTH Dimensions bi-monthly journal.

This is the official journal of the FORTH Interest Group. It is full of useful news and information about FORTH. You can reach them by writing to:

FORTH Interest Group
P.O.Box 1105
San Carlos
CA 94070
U S A

The British chapter of FIG produces a bi-monthly newsletter. For further details send a s.a.e. to:

The Honorary Secretary
FIG UK
15 St. Albans Mansions
Kensington Court Palace
London W8 5QH

Metacrafts FORTH comprises:

- complete implementation of the 1979 definition of FORTH with many extensions.
- 32 bit arithmetic capability
- powerful set of string and array handling functions
- low resolution graphics
- high resolution cartesian and turtle graphics
- full-screen editor with 'undo' and syntax check feature
- powerful macro assembler with structured programming constructs
- sophisticated debugger which allows you to test code at the source level.
- block copy utility
- source code documentation utility
- compiled-program decoder and memory dumper
- support for paginated printer output with optional headings and page numbers
- support for 80 column and extended memory cards
- on-stack local variables
- program overlay mechanism
- block buffer and heap store management
- completely reconfigurable input/output system
- source code provided for upper levels of the system

A comprehensive 170 page User's Guide contains all the information you need to be able to install and use Metacrafts FORTH. The Guide assumes that you can already write programs in FORTH. If you can't, don't despair - FORTH is easy to learn from one of the excellent tutorial texts recommended in the Guide.

The system is contained on a mini-floppy disc and requires an Apple II/IIe with at least 48K RAM and one 16 sector DISK II Drive.

To order your copy of Metacrafts FORTH, send a cheque for £68.70 (+ VAT) to Metacrafts Ltd., 144 Crewe Road, Shavington, Crewe CW2 5AJ. Please allow 14 days for delivery.

METACRAFTS
FORTH
FOR
APPLE II

Most people think that FORTH is just another programming language, and a pretty weird one at that! They've probably been told that it's difficult to learn, and completely unreadable.

The thousands of programmers using FORTH know differently. They know that their application packages execute up to 20 times faster than the equivalent BASIC versions, that they take a fraction of the time to develop, and that, because FORTH lets them use names of up to 32 characters (any, not just letters and numbers), the end product is more readable and, therefore,easier to maintain. In fact, they've found out that FORTH brings them real financial savings.

How can FORTH be so much better than BASIC?

To understand the answer to this question, you must understand the reason for BASIC's popularity.

In the early 60's, programmers had to use memory sparingly, and languages were designed to reflect this concern, even at the expense of being unwieldy. The concern for conserving memory permeated not only the language, but the whole computer system.

The languages of the early 60's, in particular BASIC, were resurrected in the mid-70's when personal computers came on the scene, which, although they were cheap, still didn't have much memory. Unfortunately, the explosive growth of the personal computer market has led to the widespread, but misguided, belief that a language that is simple to implement on a computer, is also simple for people to use.

FORTH was created in the early 70's by professional programmers who were dissatisfied with traditional approaches to developing software. They believed that ways must exist to significantly improve programmer productivity, and FORTH is their answer.

The heart of every FORTH system is a sophisticated interactive programming language, embedded in an extremely friendly program development environment. Taken together these constitute a programming system of incredible power and flexibility, and it is this total system that is called FORTH.

If you've grown tired of BASIC's line numbers, obscure variable names, GOTO and the rest of the features of the language that one distinguished computer scientist considers to be mentally mutilating, then FORTH is for you. Never mind the pundits who tell you that BASIC is best, that COBOL is the only business language, and that Pascal is the language for writing structured programs. Their vision of the future has been clouded by years of mainframe programming tradition.

Face the facts: programming always was, and still is, a difficult and costly business, and non of the traditional approaches have succeeded in reducing costs much on large mainframes, so ask yourself why applying them to micros should be any more successful.

FORTH, on the other hand, has demonstrated its success in the small computer field over more than ten years of applications development, ranging from observatory automation to fault-tolerant real-time commercial systems.

Now you too can escape from BASIC's mind crippling limitations and experience the full power of Metacrafts FORTH on your Apple II. Watch your productivity treble as you become familiar with the system, and treble again when you become an expert.

Not only will you be able to produce more, you will be astonished at how fast your Apple executes what you produce. For example, the January '83 edition of Byte magazine published benchmark figures for the Apple II which show that not only is Metacrafts FORTH 17 times faster than Applesoft BASIC and 3 times faster than Apple Pascal, but that it is also 14% faster than the fastest FORTH reviewed.

If you're still not convinced, examine the list of features supported by Metacrafts FORTH and ask yourself what other software system for Apple II provides as much for the same price.

