****

H A R T F O R T H

****

Table of Contents

Distribution Diskette

The distribution diskette has one file which contains the HARTFORTH compiler. The file is named "FORTH/CMD". You should make a working copy of HARTFORTH and preserve the distribution diskette safely. This manual covers both the Model I/III version of HARTFORTH and the Model 4 TRSDOS 6.x version of HARTFORTH. For use on the Model I or Model III, HARTFORTH is distributed on a 35-track single density LDOS formatted data diskette. The diskette is readable with DOSs other than TRSDOS on either machine. For Model I TRSDOS 2.3, read the "MODEL I TRSDOS PATCH" section before you attempt to use or BACKUP the HARTFORTH distribution diskette. For Model III TRSDOS 1.3, use the CONVERT program supplied with your DOS to copy HARTFORTH to your DOS diskette. For TRSDOS 6.x, HARTFORTH is distributed on a double density 40 track data diskette.

Model I TRSDOS Patch

Model I TRSDOS users will find difficulty in reading the distribution disk due to the data address mark used for the directory. Therefore, before making a BACKUP or copying HARTFORTH from the diskette, you will need to change one byte of the TRSDOS 2.3 disk driver using one of the following 3 methods. This change will in no way affect the operation of your TRSDOS.

Method (1) directly modifies the system diskette with a patch. To prepare for this patch, obtain a fresh BACKUP of your TRSDOS 2.3 to use for this operation. Then enter the following BASIC program and RUN it. After you RUN the program, re-BOOT your TRSDOS diskette to correct the byte in memory.

```
10 OPEN"R",1,"SYS0/SYS.WKIA:0"
20 FIELD 1,171 AS R1$, 1 AS RS$, 84 AS R2$
30 GET 1,3: LSET RS$="<": PUT 1,3: CLOSE: END
```

Method (2) uses DEBUG to change the byte in memory. Use this if you do not want to patch your TRSDOS system diskette and are familiar with DEBUG.

1. At TRSDOS Ready, type **DEBUG** followed by **<ENTER>**.
2. Depress the **<BREAK>** key to enter the DEBUGger.
3. Type **M46B0** followed by the **<SPACE>** bar.
4. Type **3C** followed by **<ENTER>**.
5. Type **G402D** followed by **<ENTER>**.

Method (3) uses a POKE from BASIC to change the value directly in memory. This procedure is as follows:

1. Enter BASIC (files = 0, protect no memory)
2. Type **POKE &H46B0,60** followed by **<ENTER>.**
3. Type **CMD"S** followed by **<ENTER>**.

Now, after using any one of the methods noted above, COPY the FORTH/CMD file from the HARTFORTH diskette to your TRSDOS diskette.

Note on Model I/III Compatibility

The Model I/III version of HARTFORTH should be compatible with all TRSDOS compatible DOS's as it uses only documented DOS and ROM calls that are common to both the Model I and Model III computers. This U.S. version detects whether HARTFORTH is running on a Model I or a Model III by the ROM contents at address 125H (Model III = 49H; Model I anything else). The contents of SYS+21 (see "An Overview of HARTFORTH") are modified to use the proper address for the High Memory pointer (4049H for Model I; 4411H Model III).

Model I users of TRSDOS 2.3 who suffer from the error in TRSDOS that crashes the system, or enters DEBUG, when the BREAK key is pressed can use the following FORTH word in your applications to eliminate this problem.

```
: BREAK.OFF 0 17173 C! ;
```

This error occurs because a flag is erroneously set inside TRSDOS that makes it think that DEBUG has been activated when it has not. The word above clears this flag and ensures that DEBUG cannot be entered. After typing the definition of BREAK.OFF exactly as shown and pressing <ENTER>, type BREAK.OFF (followed by <ENTER>) to action the definition and disable DEBUG. Alternatively you could type "0 17173 C! <ENTER>" to clear the flag directly without actually defining the word, BREAK.OFF.

Getting Started

Welcome to the world of FORTH. You now have available to you a powerful high-level language development system that will execute your programs at least ten times faster than interpreted BASIC while also supporting interactive modification and debugging. It is well worth reading this manual thoroughly and with care from front to back, even if a lot of it does not at the moment make sense. There is a lot of important detail given in it that may not at first reading be apparent, but is as well to have in the back of your mind for when you may need it. Also please remember that Rome wasn't built in a day and it will take some time to become fluent in FORTH (do you remember how difficult it seemed when you first started learning BASIC?). Start gently and work your way in slowly, the effort will be rewarded as you become fluent in this fascinating language.

The procedure to use HARTFORTH is very simple. Using the working copy that you have just made, run HARTFORTH by typing

```
FORTH <ENTER>
```

The program title will appear together with the command:

```
ENTER FILESPEC OF FORTH VIRTUAL MEMORY
```

Reply to this by typing:

```
FORTH/CMD <ENTER>
```

After a couple of seconds the compiler's prompt of 'OK' will appear. You are now up and running!

Note that this filename, FORTH/CMD, must be entered in capital letters as HARTFORTH does not support automatic lower to upper-case conversion. To assist this HARTFORTH sets the capital lock bit of the TRSDOS 6.x system flag table on entry (Model 4 operation) although lower case may be accessed as normal by toggling the CAPS key or using Shift-0.

The FORTH/CMD File

The file FORTH/CMD is 80K bytes long (where a 'K' is 1024) and contains a pre-compiled HARTFORTH system together with the FORTH source code for many utilities and extensions to the 79-STANDARD HARTFORTH kernel. When you are typing into HARTFORTH from the keyboard, the Left-Arrow key will backspace and erase a character, and Shift Left-Arrow will delete the entire line of text. These facilities to correct mistakes are provided by the FORTH word EXPECT which is also used by QUERY and by the outer interpreter and so are available when inputting to any programs that also use these words.

If you type 'INDEX' <ENTER> you will be given a list of the contents of the file. The numbers to the left of the vertical line are merely line numbers as the INDEX word uses the standard word LIST to list the contents of Screen 13 (i.e. the contents of the thirteenth block of FORTH/CMD, each block being 1K bytes long). The numbers to the right of the vertical line are the screen numbers of the relevant blocks of source code. Some of these screens are already compiled into HARTFORTH, these being screens 21 to 36 inclusive. If you wish to execute any of the other screens you will first have to compile them into HARTFORTH's dictionary using either the LOAD or the LOADS commands. Blocks 1 to 16 (17 to 19 are spare) are used to store the compiled code of HARTFORTH (this is the function of SAVE-SYSTEM) and so do not appear on the index. A detailed description of the purpose and words of each of the screens is given later in this manual.

The first thing that you will probably want to do (and should do) is to look at these screens and to do this you will need to use some of the EDITOR functions, so after reading the rest of the manual sit down and play with the editor until you feel happy using it. It is important to get completely familiar with the editor's facilities as you will be using it a lot to write and modify your screens of source code. To enter the full screen editor type:

```
EDITOR x VIEW <ENTER>
```

where "x" is the number of the screen you wish to look at. The screen will now be shown on the display and using the commands listed in the glossary you can move the cursor around and edit the screen at either the character or the line level. To save any changes you have made you need to press 'S' (save) which copies the contents of the display back to the area in memory where the Virtual Memory system placed the screen when it read it from disc. If you didn't do this your changes would be lost. By using the '+' and '-' keys you can see the last or the next screen and pressing 'Q' gets you back to the

EDITOR vocabulary. Pressing 'Q' again will return you to FORTH and save any screens that have been updated to disc.

At the end of the FORTH/CMD file there are three screens left blank for you to practice with, and to store your first programs in. Later you will be able to use the NEW.FILE utility to create new Virtual Memory files, and RESTART and SAVE-SYSTEM to transfer your programs to them so giving you a lot more space to work with. One file can occupy a whole disc (except that single disc drive owners will need to have a minimum DOS on their discs) so you will not be short of space in the future.

There is a function VLIST that displays all the words in the CONTEXT vocabulary and the output of VLIST may be halted by pressing any key; a further depression of any key will restart the listing. If a word doesn't seem to be present it is probably because it has not been LOADed. The following short definition will do a check and tell you without having to wade through VLIST's output. Try putting it on one of the spare screens with the EDITOR and then LOAD it so you won't have to type it in next time (though you will have to LOAD it again).

```
: GOT FIND IF ." Present" ELSE ." Not found " THEN CR ;
```

Use it to see if the word FRED is present by typing GOT FRED <ENTER>.

You will notice a thick vertical bar by some of the words in VLIST's output, these words are IMMEDIATE words that are always executed even if encountered in definitions. The reason that a word is IMMEDIATE is normally because it is required to take some compile-time action, for example, LITERAL takes a number from the stack and encloses it, together with the run-time literal handler, in the new word so that the number may be returned to the stack later when the word is executed.

Before trying to use a utility or facility it is important to read the enclosed description of the utility and the associated glossary that will detail the action of the FORTH words associated with that utility. Note that not every word that you will find on the HARTFORTH screens is described, those words that are not are intermediate words used to implement some higher function and are not meant to be used on their own.

An Introduction to FORTH

This manual is not intended to be a course of instruction in the FORTH language. However some of the inherent characteristics of the language will be briefly described. It is recommended that the user invests in some other literature concerning the FORTH language, bearing in mind that there are differences between implementations of FORTH, and that HARTFORTH conforms to the 79-STANDARD.

It can at times be difficult to trace FORTH literature but Foyle's of London carries a small stock in their section on computer languages as do some other specialist booksellers. In case of desperation I have found that Mountain View Press Inc, PO Box 4656, Mountain View, CA 9404, U.S.A. who specialize in the FORTH language are very helpful. They publish a range of FORTH literature and to order from them just send them your Visa (Barclaycard) or Mastercard (Access) number together with the expiry date (they need that in the States for mail-order whereas over here they don't seem to). They advertise in the BYTE magazine from time to time, and might be able to supply any books that are difficult to obtain from normal booksellers.

Amongst the books I would recommend are:-

FORTH-79 STANDARD published by the Forth Interest Group available from Mountain View. A copy is highly recommended though it is a standards document and needs careful study. It is not a tutorial but a reference.

STARTING FORTH by Brodie, published by Prentice-Hall. A copy of this is recommended for the complete novice but is rather 'American' in its treatment of the subject.

DISCOVER FORTH by Thom Hogan, published by Osborne/McGraw Hill. Another good book for the beginner but as with Starting Forth above be aware that some of the FORTH words described in these books do not belong to the 79-STANDARD but to other de-facto standards such as FIG-FORTH.

THREADED INTERPRETIVE LANGUAGES by R. Loeliger published by Byte Books. This is for the Z80 assembly programmer who is interested in how FORTH "ticks" internally. It does not describe a 79-STANDARD FORTH but a "home-brew" Forth-like language called ZIP. However the implementation techniques are the same and very interesting if you wish to have some idea as to how FORTH works. One for the technically minded who knows the Z80 processor.

