HT-FORTH

Hawthorne Technology

HT-FORTH

for K-OS ONE

Hawthorne Technology
8836 S. E. Stark
Portland, OR 97216

(503) 254-2005

232-7332

* * * PRODUCT DISCLAIMER * * *

This software and manual are sold 'as is' and without warranties as to performance or merchantablility. These programs are sold without any express or implied warranties. No warranty of fitness for a particular purpose is offered. The user must assume the entire risk of using the programs and is advised to test the programs thoroughly before relying on them.

Any liability of seller or manufacturer will be limited exclusively to product replacement or refund of the purchase price.

* * * * * * * * * * * * * * *

HT-FORTH

TABLE OF CONTENTS

INTRODUCTION iv

OPERATION 1-1

WORDS:
- CONTROL STRUCTURES 2-1
- TERMINAL INPUT / OUTPUT 2-4
- PROGRAM BEGINNING AND TERMINATION . . 2-6
- DICTIONARY ADDRESSES 2-7
- COMPILER AND INTERPRETER WORDS . . . 2-8
- VOCABULARIES 2-12
- DEFINING WORDS 2-14
- LOGIC 2-16
- STACK MANIPULATION 2-17
- COMPARISON 2-20
- ARITHMETIC 2-22
- MEMORY 2-26
- STRINGS 2-28
- NUMERIC CONVERSION 2-31
- FILES / K-OS 2-34

EXTENSIONS:
- POINTER WORDS 2-36

HT-FORTH EDITOR 3-1

HT-FORTH ASSEMBLER 4-1

DUMP 5-1

KERNAL MODIFICATION 6-1

INTRODUCTION

The K-OS ONE operating system is written in HTPL, a language that has a lot of features like the features of FORTH. Even though we supply the compiler for HTPL with each system, we have had a lot of requests for a standard FORTH to run with K-OS ONE and for use with the TinyGiant. Because of this demand, we have written a standard FORTH.

This manual assumes that the reader is familiar with the FORTH language. This is not intended to be a textbook to teach you how to program in FORTH or even how to use HT-FORTH to its best advantage. If you want to learn more about FORTH we recommend that you purchase one of the standard text books on FORTH.

If you already program in FORTH, this will make your K-OS ONE system more familiar and usable without having to learn a new language. It also means that you can transport an existing application to a newer machine with little effort. Another advantage is that there are a lot of books on FORTH and many user groups. Many of the books have sample programs that are usable. Because K-OS ONE uses the same disk format as MS/DOS you can easily take advantage of any FORTH programs for the PC.

K-OS FORTH is a full featured standard FORTH that runs with the K-OS ONE operating system. A full 32 bit stack is used to allow access to all of the memory. The generic arithmetic operators are all 32 bit. There are some special 16 bit arithmetic operators for special cases. The source code is compiled to inline macros or JSR or BSR. This allows for large programs but remains position independent and is very fast.

The traditional FORTH screens are not supported. Most K-OS ONE users indicated a preference for a screen editor and standard ASCII files. For this reason a Wordstar(tm) like screen editor is included. The words that support FORTH blocks can be easily added. Normal system ASCII files can be edited with any editor and then loaded. This is the format we use for programs written in FORTH.

While there are some significant differences in the way HT-FORTH is implemented, the words act the same as in other implementations. This manual explains the internal structure of the kernal used. The manual also explains the words that are included with the release version of HT-FORTH. Because we provide the complete source code it is possible to add any words that are needed but are not present. Also it is easy to edit the source to remove any uneeded words to make the kernal smaller. If there is any question on how a particular word works you should look at the source for that word.

A FORTH style 68000 assembler is included for small inline tasks. The conventional 68000 assembler that comes with the K-OS ONE operating system can be used instead of the built in assembler. The kernal is written for the standard assembler and not the FORTH assembler.

A set of utilities to use the K-OS ONE file system is included. These utilities include such things as opening files, reading from and writing to the disk and other devices. All the standard console I/O uses the operating system calls so it is portable to any set of hardware. With the use of the TRAP word, all of the system services of K-OS ONE are available to any program.