FLOG	(f --- log(f))	Leave the natural log of f.
FEXP	(f --- exp(f))	Leave e to the power of f.
FSIN	(f --- sin(f))	Leave the sine of f radians.
FCOS	(f --- cos(f))	Leave the cosine of f radians.
FTAN	(f --- tan(f))	Leave the tangent of f radians.
FATAN	(f --- atan(f))	Leave the inverse tangent of f.
FENTIER	(f --- entier(f))	Leave the greatest integer, greater than or equal to f, as a floating point number.

Memory

F@	(addr --- f)	Fetch the number f from memory.
F!	(f addr ---)	Store f in memory at addr.

Defining words

FCONSTANT	(f+++)	Define constant ⟨name⟩ value f,
where ⟨name⟩	(--- f)	returns f at run time.
FVARIABLE	(+++)	Define variable ⟨name⟩
where ⟨name⟩	(--- addr)	leaves the variable's addr.
FARRAY	(n0 n1+++)	Define array ⟨name⟩ with lower bound n0, upper bound n1,
where n ⟨name⟩	(n --- addr)	leaves the addr of the n-th element.

Compiler

F,	(f ---)	Compile f into the dictionary.
FLITERAL	(f ---)	Compile f into current definition.

Error handling

?FERROR	(---)	Call FERROR if an arithmetic error occured in the last operation.
FERROR	(---)	An execution vector called by ?FERROR if an error occurs.
⟨FERROR⟩	(n ---)	Default definition called by FERROR. Aborts with a message as follows:
n=1	OVERFLOW	Result too big.
n=2	ZERO DIVIDE	Zero dividend for F/.
n=3	INVALID ARGUMENT	zero/-ve arg for FLOG or -ve arg for FSQR or F**.

Miscellaneous

2PI	(--- 2×pi)	Leave the value of 2xpi.
INT	(---)	Assign (NUMBER) to NUMBER.
FP	(---)	Assign ⟨NUMBER⟩ to NUMBER.
DEGREES	(f --- f)	Converts f degrees to radians.
FIX	(f --- d n)	Converts f to a double number d and exponent n. The value of d is in the range 99999999.9 ⟨d⟨ 1000000000, and n determines the position of the point as follows:
f=123456789.0	n=0	d=123456789
f=12345678.9	n=−1	d=123456789
f=1234567890	n=1	d=123456789
F⟩D	(f --- d)	Converts f to a double number after applying the entier function.
D⟩F	(d --- f)	Convert d to floating point form.

METACRAFTS
FORTH
FOR
APPLE II

FLOATING POINT WORD SET

Introduction

Metacrafts FORTH Floating Point Word Set is intended for users of Metacrafts FORTH who wish to write programs with computational limits that lie beyond the numeric range of the standard fixed point operators. Although floating point calculators impose a severe performance overhead, the resultant applications should nevertheless execute faster than the equivalents written in other high level languages.

Installation

Before doing anything else, you should make a working copy of the floating point master disc using the COPY utility described in Part II chapter I of the Metacrafts FORTH User's Guide. When you have done that, store the master disc in a safe place.

The floating point word definitions are stored on the disc in both source screen and overlay format. The source screens are held in blocks 1-56, and should be loaded as part of an application in the normal way.

For convenience, two identical overlays have been created from the source screens for you. The first overlay, held in blocks 60-65, restores to the end of the BOOT3 overlay. The second overlay, held in blocks 70-75, restores to the end of the editor overlay, and so you should first ensure that the editor is loaded before using this one.

You can create your own floating point overlay by SAVEing the dictionary from the load point REALS.

All of the words that appear in the glossary are loaded into the FORTH vocabulary. There are many word definitions, however, that belong to a vocabulary called FINTERNAL. These are internal subroutines used by the floating point word set, they don't appear in the glossary, and should be ignored.

Input Number Format

The floating point word set extends the set of valid input number formats to include floating point numbers. This feature must be switched on, after loading the word set, by executing the word FP. Before FORGETting, or otherwise removing, the word set from the dictionary, the feature must be switched off by executing the word INT (the system will crash if you forget).

All floating point numbers entered in interpret/compile mode must be prefixed by a $ (dollar) character to distinguish them from standard FORTH integers, and should conform to the Applesoft number syntax and numeric range defined on pages 4,31 of the Applesoft Reference Manual.

Floating point numbers may be read directly by an application, and the character string converted to internal floating point form by FNUMBER. Such numbers should not be prefixed by $.

Glossary

The stack notation used in the glossary is the same as in the User's Guide. The only addition is the letter f which is used to denote a floating point number. It is also useful to note that each floating point number occupies 6 bytes of memory.