THE COMPLETE FORTH by Alan Winfield, published by Sigma Technical Press. A highly recommended book and a complete introduction to 79-STANDARD FORTH. Be warned that the single/double precision input from the input stream described in Chapter 8 is not 79-STANDARD as implied and that as such HARTFORTH does not provide it, and that Chapter 7 does not stress the importance of clearing ">IN" before using QUERY. Otherwise an excellent book - and British!

BEGINNING FORTH by Paul M. Chirlian, Matrix Publishers, Beaverton, OR.

There is a FORTH Interest Group in the U.K. and the contact is The Hon. Sec., Forth Interest Group U.K., 15 St. Albans Mansion, Kensington Court Place, London, W8 5QH. The 79-STANDARD is available through them, and I would think that a stamped addressed envelope for their reply might be much appreciated.

The FORTH language originated from a man called Charles Moore who evolved it over about ten years for the rapid design and debugging of fast real-time programs. He subsequently founded a company, FORTH INC., whose business is the exploitation of the FORTH language, primarily as POLYFORTH, a multi-user system for the larger minicomputers although they also have a MICROFORTH for smaller computers. A Forth Interest Group grew up and has contributed greatly to the spread of the language on smaller computers by making widely available a version of FORTH called FIG-FORTH, and more recently by refining and publishing the 79-STANDARD definition of the language in an attempt to gain a higher level of transportability from one computer to another of programs written in FORTH using only this standardized set of words.

FORTH is hard to describe in the same terms as other computer languages. Its major characteristic is that it enables a set of commands that perform a function to be given a name, which name may then be used with other commands to form another name which performs a more complicated function.

A FORTH command consists of a single word which can be any combination of letters, numbers and punctuation, except that it cannot contain any spaces as FORTH uses spaces as the gaps between words. The 79-STANDARD defines a minimum set of these words that are to be supplied with any FORTH that complies with the standard and these words are then used as the basis for defining other more complex words. Because of this technique of building up functions from other lower level functions FORTH could be regarded as a very convenient way of defining and linking sub-routines or procedures. As you cannot use a name that has not yet been defined FORTH tends to produce highly structured programs and its control structures are designed to encourage this. This makes large programs much easier to understand and debug than BASIC. For example a PASCAL to FORTH translation and vice-versa is quite easily done, but it is much more difficult to translate BASIC into either FORTH or PASCAL due to the lack of structure in most BASIC programs.

FORTH is designed to operate on 16 bit words which may be data or byte addresses, thus giving an addressing range of 64K bytes. It is intended to operate with a mass storage medium (normally disc) and has no file structure. It regards the mass storage as a set of numbered 'blocks' of 1024 bytes and a Virtual Memory system is provided to access these blocks by swapping them in and out of the physical memory of the computer when required. The programmer accesses any desired block by putting the block number on the stack and invoking the word BLOCK which reads the block from disc to a buffer in memory and returns to the stack the address in memory of the start of the buffer. That address is valid only for the 1024 bytes of the block requested. By this means the interface to the mass storage device is entirely processor and memory device independent and provides a simple interface to the programmer who merely asks for the desired block and finds it placed in memory for him. If a conventional file structure is required it is entirely possible to write it in FORTH on top of the standard Virtual Memory system, but this is normally of little point for single-user micro-processor implementations of the language.

FORTH is a stack oriented language in which data is first put onto a Data Stack and then manipulated on the stack. This leads to the language having post-fixed operators, better known as Reverse Polish Notation, which looks odd to many people at first but a little practice is sufficient to make it second nature, and this has the advantage of not requiring the parentheses in expressions or the ordering of the priority of operators that conventional notation requires. Variables and Constants may be declared as required and this contributes to FORTH's structuring as non-important local variables are not named but are carried on the stack, while important global Variables and Constants are named.

In fact there are two stacks in FORTH; the first being the Data Stack which is very obvious to the programmer, the second being the Return Stack which is less obvious, though it can also be used to carry data - but with great care - because this stack is used by FORTH to hold its linking information as it works its way down a word's definition to get to the primitive definitions that do the actual work. It is also used by some versions of FORTH (HARTFORTH is one) to hold the parameters of a DO ... LOOP construct thus providing an automatic nesting mechanism to allow loops to be embedded in other loops. It is thus important that if the Return Stack is used by the programmer within a DO ... LOOP it is returned to its original state before the end of the loop. Similarly if the Return Stack is used within the definition of a word it must be cleaned up by the end of that word or FORTH's linking back to the higher level that invoked the word will be wrong and the system will crash.

All FORTH words are held in a 'dictionary' which can be extended by defining new words. These new words can be either machine code 'primitives' which are implemented using the Assembler supplied with FORTH, or they are 'secondaries' built out of words that already exist in the dictionary. It may not be obvious that the FORTH compiler, also called the 'outer interpreter', works in two modes depending on the value of a system variable called STATE. Whenever a word is given to the compiler, either from the keyboard or from a disc Screen (a disc block that contains FORTH source code is termed a 'screen') then FORTH looks up the address of that word in its dictionary of existing words. If the word doesn't exist then it is assumed to be a number and is converted to binary using the current value of the system variable BASE (normally 10 for decimal input) and placed on the stack if STATE=0, otherwise it is put into the dictionary as a literal to be returned to the stack when the word is executed. If it is not a valid number in that base then a WORD ERROR is displayed and the compiler returns to interpret input from the keyboard canceling any word that was part compiled. When a word is found that does exist then if STATE=0 the word is immediately executed, otherwise the word's address is compiled into the newest dictionary entry to become a part of the definition of a new word.

This switching of STATE is mainly performed by two FORTH words, ':' and ';'. The function of ':' is to take the next word <name> that follows, to create a new entry in the dictionary for that <name> and to set STATE=1. Following words thus have their addresses compiled into the definition of <name> so that they may be executed when <name> is invoked. When ';' is encountered at the end of a definition it sets STATE=0 and checks that no structuring errors have occurred during the definition. Further words from the input stream will now be executed until STATE is set non-zero again to compile the definition of another new word.

To give a simple example the following trivial word will get a line of text up to 32 characters long from the keyboard and echo it to the display. Note that you will have to type this definition into a screen because it is more than 80 characters long and so overfills the keyboard input buffer.

```
: ECHO ." Enter text - " PAD  32 EXPECT CR
       ." Your text was "
       PAD BEGIN DUP C  DUP 0> WHILE EMIT 1+ REPEAT DROP  DROP ;
```

The ':' creates a new word called ECHO in the dictionary and as STATE is set to 1 the outer interpreter compiles the addresses of the following words into the definition of ECHO until ';' is reached. In fact there exists a class of IMMEDIATE mode words that the compiler will recognize and will execute and not compile even if STATE=1, and ';' is one of these words as it must be to actually execute during compilation. Typing ECHO will now ask you for a string of text, will wait for you to type up to 32 characters and an <ENTER> and will then echo your line and re-enter the outer interpreter with the 'OK' prompt. Note that this definition uses the fact that EXPECT returns a null character at the end of the string.

Classically the action caused by a FORTH word is described using a 'stack picture' and a written description; this is the case with the enclosed glossaries. The actual source coding on the supplied FORTH Screens is not good FORTH practice due to the need to get a lot into a small space. The closest approach to good FORTH style is probably the Floating Point Screens where you will see that the structuring is easily visible. It is a good idea to put a 'stack picture' and description with each word as it is defined using the 'brackets' that allow comments to be ignored by FORTH. An alternative idea is to use alternate screens for code and comment and to compile them with a command that only LOADs every other Screen. For example:-

```
        : ALTERNATE.LOAD   ( n1 n2 ->   ,compiles every alternate )
                           (   screen from screen n1 to screen n2 )
            1+ SWAP DO              ( set up loop start and end   )
                CR I                 ( get index and do a newline )
                ." Loading Screen " .      ( print screen to load )
                I LOAD                          ( load the screen )
            2 +LOOP  ;                ( increment loop index by 2 )
```

Used as '2 8 ALTERNATE.LOAD' <ENTER>, this would compile screens 2,4,6, and 8 only, while showing how far it had got in the compilation by displaying the number of the screen that was currently being interpreted. This allows all the odd numbered screens to be used for holding comments. Commenting your source code is highly recommended as FORTH can be a terse language at times and it helps to know what a word does when you come to try to understand what you were doing six months later!

It is worth emphasizing that FORTH makes no distinction between the keyboard and the screens on disc as it has only one 'input stream'. While it is waiting for the keyboard it is in fact LOADing from BLOCK 0, although this block is in fact the keyboard input buffer and is only 80 bytes long. The blocks on disc start at BLOCK 1 and all that LOAD does is to point the outer interpreter to the appropriate disc block. Therefore it is possible to do anything from a screen on disc that can be done from the keyboard, and vice versa. It is this factor that makes FORTH so interactive and easy to debug. Any definition, once compiled can be exercised from the keyboard and the stack and variables examined to check that it is doing the correct thing.

Note also that several FORTH commands can be given on one line separated only by spaces. The action of the <ENTER> key only makes the keyboard input buffer available as Block 0 to be interpreted. During interpretation FORTH actions each word as it encounters it and does not need to parse commands like normal compilers with a conventional structure.

Any FORTH program ends up as a single word definition that is invoked to execute all the lower level definitions that constitute the program. It is not necessary for a FORTH program to ever return to the outer interpreter once it is invoked. The outer interpreter is present only to add new definitions to the dictionary and takes no part in the execution of a FORTH program.

FORTH's concept of vocabularies is sometimes confusing to beginners. At any time the outer interpreter acknowledges two vocabularies. These are the CONTEXT vocabulary which is searched for words to be executed, and the CURRENT vocabulary to which new words are added. Often these are the same, and usually they are the FORTH vocabulary. A vocabulary is merely a list of FORTH words within the dictionary that are linked together and all the words in the 79-STANDARD kernel are in the FORTH vocabulary. It is possible to define new vocabularies and two such, the EDITOR and the ASSEMBLER, are defined in the HARTFORTH screens. The purpose of a vocabulary is to keep a set of words separate from another set of words. For example INDEX in the FORTH vocabulary (screen 20) lists the contents of screen 20, while INDEX is redefined in the EDITOR to list the top (comment) lines of a range of screens. All vocabularies terminate back at the top of the FORTH vocabulary so that all vocabularies, besides containing their own words also contain all of the FORTH vocabulary, even the words that were defined in FORTH after the new vocabulary was defined. To access the words in any vocabulary it is necessary to make that vocabulary the CONTEXT vocabulary, which is done merely by invoking its name. To add words to a vocabulary it is necessary to make it the CURRENT vocabulary, which is the function of the word DEFINITIONS. This can be seen in the EDITOR screens and to use any of the EDITOR functions it is necessary to make EDITOR the CONTEXT vocabulary by typing 'EDITOR'. Note that all vocabulary names are IMMEDIATE so that you can change CONTEXT within a definition while compiling; if for some reason you wish to change CONTEXT during execution you will need to precede the vocabulary name by (COMPILE). Also note that ' and FIND operate on the CONTEXT vocabulary, while FORGET operates on the CURRENT vocabulary. To FORGET a word in the EDITOR vocabulary for example, you will need to say EDITOR DEFINITIONS to make EDITOR the CURRENT vocabulary. After FORGET both CONTEXT and CURRENT will be set to FORTH.