Some screen utilities are included. They can be customised for any terminal. String operators and definers are included so you can use Pascal like strings. A set of pointer manipulating words is included for using C style pointers.

An option when saving files is included so you can create standard .bin executable files. A file can be saved in a fixed mode that can't be changed for turnkey applications, or it can be saved in a modifyable, live mode for further development. As long as the end user does not have to ability to add words or change words, the resulting object can be distributed without royalties.

Many of the tricks that published FORTH programs use to save space or to get special features either won't work with HT-FORTH or won't save any space. In particular defining a number as a constant mearly wastes space and runs slower. Also, if something wasn't created with CREATE then DOES> will probably mess everything up.

Full source code for the compiler is provided. The kernal is written in 68000 assembler and can be reassembled with the standard K-OS ONE assembler. Existing words can be deleted to save space, or new words can be added. Many of the utilities that are supplied with HT-FORTH are written in FORTH. This manual will describe the action of each FORTH word included. A separate section will describe the theory of how the system is written and how it operates.

OPERATING

HT-FORTH is a FORTH dialect that runs under the K-OS ONE operating system on 68000 based micro-computers. HT-FORTH conforms, to a large extent, to the F83 standard. A major difference is that HT-FORTH uses the host file system rather than blocks or screens like other FORTH systems do. The philosophy is that the FORTH should work with the other parts of the operating system and not try to replace them.

Any reference to a variable will push a 32 bit address on the stack. The default size for arithmetic operators is 32 bits. All strings and messages are padded to end on an even boundary. Because of the need for certain operations in the 68000 to use word boundaries, character constants will always use at least 2 bytes.

The form that a compiled word takes depends on the word that is being compiled and where it is in the dictionary in relation to where it is being used. When possible, the compiler will use the shortest form it can. Many of the most common words are compiled in line as macros. For some words this created a problem because of the need to be inter-active. These words are compiled as subroutines.

Many FORTH programs define compiling words. In most cases these depend heavily on assumptions about how the FORTH is implemented. When sub-routines or macros are used instead of threaded code, these have to be changed. HT-FORTH is designed to be position independent. If a portable application is wanted then it is important not to use absolute address references for invoking words.

REGISTERS USED:

A7 -- RETURN STACK
A6 -- EVALUATION STACK
A5 -- BASE ADDRESS
D7 -- BYTE TO WORD OR LONG CONVERTER
D6 -- WORD TO LONG CONVERTER

RUNNING HT-FORTH

HT-FORTH is supplied as an executable binary file. To run it from the command processor just type:

```
        FORTH
```

at the normal system prompt. This loads the program and starts it executing.

After the system starts you should type:

```
     LOAD  DUMP.FTH
```

then:

```
     LOAD  EDITOR.FTH
```

to load the editor into the system. The default extension for FORTH programs is FTH.

To enter the editor type:

```
     EDIT
```

This causes the editor to start and the filename specified to be read into the edit buffer.

```
SAVE  filename   - save the current FORTH and dict
LOAD  filename   - load and compile
BYE              - exit FORTH and return to K-OS
```

CONTROL STRUCTURES

DO (limit start --)

Start a counted loop. All DO loops are executed at least once. They may be nested. You can get out of a DO loop before limit by using LEAVE.

LOOP (--)

End point of a do loop. Increment index and repeat loop if not at limit.

+LOOP (n --)

Add top of parameter stack to index and test for limit. If not at limit go back to DO.

LEAVE (--)

Exit a DO LOOP or DO +LOOP. This forces termination of the DO LOOP. Execution continues with the word after LOOP.

?LEAVE (TF --)

Exit a DO LOOP or DO +LOOP if top of evaluation stack is true. This is a conditional form of LEAVE.

I (-- I)

Inside a loop get current index value and put it on stack. Can only be used directly. Can not be used in a word that is called from inside a DO-LOOP.

I' (-- I)

Inside a loop get current limit value and put it on the parameter stack. Can only be used directly. Canot be used in a word that is called from inside a DO LOOP.

J (-- J)

Inside a loop get index of outer loop. Puts the index of the next outer loop on the stack. Can only be used directly, not from a word that is called from inside a DO-LOOP.