Number input-output

<NUMBER>	(addr --- d)	Assigned to NUMBER by FP, it converts the string at addr into either an integer or floating point number.
E.	(f ---)	Output f in exponent form with trailing zeroes suppressed, and followed by a space.
E.R	(f n0 n1 ---)	Output f in exponent form, right justified in a field of n1 characters with n0 digits after the point.
F.	(f ---)	Output f in fixed point form if its value lies in the range 0.099999999<f<1000000000, else in exponent form. Suppress trailing zeroes and append a space.
F.R	(f n0 n1 ---)	Output f in fixed point form, right justified in a field of n1 characters, and n0 digits after the point. Fill field with ? if f too big.
F.PT	(--- addr)	Variable containing the character to be used as decimal point. Set to full stop.
FNUMBER	(addr --- f)	Convert the string at addr to a floating point number.
FTRAILER	(--- addr)	Variable containing the character to be output by E.R and F.R as trailing zero. Set to ASCII 0.

Stack manipulation

F>R	(f ---)	Transfer f to return stack.
FDROP	(f ---)	Remove f from the stack.
FDUP	(f --- f f)	Leave a duplicate copy of f.
FOVER	(f0 f1 --- f0 f1 f0)	Duplicate second item on top.
FSWAP	(f0 f1 --- f1 f0)	Reverse top two items.
FR>	(--- f)	Transfer f from return stack.
FROT	(f0 f1 f2 --- f1 f2 f0)	Move third item to top of stack.

Comparison

F0<	(f --- flag)	Leave true flag if f<0.
F0=	(f --- flag)	Leave true flag if f=0.
F0>	(F --- flag)	Leave true flag if f>0.
F<	(f0 f1 --- flag)	Leave true flag if f0<f1.
F=	(f0 f1 ---flag)	Leave true flag if f0=f1.
F>	(f0 f1 --- flag)	Leave true flag if f0>f1.

Arithmetic

F+	(f0 f1 --- f0+f1)	Leave sum of f0, f1.
F−	(f0 f1 --- f0−f1)	Leave difference of f0, f1.
F*	(f0 f1 --- f0*f1)	Leave product of f0, f1.
F/	(f0 f1 --- f0/f1)	Leave quotient of f0, f1.
FABS	(f --- abs(f))	Leave the absolute value of f.
FNEGATE	(f --- -f)	Leave the negation of f.
F**	(f0 f1 --- f0* *f1)	Leave f0 to the power of f1.
FSQR	(f --- sqrt(f)	Leave the square root of f.

SCREEN #138

```
Screen #140 on the system disc contains code for a VIDEX
compatible 80 col. card. If you have an APPLE IIe and an
APPLE 80 col card you should replace screen #140 on the
system disc with the code in the screen #139 listing below.
To activate the APPLE 80 col card use the word 80-COLS as
described in the user guide.


IMPORTANT:

-- It is not possible to switch back to 40 column mode using
   the current version of the system.

-- It is not possible to select a graphics mode with the 80 col
   card active. Any attempt to do so will result in corruption
   of the dictionary when TEXT mode is reselected.
```

SCREEN #139

```
( APPLE IIE 80 COL CARD                              6-JUN-83 KGL  )
HEX ASSEMBLER MEM  ( PATCH TO MAKE CR DO AUTO-LF)
0 150A C/   C351 15DB /   HERE      380 DP /
7F # AND,  PHA,  C357 JSR,  PLA,  D # CMP,
0= IF,  A # LDA,  C357 JSR,  THEN,  RTS,
( END OF PATCH)
DP /    CONTEXT PRUNE      50 C/T /
CODE 80INIT LCOFF JSR, XSAVE STX, C34B JSR,
     XSAVE LDX, LCON JSR, NEXT JMP, END-CODE
80INIT FORGET 80INIT  380 CSW /  C351 KSW /
FIND </CURSOR> IS /CURSOR     FIND <PAGE> IS PAGE
FIND <.OK> IS .OK   FIND <CLREOL> IS CLREOL
FIND <CLREOP> IS CLREOP  FIND <@CURSOR> IS @CURSOR DECIMAL
CR PAGE ." METACRAFTS FORTH-79 V1.2    (C) 1983" CR CR
." SERIAL NUMBER " SERIAL# S->D <# # # # # # #> TYPE
CR CR ." READY "
```

SCREEN #140

```
Some users have experienced difficulty updating the printer
overlay to handle their printer. This is due to an ommission in
the manual. The correct procedure is:
     1) edit the printer overlay
     2) dispose of the editor by typing FORGET OVERLAY
     3) load the printer overlay by typing 106 LOAD
     4) save the printer overlay by typing 31 SAVE OVERLAY.
A few DOCUMENT words are missing from the glossary. These are:
TITLE"          ( +++ )     Used as TITLE" TEXT FOR PAGE HEADING"
PAGE-NUMBER     ( n --- )   Set page number to n
NO-TITLE        ( --- )     Turn off page headings and numbering.
INDEX           ( n1 n2 --) Output an index of screens n1 to n2.
OUTLINE         ( n1 n2 --) Output an outline of screens n1 to n2
PRINT           ( n1 n2 --) Print screens n1 to n2.
PRINT1          ( n --- )   Print screen n.
1OUTLINE        ( n --- )   Produce an outline of screen n.
```