Because it is 16 bit word oriented, the FORTH kernel is specified with mainly 16 bit integer arithmetic and logical functions, with only a few 32 bit double length integer functions. However due to the extensibility of the language, and the care with which the original functions were specified, it is possible to write both extended length integer and floating point extensions to perform whatever arithmetic operations are required. A minor disadvantage of FORTH is that its structure is not ideally suited for 'number-crunching' as operands have to be transferred to the stack before being used, and stored away afterwards. The additional overhead of these moves slows FORTH down on larger more powerful processors, but on 8 bit machines other speed factors, like the lack of hardware multiply and divide instructions, far overshadow this limitation.

An Overview of HARTFORTH

HARTFORTH is a full FORTH that conforms totally to the 79-STANDARD as discussed previously. Some of the following information is of a detailed technical nature that the more advanced programmer may need to know and it is included for his reference and interest, but it is not necessary to understand all this information to use HARTFORTH effectively.

The technically interested may care to know that HARTFORTH is not a modified FIG-FORTH, but is an entirely new implementation internally designed around the 79-STANDARD. In fact HARTFORTH version 4 (the Model 4 version) is a Direct Threaded Code implementation of FORTH which provides an execution speed between 10% and 40% faster (on a Z80 at least) than the classical Indirect Threaded Code implementation. This means that all colon definitions have a three byte code field address that is in fact a CALL to the appropriate machine code COLON routine. Primitives have no code field as such, the machine code to be executed commences at the code field address. In fact all this is transparent to the programmer who need not know the details of the implementation. The Direct Threading of HARTFORTH is not in fact a CALL/RETURN implementation as this would slow the compiler down by tying up the stack for return addresses. The stack is used for data and the return stack is synthesised using the IX register as this provides the fastest possible execution speed on a Z80. Its memory usage starts at 3000H (5200H for the Model I/III version) where the 2 x 1024 byte Virtual Memory block buffers are situated, followed by the system variables such as BLK and >IN, the 80 byte terminal input buffer (BLOCK 0) and the Virtual Memory 256 byte sector buffer and 32 byte File Control Block (50 byte for Model I/III). The actual program comes next, this is the address that is returned by the word SYS and is the transfer address to start the program. The purpose of starting the Model 4 version at 3000H is to allow the advanced user to call TRSDOS 6.x library commands from within FORTH using the DO.SVC command if he so requires. During normal Model 4 FORTH operation, the area from 2600H to 2FFFH is available for use as "scratchpad" memory if required, and in fact the screen Editor uses the area to build its display output as does the memory, file and sector Editor on screens 56-62.

The two stacks are located in high memory with the Data Stack being located 64 bytes below the top of memory or F400H, whichever is the lower, in order to allow some measure of stack underflow in error conditions without overwriting anything important that might be up there (the Model I/III stack is based on the High Memory pointer contents). The Return Stack is located 512 bytes below the Data Stack. Both stacks grow downwards as is normal Z80 practice.

When adding new primitive words using the Assembler it is vital that the following Z80 registers are not altered. They may be used within the word but must be unchanged when END-CODE is invoked. The function of END-CODE is to exit from the ASSEMBLER vocabulary and terminate the machine code with a JP (IY) to return to FORTH's address interpreter.

Register BC contains FORTH's internal Program pointer.
Register IX is used as the Return Stack pointer.
Register IY contains the address of FORTH's NEXT code.
Register SP is used as the Data Stack pointer.

On entry to a machine code primitive, register HL contains the Code Field Address of that word. Both HL and DE registers may be used as required without being preserved, as can AF and the entire alternate register set. On entry to a ;CODE machine code sequence DE contains the parameter field address of the invoked word.

In addition to all the 79-STANDARD required words the HARTFORTH kernel contains some additional useful words and utilities which are documented in the glossary and which turn HARTFORTH into a fully-fledged FORTH development system.

Unlike many implementations of FORTH, HARTFORTH is designed to run under an operating system and so the Virtual Memory that it accesses for storage and retrieval purposes is not the disc medium directly (as would be normal) but is a file created and controlled by the operating system. It is the name of this file that is requested by the FORTH system when it is first entered. Doing this has several advantages in that it provides for FORTH files to be used by other programs and vice-versa and the fact that FORTH is running under an operating system is entirely transparent to the programmer, he merely appears to have a smaller disc than usual at his disposal.

In implementing this method the system has been constrained to work only within pre-created files of fixed lengths so that the possibility of interfering with other possibly valuable files on the disc by damaging the directory is eliminated. Pre-allocating the disc space ensures that there is never any need to update the GAT table in the directory so that FORTH never has to 'close' its Virtual Memory file. Therefore even if HARTFORTH crashes, the integrity of the directory is maintained. Having said this it should be noted that TRSDOS 6.0 sets an "open" bit in the directory of an open file to avoid two tasks writing to the file at once, so the word DOS in HARTFORTH, as well as flushing all the Virtual Memory buffers to disc, will also close the Virtual Memory file. Any error found when closing the Virtual Memory file will be displayed but normally the words 'NO ERROR' will be shown as confirmation that there was no problem on exit from HARTFORTH. In the event that the Virtual Memory file is found to be open when HARTFORTH is started, for example after a crash during program development, it will display the fact as 'FILE ALREADY OPEN' but will override the TRSDOS file protection mechanism and allow the file to be written to and then closed normally on exit. This effective fixing of the length of the Virtual Memory files is a slight disadvantage at times, however functions are provided within the HARTFORTH editor to help overcome this by allowing new files to be created from within HARTFORTH and to allow the current Virtual Memory file to be changed for another and manipulated at the individual block level. Enhancements to the 79-STANDARD have been built into the HARTFORTH kernel in the form of functions to call the standard operating system file handling routines so that other files may be created and accessed if required, but this is of course not 'pure' FORTH practice.

The Virtual Memory facility in HARTFORTH has two 1024 byte block buffers that are re-used on a least recently accessed basis. The terminal input buffer on this system (BLOCK 0) is 80 bytes long and filling it will cause an automatic ENTER to be generated by the 80'th character.

Both the KEY and EMIT words of HARTFORTH are defined as simple secondaries as follows:-

```
: KEY KEY.PRIMITIVE ;   : EMIT EMIT.PRIMITIVE ;
```

This allows input and output to be vectored to another device as required by 'ticking' the Code Field Address of a word that drives the required peripheral device into the Parameter Field of either KEY or EMIT. An example of this is shown in Screen 37 where EMIT is vectored to the printer. Care should be taken to store the original Parameter Field contents and restore them after use or communication with the system may be lost.

The HARTFORTH kernel has a function called 'SYS' that leaves on the FORTH stack the address of the start of the FORTH code. At this address are found a common set of parameters at standardised addresses relative to that returned by 'SYS'. A table of these parameters is given below but further knowledge will be necessary to appreciate the purpose of most of them.

SYS	FUNCTION
0	Jump to initialization routine.
3	Not used by this version of HARTFORTH.
5	Not used by this version of HARTFORTH.
7	Value of HERE if only kernel resident.
9	Value used for HERE at initialization.
11	Value of FORTH vocabulary link if only kernel resident.
13	Value used for FORTH vocabulary link at initialization.
15	Value of VLINK variable if only kernel resident.
17	Value of VLINK variable used for initialization.
19	Code Field Address of word executed after program start, normally the disc initialization word for the Virtual Memory file.
21	Pointer to address containing Data Stack address, normally HiMEM
23	Offset of initial Return Stack address from Data Stack, normally 512 bytes below the data stack.
25	Spare word for possible future use.
27	Spare word for possible future use.
29	Code Field Address of FORTH word executed after Virtual Memory file initialization, normally QUIT.
31	Length byte of following filename string. If zero then the user is prompted for the filename, otherwise the following string is used.
32	File specification (NNNNNNNN/XXX.PPPPPPPP:D) used for Virtual Memory file if SYS+31 is not zero. Maximum length is 23 bytes.
55	Byte containing processor flags after last disk access.
56	Byte containing contents of accumulator returned after last disk access.

It will be useful to know how the area of free memory at the top of the dictionary is used by the system as conflicts here may well occur. The area at HERE upwards is used by WORD to accept words from the input stream and the address returned by WORD is usually the same as HERE will return. The area from PAD (which returns HERE+65) upwards is used by both " and ." to hold the string of text specified and as the HARTFORTH string functions use PAD as a temporary storage area conflicts may arise. The print words, including '.' and 'U.' as well as the number conversion words <# # #> build the numeric output string from PAD-1 downwards before EMITting it.

Due to the absence of the square left and right brackets on the TRS-80 keyboard HARTFORTH has to use a non-standard title for the following words

'left-bracket' becomes	<(	instead of square left bracket
'right-bracket' becomes	)>	instead of square right bracket
'brackets-compile' becomes	(COMPILE)	

A library of standard screens is supplied with HARTFORTH to provide often used extensions to the language, such as double length and floating point maths, editing of source screens, string manipulation, arrays, etc.. A description of each of these classes of functions is given on the following pages and glossaries of their words are enclosed at the end of the manual, including a glossary for 79-STANDARD FORTH.

HARTFORTH Error Messages

There are six error messages that HARTFORTH may display, in addition to any of the normal DOS error messages that may be displayed in the event of an error during an attempted disc access. These error messages are detailed below.

```
-------------
Compile error
Block x ABORT
-------------
```

This error occurs if the compiler finds that the variable STATE is not zero at the end of interpretation of a block. The most likely cause is the omission of a semi-colon from the end of a definition. The block number 'x' gives the number of the source screen being interpreted when the error occurred. If 'x'=0 then the error was the result of keyboard input.

```
-------------------
<name> Forget error
-------------------
```

This error occurs if the compiler was asked to FORGET <name> when that name was not present in the CURRENT vocabulary.

```
-------------
Stack error
Block x ABORT
-------------
```

The reason for this error is that the compiler has found that the stack depth is less than zero when it has finished compiling a block, i.e. more items have been taken off the stack than have been put on. A barrier area of 32 words is provided to allow for a measure of stack underflow, but gross underflow can crash the system without giving any error message if there is any operating system code in High Memory. As before the number 'x' is the number of the block that caused the error.

```
-------------
Tick error
Block x ABORT
-------------
```

This error occurs if a word that is 'ticked' by the function ' is not present in the CONTEXT vocabulary. As before 'x' is the number of the block in which the error occurred.

```
----------------------
<name> Structure error
Block x ABORT
----------------------
```

This error occurs at the end of a definition, when semi-colon is compiled, if a DO ... LOOP/+LOOP, BEGIN ... UNTIL/AGAIN, BEGIN ... WHILE ... REPEAT, IF ... THEN or IF ... ELSE ... THEN structure is incorrectly constructed or nested. The <name> is the name of the definition in error, and 'x' is the number of the screen in which it is located.

```
-----------------
<name> Word error
Block x ABORT
-----------------
```

If a word is referenced either from the keyboard or during compilation that cannot be found in the CONTEXT vocabulary then this error occurs and compilation ceases. As before 'x' is the number of the screen in which the offending word may be found.

```
------------------------------
<name> - IS NOT UNIQUE,Block x
------------------------------
```

This is not strictly an error message, rather it is a warning that a word being defined in Block 'x' already exists in the CURRENT or FORTH vocabularies, and that the previous definition will not be accessible until the new definition is FORGOTTEN. Compilation is not interrupted and the new <name> is correctly compiled.