IF (tf --)

Execute if top of parameter stack is true.

ELSE (--)

Alternate code to execute if parameter was false.

ENDIF (--)

Terminate an IF construct. Functions same as THEN and can be used interchangeably.

THEN (--)

Terminates an IF construct the same as ENDIF.

BEGIN (--)

Start a loop that terminates with a condition.

AGAIN (--)

Uncontionally branch to the BEGIN that starts a loop.

UNTIL (xx --)

If value on parameter stack is false go to BEGIN. Marks the end of a BEGIN-flag-UNTIL loop.

WHILE (tf --)

While the condition is true perform the loop.

REPEAT (--)

Marks the end of a BEGIN ... WHILE ... REPEAT construct.

EXIT (--)

Terminate execution of a colon definition. Don't use it inside of a DO LOOP because the loop variables are kept on the return stack. If used in a DO LOOP the index and limit must be removed first.

EXECUTE (dd --)

Execute the word whose address is on the parameter stack. Execute expects a relative address.

TERMINAL INPUT/OUTPUT

.   (nn --)

Display signed number with trailing blank.

U.   (nn --)

Display unsigned number with trailing blank.

." text   (--)

Display text string that is enclosed: ." text" .

.((--)

Immediatly displays the text that is enclosed in parentheses: .(text) .

CR   (--)

Sends a Carriage Return and Line Feed to the current output device.

EMIT   (a --)

Display top of parameter stack as an ASCII character at the current output device.

TYPE   (addr nn --)

Display a string of nn characters starting at address nothing is displayed if 'nn' is zero or negative.

SPACE   (--)

Display one space.

SPACES (n --)

Display n spaces. Nothing is displayed if 'n' is zero or negative.

KEY   (-- x)

Wait for and get a keyboard character. High bit depends on system. K-OS normally sets high bit low.

EXPECT (addr n --)

Get n characters from the keyboard and store at address. This uses the line input system call.

SPAN   (-- addr)

Returns the 'addr' address of the count of characters stored by the latest execution of EXPECT.

PROGRAM BEGINNING AND TERMINATION

QUIT (--)

Return control to terminal, parameter stack unchanged, clear return stack. Prints "K-OS FORTH"

BYE (--)

Exit the FORTH language and return to the operating system.

ABORT (--)

Return to terminal, clear parameter stack, clear return stack.

ABORT" (flag --)

If flag is false, continue else display message and abort: ABORT" message"

WHAZZAT (--)

This is a general purpose error message that puts out three ??? and then pauses until a character is entered. The action taken depends on the character entered. If a return is entered the remainder of the line being interpreted is ignored. If a control c is entered FORTH is aborted and control is returned to the operating system.

DICTIONARY ADDRESSES

HERE (-- addr)

Return current dictionary pointer. This is the address of the next available dictionary location.

DP (-- addr)

This word returns the address of the dictionary pointer. DP @ is the same as HERE.

PAD (-- addr)

Get the address of scratch area of 128 bytes.

TIB (-- addr)

Get the address of the text input buffer.

>BODY (addr1 -- addr2)

Given the code-field address of a word 'addr1', >BODY returns the parameter-field address 'addr2' of the word.

COMPILER AND INTERPRETER WORDS

CAUTION: Most published FORTH programs make assumptions as to how the language is implemented and how the dictionary is set up.

LOAD filename (--)

Load the filename given until QUIT. The file is assumed to be a plain ascii text file like is created by a text editor.

SAVE filename (--)

The FORTH language and all the words that have been defined are save in the designated file. The file can later be loaded again and more words added.

LOCK (--)

This word does several things. Its purpose to to make the FORTH program that has been defined into a simple executable binary file. It then saves the file and exits back to the operating system.

SIZE

This is like a constant but it returns the current size of the FORTH language and all the definitions that have been added.

((--)

Marks the beginning of a comment: (comment). The comment continues until a closeing) is found. A space must follow the opening parenthesis but a space is not needed before the ending parenthesis of a comment. Because the content of a comment is ignored comments do not nest.