NOTE: Gross stack errors on the Data Stack or incorrectly altering the Return Stack can crash HARTFORTH without warning or error messages. This is a characteristic of all FORTH systems and is a result of giving the programmer the full freedom of the compiler. Fortunately the elegant structure and interactive nature of FORTH usually makes it extremely simple to diagnose and correct the offending definition.

Additional Functions

In the following screen references, the syntax "xx/yy" relates to the Model I/III screen number (xx) followed by the TRSDOS 6.x screen number (yy). The FORTH kernel includes some extra functions over and above those required by the 79-STANDARD. These functions are documented in the glossary and are in general self-explanatory. The major enhancements are the words that enable access to other files via the normal documented TRSDOS operating system calls, but to use them will require some technical appreciation of how to use these calls. On screen 28/36 there are a few additional functions, defined in FORTH, that complement these system calls.

IN and OUT are supplied to enable access to the input/output ports of the Z80 CPU and some 32 bit arithmetic functions are provided as primitives to speed up extended length arithmetic, such as the floating point functions.

A string literal " is provided as it seemed an obvious omission from the standard, and it should be noted that if executed directly, i.e. used outside a definition, then the address returned will be PAD. If compiled the address used will be that of the position of the string's length byte in the dictionary. The value of this length byte is used by HARTFORTH to jump over the string while executing a definition that uses " and so it is important that this value is not changed after compilation, by storing another string there for example. The word DOS is provided to flush any updated Virtual Memory buffers to disc and then close the Virtual Memory file and re-enter the DOS by means of the EXIT SVC call. A constant, VM.DCB, is provided that retains the address of the first byte of the Virtual Memory file Control Block and may be useful for advanced users.

Screen 13/20

This screen is mainly a comment screen that holds the index of the usage of HARTFORTH screens. It also contains the definition of the FORTH word INDEX whose job is to send the contents of screen 13/20 to the display.

Screens 14-15/21-22

Not all the words on these screens will be described here, the functions of the simpler ones which are not, may be found from the glossary of words of the screens. This applies also to the descriptions of the other screens as the function of these descriptions is to give an overview and some useful detail about the major functions of each screen.

It is important to realize the purpose of the words FORGET-SYSTEM and SAVE-SYSTEM. They are intended to be used in conjunction with RESTART, NEW.FILE, TO.MEMORY and TO.DISC to provide a suite of functions to ease the generation of new HARTFORTH files. When FORGET-SYSTEM is invoked it will trim off all the words that have been compiled from screen 13/20 onwards and will leave only a 79-STANDARD kernel with the additional words that are supplied in the kernel. This enables a new HARTFORTH to be compiled with only the functions present that any particular program requires.

The function of SAVE-SYSTEM is to save a memory image of the current system, that is loadable by the DOS, in the first blocks of the current Virtual Memory file. It asks if the entire system is to be saved. If the "N" key is pressed then only the kernel as left by FORGET-SYSTEM is saved to disc, the memory being unchanged. After saving the memory image it will display the number of blocks it has used. Be careful that sufficient room is left at the beginning of a Virtual Memory file if you are going to save a memory image as SAVE-SYSTEM will not care about overwriting a screen if it needs to. You do not need to save to the same file as you have compiled from of course, this is the reason that RESTART is provided to change the Virtual Memory file for another.

TO.MEMORY allows a range of screens to be read from the current Virtual Memory file and held in the spare memory from PAD upwards. The intention is that this is done prior to a RESTART to define a new Virtual Memory file and then the word TO.DISC can be used to store these screens in the new file. Note that these two words are in the EDITOR vocabulary and therefore EDITOR needs to be invoked after a RESTART because RESTART sets CONTEXT to FORTH.

The word BOOT is provided as a convenience to load the screens that are required for a given application; you can change it to suit yourself. The sequence of commands FORGET-SYSTEM 14/21 LOAD BOOT will forget the entire system and then reinstate it in the form in which you received it.

Screen 16/23

This screen contains a deliberately extremely limited Assembler for reasons that are explained in the glossary. It would of course be easy to provide a more complete Assembler, and in fact Loeliger gives a design for one in his book Threaded Interpretive Languages. However FORTH Assemblers tend to bear little resemblance to the native Assemblers for a processor because they normally rely on FORTH's stack to make their implementation easy and so have a Reverse Polish type of notation with the op-code after the operands. To people who also use the native Assemblers this can be very error prone and coding in Hex for the very minor amounts of machine code that most FORTH programs do (should!) have is quite easy. There are several examples of the use of the Assembler in a variety of these screens. In the Model 4 version, however, screens 67-71 contain a very advanced facility that is far more useful than an Assembler and takes up less space in the dictionary. It is termed a Native Code Generator and produces actual machine code sequences when invoked but using normal FORTH syntax so that additional execution speed may be obtained when required without needing to be able to write in Assembly code. This utility is more completely described later.

Screen 17/24

The terminal and print functions here are adequately detailed in the glossary, as are the useful control functions CASE: and SWITCH:. These two functions allow multi-way branching decisions to be taken with execution continuing in-line once the word branched to completes.

Screens 18-20/25-28

The EDITOR vocabulary contains both a screen editor invoked by the word VIEW as explained in the FORTH/CMD section of this manual, and also a set of words that will manipulate full screens. By this means blocks may be moved around and shuffled to suit your needs. If you need to copy screens from one Virtual Memory file to another they may be brought into memory using TO.MEMORY, the Virtual Memory file changed by using RESTART, and then the screens saved in the new file by using TO.DISC.

By convention the first line of a FORTH screen contains a title for that screen and the word INDEX (which is defined differently in the EDITOR than in FORTH) will print out the first line only of a range of screens. Be careful when using MOVE.UP and MOVE.DOWN that the screens that will be overwritten do not have valuable information on them as these words make no check whether the screens are empty.

The screen editor word VIEW also supports line editing as well as character editing and so the EDITOR vocabulary as a whole contains all that you need to create and maintain your FORTH program screens and Virtual Memory files.

Screen 21/29

The word DUMP that is provided on this screen is a facility that enables an area of memory to be displayed in both ASCII and HEX form without affecting the present setting of BASE. The display will always be in complete lines of 16 bytes, with a marker in the first line over the actual byte at the address that you gave. The words 'STAX' and 'STAX?' on this screen are in the glossary as general utility words as is 'DUMP' itself.

Screens 22-23/30-31

On these screens are provided the recommended 79-STANDARD DOUBLE NUMBER STANDARD EXTENSION word set that provides 32 bit operations together with some additional words to provide conversion between different word lengths on the stack.

Screen 24/32

Here are provided a set of array definitions for arrays of different word lengths. There is no mechanism for checking array bounds so it is as well to add checking, at least during development, as writing outside an array boundary will damage other parts of the dictionary - probably causing the system to crash. These words are examples of the use of ;CODE and CREATE ... DOES>.

Screen 25/33

A set of string manipulators are defined in this screen and should provide most, if not all, of the string handling facilities that any application may need.

Screens 26-28/34-36

Words to control the cursor of the TRS-80 display are defined here, together with a set of words to access the graphic characters and to draw horizontal and vertical lines.

The disk words defined build on the DOS calls implemented in the HARTFORTH kernel and some study of them should indicate their use, which is illustrated in the important word NEW.FILE which is used to pre-create Virtual Memory files of any required length. The word DO.SVC allows the advanced user access to the SVC calls of TRSDOS 6.x.

Screen 29/37

If you have a printer the words on this screen will provide you with some ready-made words to format output. Note however that HARTFORTH is designed to enable the KEY and EMIT words to be vectored to other drivers and there is an example of this on this screen that vectors EMIT to the printer using the PEMIT primitive.

Screen 30/38

The simple words on this screen provide the capability of generating a pseudo-random number sequence by a standard mathematical algorithm. To avoid generating the same sequence every time, the word RANDOMIZE will vary the sequence by changing the SEED value by generating and discarding an indeterminate number of random values. This indeterminate, but not necessarily truly random number is obtained by reading the refresh register of the Z80 processor.

Screens 31-35/39-43

The facilities provided here are intended to aid in the debugging of programs, and are hopefully self-explanatory with the possible exception of DEBUG. DEBUG is meant to be compiled into a definition and when invoked, perhaps conditionally on the occurrence of a fault, will enter the outer interpreter where you can examine variables and the stack to try to establish the cause of an error. The word RESUME will then continue from where the program was interrupted, but only if the number of items on the stack and the setting of HERE have not changed. DEBUG on entry stores the values of some critical parameters and restores them on exit; however it is possible that other parameters critical to the operation of the program might be altered, so some care is needed. As the ABORT function is not disabled a typing error at the keyboard may cause a HARTFORTH error that causes an ABORT. It is then not possible to RESUME as all the stacks are reset so again care needs to be taken before pressing the <ENTER> key to ensure that no mistake has been made.

Screens 36-41/44-49

These six screens provide a very full set of floating point manipulations that provide results accurate to eight decimal digits. The glossary contains a description of the format used to represent a floating point number but a few notes on the purpose of some of the functions will be given here.

Normally a floating point number is six bytes long and is stored and recalled using F! and F . Functions FPACK and FUNPACK are defined to pack a number into four bytes and unpack it again so allowing it to be stored and recalled by the double number functions 2 and 2! if it is required to save storage space because of memory limitations. The accuracy is however reduced to five or six decimal places and the constant F.LEN should be 'ticked' to a value of five if FPACK and FUNPACK are in use. As this packing reduces the exponent to eight bits and stores it at the bottom of the mantissa it is not possible to PACK a number whose exponent is less than -128 or greater than 128. FPACK checks for this and as written will ABORT. However this can of course be changed to take some application specific action if required.

Besides all the arithmetic and logical operators some scientific floating point conversions are provided. Note that the scientific notation adopted is that of single length integer representing the decimal exponent on the top of the stack, with a double length integer under it which represents the mantissa. These are provided to make it easier to enter floating point words from the keyboard if required.

Screens 42-44/* [Not available for TRSDOS 6.x version]**

Some benchmarks, that should not necessarily be taken too seriously as with all benchmarks, are given in these three screens. Probably the ones of most interest are those that are translations of the Personal Computer World magazine as these show that HARTFORTH is about ten times faster than interpreted BASIC. The times are obtained without "cheating". As variables are used they are fetched and stored each time whereas a real FORTH program would probably make more use of the stack. Also large BASIC programs run more slowly than small ones as GOTOs and GOSUBs have to search the program to find their destination, and variable references have to search the variable tables. This is why it pays in a BASIC program to use the most often referenced variables first to ensure that the search for them is short, and to put sub-routines at the front of a program. FORTH, on the other hand, runs at constant speed irrespective of the size of the program as all references are compiled.

Screens 45-47/50-52

These screens give a demonstration of a HEAPSORT which is one of the faster of the sorting algorithms and may be useful to you. The demonstration consists of two words. The first, DOIT, writes a set of random numbers to the memory at 2600H and sorts them into ascending order. The second, INVERSE, writes an inverted sequence starting at 255 down to 0 to the memory and then inverts the sequence. To compile these screens you will first need to compile screen 38 which contains the random number words. The result of the sort may be viewed by DUMP or the memory Editor utility.

The words DO.HEAP and HEAPSORT do a byte sort on an array whose address is returned by the word ELEMENT which takes a number from the stack representing the index into an array and returns the address of that element of the array. The array in this case is the display screen memory, so that you may see the sort progress. Note that the HEAPSORT algorithm assumes that the lowest index of an array is 1 whereas the array words of screen 32 use 0 as the lowest index. Therefore if the sort is to be modified for some other use the redefinition of ELEMENT should correct for this, or the sort will not reach the first element of the array. The algorithm uses a single working store whose address is returned by WORKING.

It should be quite simple to redefine ELEMENT to suit your own purposes and change WORKING (if necessary) to suit the type of value you wish to sort (byte, word, double-word, floating point, string .....). The C and C! associated with each ELEMENT and WORKING will also need changing, as will the comparisons used to make the sorting decisions. A bit of study of the algorithm, not to understand how it works but to see where it fetches, stores and compares, should enable you to make the required changes. Assuming that lines are numbered from 0 to 15 the following lines may need changing:-

```
        Screen 45/50:
          Line  1 - redefine WORKING and NO.ITEMS
          Line  2 - redefine ELEMENT;
          Line  9 - C  and C!
          Line 11 - C , C  and <
          Line 13 - C , C  and <
          Line 14 - C , C!
          Line 15 - C , C!

        Screen 46/51:
          Line  8 - C , C!
          Line 11 - C , C!
          Line 12 - C , C!
```

Screens */53-55 [Not available for Model I/III]**

These three screens contain some advanced code to interface FORTH to the TRSDOS interrupt task processor mechanism. If you LOAD these three screens and invoke INSTALL then you will hear a "click" approximately once per second as the FORTH word INTERRUPT is executed as a medium priority task in task slot 0. The word UNSTALL will remove the task.

To use this facility you need not understand how the code works but merely need to redefine INTERRUPT to invoke your required task. Be aware that if your task occupies too much time you will not be able to access the discs and if it occupies far too much time then the system will seem to lock-up. Also beware of all the usual interrupt problems of referencing the same variables etc. at interrupt as well as background level.

Screens */56-62 [Not available for Model I/III]**

These screens contain three Editors that are LOADed into the EDITOR vocabulary. They are MEDITOR for displaying and patching memory contents, FEDITOR for displaying and patching files on disk and SEDITOR for displaying and patching disc sectors directly.

SEDITOR is intelligent enough to know when you are accessing the directory and will re-write a directory sector with the correct Data Address Mark.

FEDITOR cannot tell if it has read a directory or an ordinary sector and will re-write all sectors with the normal Data Address Mark; so, although FEDITOR may be used to examine DIR/SYS it must not be used to patch it – use SEDITOR instead.

Screen */63 [Not available for Model I/III]**

This screen contains a more sophisticated floating point input routine than that provided in the floating point package. This version of F IN will accept input from the keyboard in the forms xxxx , xx.xx or xxExx.

Screen */64 [Not available for Model I/III]**

This screen contains a more sophisticated floating point output routine than that provided in the floating point package. Numbers whose absolute value is smaller than 1,000,000 or larger than 0.001 are printed as a mixed number with the number of digits defined by the constant F.LEN in the floating point package. Values outside this range are printed in scientific notation, again with the number of digits defined by F.LEN.

Screens */65-66 [Not available for Model I/III]**

One problem with FORTH is that the source code is held on screens in the Virtual Memory file and this tends to be fairly wasteful of disc storage space unless the words are packed in tightly, and then the code becomes unreadable. This waste of space also discourages adequate commenting of the source code. For those with large programs but limited disc storage this can be troublesome. These two screens allow FORTH source code to be LOADed from a word processor file such as SCRIPSIT. This is done by patching the definition of the FORTH word WORD that is used to accept the next word from the input stream. ABORT is also patched so that things are restored to normal if an error occurs. The loader is invoked by typing "WP.LOAD filename" which alters WORD and ABORT, opens the word processor file and sets BLK=0. Every time a new line character (13) is encountered BLK is incremented and if an error is encountered the block number printed is the line number that contains the error. Any other control characters are ignored except 0 which terminates the loading sequence and restores WORD and ABORT to their original functions. As LOADing is terminated when a 0 is encountered the word processor file used should use this as a terminating character.

Screens */67-71 [Not available for Model I/III]**

These screens contain a very advanced utility whose purpose is to replace the use of a normal FORTH Assembler where the speed of machine code is necessary. The purpose of this utility is to generate machine code sequences that emulate the action of normal FORTH words thus giving the convenience of writing high-level FORTH code with the increased speed of an Assembler. As always there is a compromise to be made, and in this case it is that the machine code generated occupies more memory than the equivalent high level FORTH code - however this is also true of a FORTH Assembler. The particular advantage of this Native Code Generator over an Assembler is that it occupies the same, or less, space in the dictionary but most importantly it offers the familiar FORTH syntax and thus requires no knowledge of the Z80 processor to achieve the benefits of Assembly coding.

Although the source code on the screens may appear daunting the use of the utility is very simple as the example screens following will show. The only words whose functions are necessary to understand are:-

:CODE , (COMPILE) , LITERAL , <(,)> and ;

Apart from :CODE the others are re-definitions of FORTH words whose actions are closest to those required by this code generator.

The code generator is actually in the ASSEMBLER vocabulary and :CODE performs the same function as : in that it generates the header of the definition in the dictionary but then instead of setting STATE=1 as : does it transfers control to the code generator. The code generator is thus running in execute mode (STATE=0) rather than compile mode and its function is to identify words that generate sequences of machine code in the definition created by :CODE and execute them. The words that are recognized by the code generator are all contained in the Stack Manipulation, Comparison, Arithmetic and Logical, Memory and Control Structure groups as well as numbers valid with regard to the current setting of BASE. Words not implemented are EXECUTE, LEAVE, DEPTH and NOT (which is the same as O=) and the double length words D<, D+, DNEGATE, */, U*, U/MOD.

Additional words are 2* and /2 which are self-explanatory, and IN and OUT which are equivalent to the IN and OUT defined in the HARTFORTH kernel.

In order to speed up fast loops a new loop count mechanism is provided. Because the loop counter is kept in registers B and C of the Z80 this mechanism is not nestable.

The word ; is redefined in the code generator so that it may be used to terminate :CODE definitions. When invoked it checks that there have been no control structuring errors and that the stack depth is the same as it was when :CODE was invoked; note that a stack error can also be caused by a structuring error. The code generator may give the following errors:

Code Generator Word error in xxx

This signifies that a word is invoked in the :CODE definition of xxx that is not implemented in the code generator.

Code Generator Compile error in xxx

This signifies that a stack or control structure error exists in the :CODE definition of xxx.

Code Generator Structure error in xxx

This signifies that a stack or control structure error exists in the :CODE definition of xxx.

Code Generator Stack error in xxx

This signifies that the depth of the stack when the ; of the definition of xxx is different to the depth when :CODE was invoked. Note that this means that unlike high level FORTH you cannot pass parameters into a definition using the stack but can use <(and)> followed by LITERAL to achieve the same effects.

It is useful to be able to invoke a high level FORTH definition from within the Code Generator and the redefinition of (COMPILE) allows this by compiling a machine code sequence that will execute the following word which (COMPILE) will FIND in the CURRENT or FORTH vocabularies. If the word is not found, the first error message above is given. Note that IMMEDIATE and CREATE...DOES> words and any other words that have both a compile and a run-time action cannot be invoked, e.g. ." because the code generator is executing with STATE=0. To use such words it is necessary to invoke them by a normal colon definition and then reference this definition using (COMPILE) as with the PRINT function in the sieve of Eratosthenes example screen.

It is important to be able to reference constants and variables and perhaps perform compile time address calculations. The word <(will suspend code generation and will allow words in the CURRENT or FORTH vocabularies to be executed, normally to leave a number or numbers on the stack, invoking)> will then re-activate the code generator and the word LITERAL may be used to take the number from the stack and create a code sequence to return it to the stack at run-time.

Please take note that the code generator is intended for use only to speed up time critical parts of FORTH code, say interrupt words or inner loops. It is not meant as a general purpose programming tool. Program in FORTH, tune your FORTH with the code generator!

Screens */72-73 [Not available for Model I/III]**

Screen 72 contains a colon definition of the well-known sieve of Eratosthenes benchmark and screen 73 is the :CODE definition of the same algorithm demonstrating (COMPILE), <(,)> and LITERAL.

Screens */74-76 [Not available for Model I/III]**

These screens contain a :CODE definition of the HEAPSORT algorithm to demonstrate the code generator.

Screen */77 [Not available for Model I/III]**

It may be that you wish to change the Virtual Memory file from within a program without doing a RESTART which will only accept the new name from the keyboard. This screen contains the code to show you how to do it. As shown it is invoked by typing "VM.FILE filename" but the '32 WORD' phrase in line 3 could be replaced by a string function that returns the address at which the filename can be found and the error trapping could take some application specific action.

Screens 48-53/78-80

These are six/three spare screens.

FORTH-79 Standard References

Stack inputs and outputs as shown; top of stack on right. See operand key at bottom.

Stack Manipulations

DUP	n -> n n	Duplicate top of stack.
DROP	n ->	Discard top of stack.
SWAP	n1 n2 -> n2 n1	Exchange top two stack items.
OVER	n1 n2 -> n1 n2 n1	Make copy of second item on top.
ROT	n1 n2 n3 -> n2 n3 n1	Rotate third item to top.
PICK	n1 -> n2	Copy n1-th item to top.
ROLL	n ->	Rotate n- th item to top.
?DUP	n -> n (n)	Duplicate only if non-zero.
>R	n ->	Move top item to "return stack" for temporary storage (use caution).
R>	-> n	Retrieve item from return stack.
R	-> n	Copy top of return stack onto stack.
DEPTH	-> n	Count number of items on stack.