\ (--)

Cause the compiler to ignore the remainder of the line. Anything after \ on the line is ignored.

, (n --)

Compile a 32 bit constant value from the parameter stack.

,W (n --)

Compile the low order 16 bits of the top item on the parameter stack.

ALLOT (w --)

Add 'w' bytes to the current dictionary pointer. The result must end on an even boundary or an error will occur.

DOES> (--)

Mark the beginning of the run time behavior of a newly created defining word. This can only be used with CREATE. If used with constant or other word, it will result in an error condition.

IMMEDIATE (--)

Causes the last word defined to execute even when in a colon definition.

[COMPILE] (--)

When doing a colon definition compile the next word even if it is immediate.

COMPILE (--)

When doing a colon definition, compile the next word when the definition is executed.

STATE (-- s)

Return the state of the FORTH system. Is it compiling (non-zero) or is it executing (zero)?

LITERAL (--)

Compile the top item on the parameter stack as a literal. This allows the value of a constant to be calculated at compile time rather than run time.

[(--)

Exit compile mode. Enter execute mode.

] (--)

Begin compile mode again.

WORD (delim -- adrs)

Scan the input line and get the next group of characters delimited by the given character. The group of characters is stored beginning at adrs.

GETCX (-- chr)

Get the next character from the input source. This can be a line of text from the terminal or the next character from a text file tha is being loaded.

>IN (-- p)

Return a pointer to the next input character that will be scanned.

#TIB (-- p)

Return the address of the Text Input Buffer that gets command lines from the console.

NUMBER (-- nn)

This converts the last word found to a nmber using the current base if it is possible to do so.

VOCABULARIES

FORTH (--)

Makes the FORTH vocabulary the only one to be searched until another vocabulary is established.

DEFINITIONS

Makes the CONTEXT vocabulary current.

' (-- adrs)

Tick "'". Find the next word. Return execution address.

['] (-- adrs)

Returns and compiles the code-field address of a word in a colon definition.

FIND

Search the dictionary for the specified word.

FORGET (--)

Remove specified name from the dictionary and all words that were added after the one designated. This affects all vocabularies not just the current one. FORGET <name>

KILL (--)

Remove the next word found from the dictionary. All other words defined before and after the one word are not affected. KILL <name>

CONTEXT (-- adrs)

Returns the address of the vocabulary that will be searched first.

CURRENT (-- adrs)

Returns the address of the vocabulary where new words will be added.

VOCABULARY (--)

Create a new vacabulary that has for a name the next word found. Make the new vocabulary current.

VOCAB

This returns the address of a user variable that points to the last vocabulary link defined.

DEFINING WORDS

:   (--)

Begin a colon definition.

;   (--)

Terminate a colon definition.

HEADER   (--)

Create a dictionary entry but don't put anything in it. If a header is executed before something is put in it it will probably mess up the system.

CREATE   (--)

Create a dictionary entry that leaves its address at run time. Extra space is allocated so DOES> can be used to define a runtime behavior.

VARIABLE   (--)

Create a dictionary entry for a 4 byte variable. Does not initialize the contents of the variable.

CONSTANT   (k --)

Defines a word as a constant using the top of the parameter stack as the value: k CONSTANT <name> This creates a dictionary entry for <name> and compiles 'k' into the <name> parameter field. When <name> is executed, 'k' is put on the stack.

CODE (--)

This sets the numeric base to hex and allows direct entry of hex codes into a word.

REL   (x -- x)

This converts an absolute address into an address that is relative to the base of the FORTH program. Because HT-FORTH is position independent this is important. All variable and memory refrences use real addresses that are calculated when executed. The addresses of all routines are base relative.

UNREL   (x -- x)

This converts a base relative address into an absolute real address. This is needed if you want to look at the code for a routine.

LOGIC

NOT (a -- a)

Does bit-by-bit inversion of top of parameter stack.

AND (a b -- c)

Does bit-by-bit logical AND of the top two items of parameter stack.

OR (a b -- c)

Does bit-by-bit logical OR of the top two items of the parameter stack.

XOR (a b -- c)

Does bit-by-bit eXclusive OR of the top two items of the paramter stack.