Comparisons

<	n1 n2 -> flag	True if n1 less than n2.
=	n1 n2 -> flag	True if top two numbers are equal.
>	n1 n2 -> flag	True if n1 greater than n2.
0<	n -> flag	True if top number negative.
0=	n -> flag	True if top number zero. (Equivalent to NOT).
0>	n -> flag	True if top number greater than zero.
D<	d1 d2 -> flag	True if d1 less than d2.
U<	un1 un2 -> flag	Compare top two items as unsigned integers.
NOT	flag -> not flag	Reverse truth value. (Equivalent to 0=)

Operand key:

d, d1,...	32-bit signed numbers
addr, addr1,...	addresses
char	7-bit ASCII character value
n, n1,...	16-bit signed numbers
u	unsigned
byte	8-bit byte
flag	boolean flag

Arithmetic and Logical

+	n1 n2 -> sum	Add.
D+	d1 d2 -> sum	Add double-precision numbers.
-	n1 n2 -> diff	Subtract (n1 - n2).
1+	n -> n+1	Add 1 to top number.
1-	n -> n-	Subtract 1 from top number.
2+	n -> n+2	Add 2 to top number.
2-	n -> n-2	Subtract 2 from top number.
*	n1 n2 -> prod	Multiply.
/	n1 n2 -> quot	Divide (n1/n2). (Quotient rounded toward zero)
MOD	n1 n2 -> rem	Modulo (i.e. remainder from division n1/n2). Remainder has same sign as n1.
/MOD	n1 n2 -> rem quot	Divide, giving remainder and quotient.
*/MOD	n1 n2 n3 -> rem quot	Multiply, then divide (n1*n2/n3), with double-precision intermediate.
*/	n1 n2 n3 -> quot	Like */MOD, but give quotient only, rounded toward zero.
U*	un1 un2 -> ud	Multiply unsigned numbers, leaving unsigned double-precision result.
U/MOD	ud un -> urem uquot	Divide double number by single, giving remainder and quotient, all unsigned.
MAX	n1 n2 -> max	Leave greater of two numbers.
MIN	n1 n2 -> min	Leave lesser of two numbers.
ABS	n -> 1n1	Absolute value.

NEGATE	n -> -n	Leave two's complement.
DNEGATE	d -> -d	Leave two's complement of double-precision number.
AND	n1 n2 -> and	Bitwise logical AND.
OR	n1 n2 -> or	Bitwise logical OR.
XOR	n1 n2 -> xor	Bitwise logical exclusive-OR.

Memory

	addr -> n	Replace address by number at address.
!	n addr ->	Store n at address.
C	addr -> byte	Fetch least significant byte only.
C!	n addr ->	Store least significant byte only.
?	addr ->	Display number of address.
+!	n addr ->	Add n to number at addr.
MOVE	addr1 addr2 n->	Move n numbers starting at addr1 to memory starting at addr2, if n>0.
CMOVE	addr1 addr2 n->	Move n bytes starting at addr1 to memory starting at addr2, if n>0.
FILL	addr n byte ->	Fill n bytes in memory with byte beginning at addr, if n>0

Control Structures

DO...LOOP	do: end+1 start ->	Set up loop, given index range.
I	-> index	Place current loop index on data stack.
J	-> index	Return index of next outer loop in same definition.
LEAVE	->	Terminate loop at next LOOP or +LOOP, by setting limit equal to index.
DO...+LOOP	do: limit start ->	Like DO...LOOP, but adds stack value (instead of always 1) to index.
	+loop: n ->	Loop terminates when index is greater than or equal to limit (n>0), or when index is less than limit (n<0).
IF...(true)...THEN	if: flag ->	If top of stack true, execute.
If...(true)...ELSE ...(false)...THEN	if: flag ->	Same, but if false, execute ELSE clause.
BEGIN...UNTIL	until: flag ->	Loop back to BEGIN until true at UNTIL.
BEGIN...WHILE	while: flag ->	Loop while true at WHILE; REPEAT loops unconditionally to BEGIN.
...REPEAT		When false, continue after REPEAT
EXIT	->	Terminate execution of colon definition. (May not be used within DO...LOOP.)
EXECUTE	addr ->	Execute dictionary entry at compilation address on stack (e.g. address returned by FIND).

Terminal Input-Output

CR	->	Do a carriage return and line feed.
EMIT	char ->	Type ASCII value from stack.
SPACE	->	Type one space.
SPACES	n ->	Type n spaces, if n>0.
TYPE	addr n ->	Type string of n characters beginning at addr, if n>0.
COUNT	addr -> addr+1 n	Change address of string (prefixed by length byte at addr) to TYPE form.
-TRAILING	addr n1 -> addr n2	Reduce character count of string at addr to omit trailing blanks.
KEY	-> char	Read keyboard and leave ASCII value on stack.
EXPECT	addr n ->	Read n characters (or until carriage return) from terminal to address, with null(s) at end.
QUERY	->	Read line of up to 80 characters from terminal to input buffer.
WORD	char -> addr	Read next word from input stream using char as delimiter, or until null. Leave addr of length byte.

Numeric Conversion

BASE	-> addr	System variable containing radix for numeric conversion.
DECIMAL	->	Set decimal number base.
.	n ->	Print number with one trailing blank and sign if negative.
U.	un ->	Print top of stack as unsigned number with one trailing blank.
CONVERT	d1 addr1 -> d2 addr2	Convert string at addr1+1 to double number. Add to d1 leaving sum d2 and addr2 of first non-digit.
<	->	Start numeric output string conversion.
	ud1 -> ud2	Convert next digit of unsigned double number and add character to output string.
S	ud -> 0 0	Convert all significant digits of unsigned double number to output string.
HOLD	char ->	Add ASCII char to output string.
SIGN	n ->	Add minus sign to output string if n<0.
>	d -> addr n	Drop d and terminate numeric output string, leaving addr and count for TYPE.

Mass Storage Input/Output

LIST	n ->	List screen n and set SCR to contain n.

LOAD	n ->	Interpret screen n, then resume interpretation of the current input stream.
SCR	-> addr	System variable containing screen number most recently listed.
BLOCK	n -> addr	Leave memory address of block, reading from mass storage if necessary.
UPDATE	->	Mark last block referenced as modified.
BUFFER	n -> addr	Leave addr of a free buffer, assigned to block n; write previous contents to mass storage if UPDATEd.
SAVE-BUFFERS	->	Write all UPDATEd blocks to mass storage.
EMPTY-BUFFERS	->	Mark all buffers as empty, without writing UPDATEd blocks to mass storage.

Defining Words

: xxx	->	Begin colon definition of xxx.
;	->	End colon definition.
VARIABLE xxx	-> xxx: -> addr	Create a two-byte variable named xxx; returns address when executed.
CONSTANT xxx	n -> xxx: (-> n)	Create a constant named xxx with value n; returns value when executed.
VOCABULARY xxx	->	Create a vocabulary named xxx; becomes CONTEXT vocabulary when executed.
CREATE...DOES>	does: -> addr	Used to create a new defining word, with execution-time routine in high-level FORTH.

Vocabularies

CONTEXT	-> addr	System variable pointing to vocabulary where word names are searched for.
CURRENT	-> addr	System variable pointing to vocabulary where new definitions are put.
FORTH	->	Main vocabulary, contained in all other vocabularies. Execution of FORTH sets context vocabulary.
DEFINITIONS	->	Sets CURRENT vocabulary to CONTEXT.
' xxx	-> addr	Find address of xxx in dictionary; if used in definition, compile address.
FIND	-> addr	Leave compilation address of next word in input stream. If not in CONTEXT or FORTH, leave 0.
FORGET xxx	->	Forget all definitions back to and including xxx, which must be in CURRENT or FORTH.

Compiler

,	n ->	Compile a number into the dictionary.
ALLOT	n ->	Add n bytes to the parameter field of the most recently defined word.
."	->	Print message (terminated by "). If used in definition, print when executed.
IMMEDIATE	->	Mark last-defined word to be executed when encountered in a definition, rather than compiled.
LITERAL	n ->	If compiling, save n in dictionary, to be returned to stack when definition is executed.
STATE	-> addr	System variable whose value is non-zero when compilation is occurring.
<(	->	Stop compiling input text and begin executing.
)>	->	Stop executing input text and begin compiling.
COMPILE	->	Compile the address of the next non-IMMEDIATE word into the dictionary.
(COMPILE)	->	Compile the following word, even if IMMEDIATE.

Note: The <(,)> and (COMPILE) are slightly different from the 79-STANDARD symbols due to the lack of the necessary square bracket keys on the TRS-80 keyboard.

Miscellaneous

(	->	Begin comment, terminated by) on same line or screen; space after (.
HERE	-> addr	Leave address of next available dictionary location.
PAD	-> addr	Leave address of a scratch area of at least 64 bytes.
>IN	-> addr	System variable containing character offset into input buffer used, e.g. by WORD.
BLK	-> addr	System variable containing block number currently being interpreted, or 0 if from terminal.
ABORT	->	Clear data and return stacks, set execution mode, return control to terminal.
QUIT	->	Like ABORT, except does not clear data stack or print any message.
79-STANDARD	->	Verify that system conforms to FORTH-79 Standard.

Additional Words in FORTH Kernel

NFA	n1 -> n2)	Return Name Field Address n2 given Code Field address n1.
CFA	n1 -> n2	Return Code Field Address n2 given Name Field Address n1.
IN	n1 -> n2	Leave contents n2 of I/O port n1.
OUT	n1 n2 ->	Output n1 to port n2.
BEGIN...AGAIN	->	Unconditional form of BEGIN...UNTIL. Equivalent to 0 UNTIL.
INTERPRET	->	Begin interpretation of block number in BLK.
SEND	n ->	Equiv. to COUNT followed by TYPE.
HEX	->	Set BASE to 16. Equivalent to DECIMAL 16 BASE !
SYS	-> n	Returns address n of start to give access to certain parameters.
H	-> n	Variable used to store HERE.
" xxx"	->	String literal. Returns address of length byte of string xxxx.
DU*	ud1 ud2 -> uq	Multiply unsigned double numbers leaving quad result.
DU/MOD	uq ud -> udrem udquot	Divide quad number by double, both numbers assumed unsigned.
D-	d1 d2 -> diff	Subtract (d1 - d2).
DOS	->	Flush any updated buffers to disk, close VM file, enter DOS.
VM.DCB	-> n	Return address n of first byte of VM file DCB.

The following words invoke DOS standard file handling routines. The logical record length of all transactions is 256.

ERROR	flag -> flag+192	Display DOS error message. Flag is DOS error number.
WRITE	dcb -> flag	Write and verify a sector. Flag = 0 if no error occurred.
READ	dcb -> flag	Read a sector.
INIT	buffer, dcb -> flag	Open or create a file.
OPEN	buffer, dcb -> flag	Open an existing file.
CLOSE	dcb -> flag	Close an open file.
KILL	dcb -> flag	Kill an open file.
POSN	lrn dcb -> flag	Position to read or write a sector.

dcb	- is address of 32 byte area for DOS to use.
buffer	- is address of 256 byte area for DOS to read/write disk sector.
lrn	- is logical record number of sector to be read or written.