STACK MANIPULATION

DUP (a -- a a)

Make a copy of the top of the parameter stack.

2DUP (a b -- a b a b)

Make a copy of the top two items on the parameter stack. Acts the same as OVER OVER.

DROP (a b -- a)

Remove the top item from the parameter stack.

2DROP (a b --)

Remove the top two items from the parameter stack. The same effect as DROP DROP.

SWAP (a b -- b a)

Exchange the places of the top two itmes of the parameter stack.

OVER (a b -- a b a)

Copy the second number on the stack to to top of the stack.

ROT (a b c -- b c a)

Rotate the top three items on the stack.

-ROT (a b c -- c a b)

Rotate the top three items on the parameter stack in the opposite direction of ROT.

PICK   (n -- a)

Copy the n'th item of the parameter stack, (not counting 'n' itself).
          0 pick is like DUP
          1 pick is like OVER

ROLL   (x x n --)

Remove the n'th number on the stack (not counting 'n') and puts 'n' on top of the stack.

?DUP   (a -- a a) or (0 -- 0)

Duplicate the top item of the parameter stack if it is not zero.

>R   (n --)

Move the top item of the parameter stack to the return stack.

R>   (-- n)

Move the top item of the return stack to the parameter stack.

R@   (-- n)

Copy the top item of the return stack to the parameter stack.

DEPTH   (-- n)

Return the number of items on the parameter stack.

NIP   (a b -- b)

Delete the next to the top item of the parameter stack.

TUCK   (a b -- b a b)

Put a copy of the top item of the parameter stack under the next to the top item. Similar to SWAP.

FLIP   (a b c -- c b a)

Flip the order of the top three items of the paramter stack.

COMPARISON

`<` (a b -- c)

Compares 'a' with 'b'. 'c' flag is true (non zero), if 'a' is less than 'b'.

`<=` (a b -- c)

Compares 'a' with 'b'. 'c' flag is true (non zero), if 'a' is less than or equal to 'b'.

`=` (a b -- c)

Compares 'a' with 'b'. 'c' flag is true (non zero), if 'a' is equal to 'b'.

`<>` (a b -- c)

Compares 'a' with 'b'. 'c' flag is true (non zero), if 'a' and 'b' are not equal.

`>` (a b -- c)

Compares 'a' with 'b'. 'c' flag is true (non zero), if 'a' is greater than 'b'.

`>=` (a b -- c)

Compares 'a' with 'b'. 'c' flag is true (non zero), if 'a' is greater than or equal to 'b'.

`0<` (a -- tf)

Compares 'a' with 0 (zero). Result is true (non zero), if 'a' is less than zero.

`0=` (a -- tf)

Compares 'a' with 0 (zero). Result is true (non zero), if 'a' is equal to zero.

`0>` (a -- tf)

Compares 'a' with 0 (zero). Result is true (non zero), if 'a' is greater than zero.

`U<` (a b -- tf)

Unsigned compare of 'a' with 'b'. Result is true (non zero), if 'a' is less than 'b'.

RANGE (a b c -- tf)

A range comparison is done. The result is true if the third item on the stack is equal to or included in the range of the other two items.

TRUE (-- t)

A symbolic constant that returns the value -1 that represents true when invoked.

FALSE (-- f)

A symbolic constant that returns the value 0 that represents false when invoked.

: 0<> 0= NOT ;

: <> = NOT ;

ARITHMETIC

```
+    ( a b -- c )
```

Add the top two items on the parameter stack. 'a' is added to 'b' and the result 'c' is left on the stack.

```
1+   ( a -- b )
```

Add 1 to the top item on the parameter stack.

```
2+   ( a -- b )
```

Add 2 to the top item on the parameter stack.

```
4+   ( a -- b )
```

Add 4 to the top item on the parameter stack.

```
-    ( a b -- c )
```

Subtract the top item on the parameter stack from the next to the top item. 'b' is subtracted from 'a' and the result 'c' is left on the stack.

```
1-   ( a -- b )
```