General Utility Words (In FORTH Vocabulary)

DOS	->	Save buffers and re-enter DOS.
CURRENT?	->	Print name of CURRENT vocabulary.
CONTEXT?	->	Print name of CONTEXT vocabulary.
BASE?	->	Print current BASE in decimal.
VLIST	->	Display name of all words in CONTEXT vocabulary. Pressing any key will temporarily stop/start the listing.
FORGET-SYSTEM	->	Used to clear the entire dictionary.
SAVE-SYSTEM	->	Will save either a fully compiled system or a basic system to the first blocks of the VM file in TRS-80 load format.
RESTART	->	Will restart FORTH at the Virtual Memory file query but will retain the system intact thus allowing a system compiled from one file to be stored in another.
LOADS	n1 n2 ->	Will load n2 blocks commencing with block n1.
BOOT	->	Will load resident system blocks.
STAX	-> n1 n2	Leave current data stack pointer n1, and return stack pointer n2.
STAX?	->	Print current values of data and return stack pointers.
DUMP	un1 un2 ->	Display memory contents from address un1 for un2 bytes.

Assembler: FORTH Vocabulary

CODE xxx	->	Create dictionary entry for following <name>. Set ASSEMBLER as context vocabulary. BASE is stored and reset to hex.
;CODE	->	Terminate a definition, set ASSEMBLER as context vocabulary. When definition is executed the code sequence following ;CODE will be invoked. BASE is set to hex. On entry to the definition DE contains the address of the parameter field of the word.
LABEL xxx	->	Create a header and enter ASSEMBLER vocabulary.
xxx	-> n	Normally used for machine code subroutines to be called by words defined by CODE. Invoking xxx puts the address of the subroutine on the stack.

Assembler: ASSEMBLER Vocabulary

C,	byte ->	Take byte from stack and put in dictionary advancing HERE. If word on stack not a byte value i.e. >255, then Abort giving error message.
END-CODE	->	Terminate code sequence with JP (IY) then set context vocabulary to current vocabulary and restore BASE to its previous value.

The ASSEMBLER offered is deliberately limited to the entering of hand-assembled hex code using ',' and 'C,'. The facilities it offers should be entirely adequate for most necessary purposes. The restriction of facilities is to avoid the syndrome of using a high-level language to provide an assembler to overcome the limitations of the high-level language to make it run as fast as possible. The language is for implementation, the assembler for fine-tuning, not vice-versa!

Additional Control Functions

CASE: xxx	->	Used to create dictionary entry for following <name> and compile execution addresses of words following <name> into <name's> body. When <name> is executed the word n from the stack is used as an index into the following words (0 gets first of list, 1 gets second, etc.) to choose which one to execute. After that word terminates execution continues with the next word after xxx. **No checks** are made on value of index. Beware of crashing the system if n is negative or larger than the list allows; e.g. CASE: CHOOSE ZERO ONE TWO THREE ; defines CHOOSE. Entering 1 CHOOSE will cause ONE to be executed.
xxx	n ->	
SWITCH: xxx	->	Used to create dictionary entry for following <name> and compile following list of numbers and execution addresses of words into <name's> body. When <name> is executed the value from the stack is compared with the number in front of each word's execution address in the list, and if the values are equal that word is executed. Only one word from the list is executed and if there is no match with the number execution continues with the word after xxx. The
xxx	n ->	
;SWITCH	->	

		definition **must** be terminated by ';SWITCH', e.g. SWITCH: NUMBER 10 TEN 8 EIGHT 2 TWO ;SWITCH defines NUMBER. Entering 8 NUMBER will cause EIGHT to be executed.
-->	->	An Immediate word allowing colon definitions to cross block boundaries. Causes interpretation on next block whether in compile mode or not.

Terminal and Print Functions

"IN	-> n	Get string from keyboard leaving address of length byte (PAD).
IN	-> n	Get number from keyboard to stack.
D IN	-> d	Get double length number from keyboard to stack.
?KEY	-> n	Scan keyboard, n=0 if no key pressed, else n=ASCII code.
U.R	n1 n2 ->	Print unsigned number n1, right aligned in field n2 long.
.R	n1 n2 ->	Print signed number n1, right aligned in field n2 long.
D.	d ->	Print double number signed.
D.R	d n ->	Print double number signed right aligned in n character field.

Virtual Memory Editor [These words are in the EDITOR vocabulary]

n CLEAR	n ->	Clear screen n to contain all spaces and mark as updated.
n1 n2 COPY	n1 n2 ->	Copy screen n1 to screen n2. Mark n2 as updated.
n1 n2 INDEX	n1 n2 ->	List line 0 of screens n1 to n2 inclusive.
n1 n2 TO.MEMORY	n1 n2 ->	Temporarily store in RAM screens n1 to n2 inclusive.
n1 n2 TO.DISC	n1 n2 ->	Reverse action of TO.MEMORY.
n1 n2 n3 MOVE.UP	n1 n2 n3 ->	Moves screens n1 to n2 inclusive up by n3 offset.
n1 n2 n3 MOVE.DOWN	n1 n2 n3 ->	Moves screens n1 to n2 inclusive down by n3 offset.
Q	->	Save all updated screens and make FORTH the context vocabulary.

Screen Editor [These words are in the EDITOR vocabulary]

n VIEW	n ->	Enter Full-screen editor, the following commands are not FORTH words and are actioned immediately.

H	Home the cursor to the top left of the screen.
T	Toggle the character under the cursor from upper to lower-case or vice-versa.
S	Save current screen contents in current buffer and mark as updated.
Q	Leave screen editor and re-enter outer interpreter.
I	Insert characters into current line, moving characters under and after cursor to the right, until ENTER, <UPARW>, <DNARW>, or <RTARW> are pressed.
R	Replace existing characters in current line, by typing over them, until ENTER, <UPARW>, <DNARW>, or <RTARW> are pressed.
D	Delete character under cursor and close up line from right.
<CLEAR>	Clear line from cursor to end of line.

<UPARW>, <DNARW>, <LTARW>, <RTARW>, <ENTER>, <SHIFT-RTARW>, and <SHIFT-LTARW> move cursor about the screen if not in insert or replace mode.

control C	Copy current line to PAD.
control P	Put line from PAD at this position and move other lines down.
control D	Delete this line and move other lines up. Line is saved in PAD if needed.
control R	Replace this line with the one at PAD.
control E	Empty the entire screen, fill with spaces.

+ OR ;	Will display next screen.
-	Will display previous screen.

Double Length Words

2!	d, n ->	Store double length number d at address n.
2	n -> d	Fetch double length number from address n.
2CONSTANT xxx	d ->	Define double length constant xxx. When invoked will leave d on stack.
2DROP	d ->	Drop double number.
2DUP	d -> d d	Duplicate double number.
2OVER	d1 d2 -> d1 d2 d1	Copy double number.
2ROT	d1 d2 d3 -> d2 d3 d1	Rotate set of three double numbers.
2SWAP	d1 d2 -> d2 d1	Swap two double numbers.

2VARIABLE xxx	->	Create double length variable of name xxx.
D0=	d -> f	Leave true flag if d is zero.
D=	d1 d2 -> f	Leave true flag if d1 = d2.
DABS	d1 -> d2	Leave d2 as absolute value of d1.
DMAX	d1 d2 -> d3	Leave larger of d1 and d2.
DMIN	d1 d2 -> d3	Leave smaller of d1 and d2.
DU<	ud1 ud2 -> f	True if ud1 is less than ud2. Both unsigned.
DNEGATE D+ D- DU* DU/MOD		These words exist in the FORTH kernel. See glossary of 'Additional words in FORTH kernel'.
D*	d1 d2 -> d3	Double length signed multiply.
D/	d1 d2 -> d3	Double length divide d1/d2 = d3. All signed.
D/MOD,DMOD,D*/,D*/MOD		Are analogs of single-length functions.
S>D	n -> d	Convert signed number to double number.
D>S	d -> n	Convert signed double number to single number.
D>Q	d -> q	Convert signed double number to quad number.
Q>D	q -> d	Convert signed quad number to double number.

String Handling Words

" ." SEND	These words exist in the FORTH kernel. Do **NOT** do any string operations on a string literal as the byte count is used to jump over the string. The string should be copied elsewhere if it is to be altered.	
"VARIABLE xxx	n ->	Create variable named xxx with length n. Do not try to store a string longer than n in this variable.
"CONSTANT xxx	n ->	Create string constant named xxx from string at n. Invoking xxx leaves address of byte count of string on stack.
"	n1 -> n2	Fetch string from address n1 to PAD leaving address n2 of PAD.
"!	n1 n2 ->	Store string at address n1 to address n2.

"LEFT	n1 n2 ->	Alter string at n1 to contain n2 left-most characters of original.
"RIGHT	n1 n2 ->	Alter string at n1 to contain n2 right-most characters of original.
"MID	n1 n2 n3 ->	Alter string at n1 to contain n2 characters of the original commencing from character n3.
"+	n1 n2 ->	Alter string at n1 by appending string at n2 to it and adjusting length byte.
COMPARE	n1 n2 n3 -> f	Returns f=0 if number of bytes n3 are identical starting at n1 and n2. Returns f=1 if bytes at n1 are alphabetically greater than those at n2. Returns f=-1 if bytes at n1 are alphabetically less than those at n2, used by "COMPARE.
"COMPARE	n1 n2 -> f	Return f=0 if strings at n1 and n2 are identical, f=1 if string at n1 is greater, f=-1 if string at n1 is less.
"=	n1 n2 -> f	Return flag=1 if string at n1 is ident-ical to string at n2, f=0 otherwise.
">	n1 n2 -> f	Return f=1 if string at n1 greater than string at n2, f=0 otherwise.
"<	n1 n2 -> f	Return f=1 if string at n1 less than string at n2, f=0 otherwise.

Arrays

ARRAY xxx xxx	n -> n1 -> n2	Create an array of n elements of single precision numbers. When xxx is refer-enced the address n2 returned is the address of the n1-th element. 0 < n1 <n.
2ARRAY xxx xxx	n -> n1 -> n2	As ARRAY but for double precision numbers.
"ARRAY xxx xxx	n1 n2 -> n1 -> n2	Create a string array of n2 elements n1 bytes long. Strings may be shorter than element size but must not be longer or the system may crash as the dictionary may be corrupted.

CARRAY xxx	n ->	Create an array of n bytes.
xxx	n1 -> n2	As ARRAY but for bytes.
FARRAY		Defined in the floating point screens.

TRS-80 Device Words

CURSOR	n ->	Move cursor to screen position n counting from top left corner; 0 <n < m. [m=1023 for Mod 1/3, 1919 for Mod 4]
CLS	->	Clear screen and home cursor.
LINE	n ->	Position cursor to start of line n; 0< n <m. [max=15 for Mod 1/3; 23 for 4]
TAB	n ->	Move cursor to n- th position of current line.
CURS.OFF	->	Turn cursor off.
CURS.ON	->	Turn cursor on.