Subtract 1 from the top item on the parameter stack.

```
2-   ( a -- b )
```

Subtract two from the top item on the parameter stack.

```
4-   ( a -- b )
```

Subtract 4 from the top item on the parameter stack.

```
*    ( a b -- c )
```

Multiply the top two items on the parameter stack and return the 32 bit product on the stack. If there is an overflow, all but the least significiant of the top 32 bits are discarded.

```
2*   ( a -- b )
```

Multiply the top item on the parameter stack by two, by doing a shift left of one bit.

```
4*   ( a -- b )
```

Multiply the top item on the parameter stack by four, by doing a shift left of two bits.

```
/    ( a b -- c )
```

Divide the next item on the parameter stack by the top item on the stack. 'a' is divided by 'b' and leaves the quotient 'c' on the stack. If an underflow results, the result 'c', is 0. If division by zero is attempted the result 'c', will be -1.

```
2/   ( a -- b )
```

Divide the top item on the parameter stack by two by doing a shift right one bit.

```
4/
```

Divide the top item on the parameter stack by four by doing a shift right two bits.

```
MOD   ( a b -- c )
```

Divide the next item on the parameter stack 'a', by the top item 'b' and return the remainder of the division 'c'.

/MOD   (a b -- c d)

Divide the next item on the parameter stack 'a', by the top item 'b' and return both the quotient 'd' and the remainder 'c'.

*/   (a b c -- x)

Multiply the third item 'a' by the second item 'b' on the parameter stack. Divide the 64 bit double length product by the top item 'c'. Return the 32 bit quotient 'x'.

*/MOD   (a b c -- x y)

Multiply the third item 'a' by the second item 'b' on the parameter stack. Divide the 64 bit double length product by the top item 'c'. Return the 32 bit quotient 'x'and the remainder 'y'.

MAX   (a b -- c)

Compare the top two items on the parameter stack and return the larger value.

MIN   (a b -- c)

Compare the top two items on the parameter stack and return the smaller value.

ABS   (a -- a)

Return the absolute value of the top item on the parameter stack.

NEGATE   (a -- b)

Negate the top item of the parameter stack. Zero minus 'a' gives you 'b'.

INCDBL   (a b -- a b)

Add one to each the top item and the next item on the parameter stack.

DECDBL   (a b -- a b)

Subtract one from each the top and the next to the top items on the parameter stack.

INCMEM   (p --)

Increment long word the value that is pointed to by the item on the top of the parameter stack.

DECMEM   (p --)

Decrement long word the value that is pointed to by the item on the top of the parameter stack.

MEMORY

    @   (a -- b)

Load long 32 bits. Loads a 32 bit value from address 'a' onto the stack 'b'.

    @L   (a -- b)

An alternate version of @. Load 32 bits.

    @W   (a -- b)

Load 16 bits from the address on top of the parameter stack. Fill the high word of the stack item with 0.

    @B   (a -- b)

Load a single byte from the address on top of the parameter stack. Fill the top three bytes of the stack item with 0.

    !   (a b --)

Store long 32 bits. Stores 'a' at address 'b'.

    !L   (a b --)

Store long 32 bits. An alternate for !.

    !W   (a b --)

Store the low two bytes at the address on top of the param stack.

    !B   (a b --)

Store the low byte of the second item at the address given by the top item.

    CWL   (a -- b)

Convert word to long word by doing sign extension.

    +!   (a b --)

Add word at address. Add word 'a' to the value at address 'b', leaving the result at address 'b', replacing the original value there.

    -!   (a b --)

Subtract word at address. Subtract word 'a' from value at address 'b' leaving result there replacing the original value.

STRINGS

CMOVE (a b c --)

Copies 'c' bytes beginning at address 'a', to address 'b'. The move proceeds toward high memory. If 'c' is zero, nothing is moved.

CMOVE> (a b c --)

Copies 'c' bytes beginning at address 'a', to address 'b'. The move proceeds toward low memory. If 'c' is zero, nothing is moved. This can be used to move bytes upward in memory when the destination overlaps the source.

FILL (a b c --)

Fills 'b' bytes of memory beginning at address 'a' with the least significant byte of 'c'. 'b' is an unsigned number of bytes and no action is taken if 'b' is zero.