The following six words are graphics functions. In the stack pictures, 0<x<h and 0<y<v; Model 4: h=159, v=71; Model 1/3: h=127, v=47.

GSET	x y ->	Set graphics bit at co-ordinates x,y.
GCLR	x y ->	Clear graphics bit at co-ordinates x,y.
G?	x y -> f	f=1 if graphics bit at x,y is set, f=0 otherwise.
HLINE	x y l ->	Draw a horizontal line of length l from co-ordinates x,y.
VLINE	x y l ->	Draw a vertical line of length l from co-ordinates x,y; l may be negative in both HLINE and VLINE.
BOX	x1 y1 x2 y2 ->	Draw a rectangular box, top left corner at x1,y1; bottom right corner at x2,y2.
DCB xxx	->	Allocate space for 32 byte DCB and 256 byte sector buffer. xxx then leaves address of this area.
xxx	-> n	
DATA	n -> n+50	Used after DCB name to access buffer.

FILENAME	n1 n2 ->	Moves string, length byte at n1, to DCB at n2 for use as filename for OPEN or INIT. Appends 0DH to string in DCB.
CHECK	f ->	If f>0 then do ERROR to display DOS error message.
NEW.FILE	->	Create new file or extend (but not shorten) existing file.
DO.SVC	bc de hl n - bc, de, hl, af	Do TRSDOS SVC call number n setting the processor registers to the initial values on the stack and leaving the values returned by the SVC call. [Valid for TRSDOS 6.x only].
PEMIT	c ->	Send ASCII character to printer.
PCR	->	Send CR to printer.
PSPACE	->	Send one space to printer.
PSPACES	n ->	Send n spaces to printer.
PTYPE	n1 n2 ->	Send n2 characters from address n1 to printer.
PSEND	n ->	Send string to printer, length byte at n.
PLIST	n ->	List screen n to printer.
PLISTS	n1 n2 ->)	List screens n1 to n2 inclusive to printer.
P.	n ->	Print n as signed integer with trailing blank.
PFF	->	Send a form feed to the printer.
P."	->	Send message to printer, message terminated by ".
EMIT.TO.PRINTER	->	Send all terminal output to the printer.
EMIT.TO.DISPLAY	->	Restore terminal output to display.

Random Numbers

SEED	-> n	Variable containing seed for random number generator.

RAND	->	Calculates next random number in the sequence and updates SEED.
RANDOM	n1 -> n2	Returns a random number n2 between 1 and n1.
RANDOMIZE	->	Randomizes SEED by discarding the next 1 to 127 random numbers, the actual number to be discarded being obtained by reading the Z80 refresh register.

Floating Point

Floating point numbers are held as 3 words on the stack and in memory. On the stack the binary exponent is on top with the double length mantissa underneath. The notional decimal point is positioned one position from the left of the word giving a range of values of approximately +/-0.5, although numbers are normalized to values of +/-0.25 thus preventing overflow on add or subtract. The exponent is signed and is the actual binary exponent (no offset). If a value of zero results from an add or subtract, or is entered externally, then the normalize function will "round" it upwards by adding a one at the least significant end of the value and decrementing the exponent.

FDROP	f ->	Lose floating point number from stack.
FDUP	f -> f, f	Duplicate f.p. number.
FOVER	f1 f2 -> f1 f2 f1	Duplicate second f.p. number on stack.
FSWAP	f1 f2 -> f2 f1	Swap top two f.p. numbers on stack.
FROT	f1 f2 f3 -> f2 f3 f1	Rotate f.p. numbers on stack.
FABS	f1 -> 1f1	Return absolute value of f.p. number.
F	n -> f	Fetch f.p. number from memory address n.
F!	f n ->	Store f.p. number at address n.
FCONSTANT xxx	f ->	Create f.p. constant xxx. Invoking xxx returns the f.p. number to the stack.
FVARIABLE xxx	->	Create f.p. variable xxx. Invoking xxx
xxx	-> n	returns the address of xxx to the stack.
FARRAY xxx	n ->	Create an array of n f.p. numbers named
xxx	n1 -> n2	xxx. Invoking xxx returns the address n2 of the n1-th element of the array (0 <n1 <n).
FNORM	f -> f	Normalize f.p. number on stack by shifting right or left until in range +/-0.25 then adjusting exponent.

FPACK	f -> fd	Pack floating point number into 2 words by putting 8 bit signed exponent at LS end of mantissa.
FUNPACK	fd -> f	Reverse action of FPACK.
F*	f1 f2 -> f3	Return normalized product f3 of f1 and f2.
F/	f1 f2 -> f3	Return normalized quotient f3 of f1 divided by f2.
F+	f1 f2 -> f3	Return normalized sum f3 of f1 and f2.
F-	f1 f2 -> f3	Return normalized difference f3 of f1 minus f2.
F<	f1 f2 -> flag	Return flag = 1 if f1 less than f2, flag = 0 otherwise.
F>	f1 f2 -> flag	Return flag = 1 if f1 greater than f2, flag = 0 otherwise.
F0<	f1 -> flag	Return flag = 1 if f1 negative, flag = 0 otherwise.
F0>	f1 -> flag	Return flag = 2 if f1 positive, flag = 0 otherwise.
D>F	d -> f	Convert double length integer d to normalized f.p. number.
F>D	f -> d	Return integer part of f.p. number as double length integer.
SCI>FBIN	d1 n2 -> f	Convert integer mantissa d1 and **decimal** exponent n1 to normalized f.p. number.
FBIN>SCI	f -> d1 n1	Convert f.p. number to integer mantissa d1 and **decimal** exponent n1.
F.	f ->	Print f.p. number as mixed number between 1 and 10 followed by decimal exponent. Number of digits printed in mantissa is governed by constant F.LEN.
F.LEN	-> 8	Constant that returns number of digits that F. produces. Normally 8, but may be "ticked" if fewer digits required.
FZERO	-> f	A small floating point number (0.5 x 2)

		which is meant for use in clearing accumulated totals, etc.. **Not** meant for use as a true zero.
FCONS xxx xxx	n1 n2 -> -> f	Create a f.p. constant xxx from n1 a **single length** integer, and n2, a **decimal** exponent.

Debug Facilities

DEBUG	->	If compiled into a definition this will enter the outer interpreter to allow examination of the stacks, variables, etc. to aid fault finding. Do not alter anything crucial to the operation of the system or the application. BASE, >IN, BLK and CONTEXT are stored on entry and restored on exit. HERE and DEPTH are stored and exit not allowed if either are changed. A modified OK prompt is used while in DEBUG.
RESUME	->	Used within DEBUG to return to the application.
DSTACK?	->	Non-destructively displays the data stack contents as signed numbers according to BASE. Shows the top 6 items.
RSTACK?	->	Non-destructively displays the top 6 items of the return stack as word names, if a valid Code Field Address, or as unsigned hexadecimal numbers otherwise.
DECOMPILE <name>	->	Displays a decompilation of the word <name> as word names, if a valid Code Field Address for a named word, or as unsigned hexadecimal numbers as may occur for headerless internal words used in implementing HARTFORTH. All control structures in HARTFORTH compile down to conditional branches or unconditional branches which may jump forwards or backwards. Branches are shown followed by a byte offset, the conditional branch branching if the top stack item is zero. A few trials will show how IF...THEN etc. are compiled but remember that each FORTH word jumped over is two bytes in the definition. If it is obviously going

		wrong pressing any key stops the decompilation. Primitive functions cannot be decompiled and produce meaningless output.
XREF <name>	n1 n2 ->	Search screens n1 to n2 inclusive for words that match <name> and print out the line numbers (0 -> 15) of every occurrence in each block. Pressing any key aborts the search.
REDEFINE <name1> <name2>	->	Amend Code Field Address of <name1> so that all existing and future references to <name1> actually execute <name2>. Effectively <name1> no longer exists although VLIST will still show it. Useful to temporarily correct an early incorrect definition without recompiling all subsequent definitions.

HARTFORTH Variations From "STANDARD" Practices

(Not Necessarily Non 79-STANDARD)

1) Doesn't accept multi-line definitions from keyboard. Would need minor change to INTERPRET to check STATE only if BLK=0.

2) Doesn't accept double-precision numbers from input stream if "." is part of number. Variable DPL is not present. To do this needs a rewrite of ?NUMBER.

A fix for 1) above is to FORGET-SYSTEM.

```
: MOD1 DROP BLK   0= IF STATE   IF R> DROP R> 14 - >R THEN  THEN;

FIND MOD1 ' INTERPRET 56 + ! HERE SYS 7
```

The Following Pertains to TRSDOS 6.x Version Only

Memory File and DisK Editor

MEDITOR	Displays and amends memory contents
FEDITOR	Displays and amends disk files
SEDITOR	Displays and amends disks at the individual sector level

All are in the EDITOR vocabulary. At the display level in all three editors the '+' displays the next disc sector or next 256 byte page of memory; '-' displays the previous sector or page. Out of range or invalid disk data is shown as all zeros. The arrow keys move the cursor around the screen.

'A' enters the ASCII modification mode which allows ASCII text to be typed in; 'X' enters the hexadecimal modification mode which allows hex numbers to be typed in. To exit either mode press a cursor key or ENTER. Memory modifications are done immediately, disk sector modifications are not done until 'S' is pressed at the display level. Pressing 'Q' at the display level exits from the editor.

Pressing 'M' or 'N' in MEDITOR at the display level requests a new memory address; pressing 'N' in FEDITOR requests a new sector number while in SEDITOR it requests a disk, track and sector number. The number at the bottom right of the display represents the cursor memory address in MEDITOR, file sector and cursor byte number in FEDITOR and track sector and cursor byte number in SEDITOR which displays a '*' in front of this number if a directory sector is read.

Native Code Generator

Words for which code can be generated are as follows:

```
        DUP  DROP  SWAP  OVER  ROT  PICK  ROLL  ?DUP  >R  R>  R
        <  =  >  0<  0=  U<
        +  -  1+  1-  2+  2-  *  /  MOD  2*  /2
        MAX  MIN  ABS  NEGATE  AND  OR  XOR
           !  C   C!  +!  MOVE  CMOVE  FILL
        DO  LOOP  +LOOP  I  J  IN  OUT
        IF  ELSE  THEN  BEGIN  UNTIL  WHILE  REPEAT  EXIT
```

Note that LOOP and +LOOP are non-standard in that they terminate ONLY when the count (after incrementing) is EQUAL to the limit. This was chosen for speed of execution.

The following loop mechanism is very fast but is non- nestable.

COUNTS	n ->	Set initial count to n-1.
COUNT?	-> n	Return current value of count.
END.COUNT	->	Decrement count by 1 and re-execute the word after COUNTS if COUNTS is > 0.
LITERAL	n ->	Enclose n into the definition to be returned to the stack when executing.
(COMPILE) xxx	->	Enclose code to invoke the CURRENT or FORTH word xxx at run-time.
<(	->	Stop generating code and execute CURRENT or FORTH words normally.
)>	->	Re-activate the code generator after a previous)>.