Also see BLANK and ERASE.

COUNT (a -- b c)

This returns the number of characters in a string at address 'a'. 'b' is address 'a' plus one (start of text). 'c' is the length of the text.

-TRAILING (a b -- a c)

Adjusts the character count 'b' of the text string beginning at address 'a' to produce a count 'c' that excludes trailing blank spaces. The address 'a' remains unchanged.

BL (-- sp)

Return a single space character on the parameter stack.

BLANK (pnt cnt --)

Fill an area of memory specified with ascii spaces. Like BL FILL

ERASE (pnt cnt --)

Fill an area of memory specified with zeros. Like 0 FILL

ASCII (--)

Compile the next non space character from the input stream as a single byte literal.

$CONSTANT

Create a named string constant. The string starts with the first character after the space that delimits the name of the string constant.

$VARIABLE

Create a named string variable and set its initial length to 0. The top item on the parameter stack will determine how much space is allocated. All strings are padded with an extra byte if needed to end on an even boundary.

.$ (str --)

Print a string.

!$ (str1 str2 --)

Copy the first string to the second string. No check is made for overrun or even if the second address is that of a string.

$" (-- str)

Create an un-nammed string literal.

$POS (str1 str2 -- str)

Find the position if any where the first string is contained in the second string. If not found return 0.

CHMATCH (chr str -- adr)

Search the string for the given character. If not found return 0.

NUMERIC CONVERSION

BASE (-- adr)

The address of BASE.

DECIMAL (--)

Puts decimal 10 in the variable BASE and selects decimal notation for input and output.

HEX (--)

Puts 16 in the variable BASE and selects hexadecimal notation for input and output.

OCTAL (--)

Puts 8 in the variable BASE and selects octal notation for input and output.

CONVERT (a b -- c d)

Converts an uncounted ASCII string of digits to a number on the stack using the value in BASE. 'c' is the result of converting each digit or character in the string beginning at address 'b'+1 into a number and accumulating each number in 'a' after multiplying 'a' by the value contained in BASE. ('a' is normally 0). Conversion continues until an unconvertable character is found. The address of that character is left in 'd'.

<# (--)

Starts the conversion of an unsigned number to a formatted (pictured) output string.

<S#   (--)

Starts the conversion of a signed number to a formatted (pictured) output string.

(a -- b)

The right digit of 'a' is converted to an ASCII character according to the value of BASE. It is then appended to a formatted (pictured) output string for subsequent output by TYPE. 'b' is the number of remaining digits and is retained for further processing. Used between <# and #>.

#S   (a -- b c)

Converts 'a' to a formatted (pictured) output string. This is done digit by digit according to the value in BASE. A single zero is placed in the output string if 'a' was initially zero. Used between <# and #>.

HOLD   (a --)

Inserts a character with the ASCII value 'a' into a pictured numeric output string. Used between <# and #>.

HLD   (-- a)

Return the address of the variable that holds the pointer used by HOLD.

SIGN   (a --)

Appends an ASCII minus sign to the beginning of a formatted (pictured) numeric output string if 'a' is negative. Use between <# and #> is optional.

#>   (a -- b c)

Ends the conversion of a number to a formatted (pictured) output string. 'b' is the address of the output string and 'c' is the number of characters in it.

SSIGN   (--)

Save the sign of the number being formatted.

FILES / KOS

These words are provided to make it easy to use the files and other services of K-OS ONE with a FORTH program. The philosophy of HT-FORTH is to work with the operating system and not try to replace it as some FORTHs do. These are the basic routines. More complex words are easy to define because of the simplicity of K-OS ONE.

The routines expect strings to be in counted form like most other FORTH programs. Most of the routines will append a nul to any string used. When a parameter block is set up it must start on an even boundary.

MAKFIL (strng -- stat)

Makes a new file or erases an old file. The parameter stack contains the address of a counted string.

OPENFIL (strng -- chan)

Open a file or device for subsequent use by the program. The channel opened is returned on the stack.

CLOSEFIL (chan --)

Close a channel to a file or device.

READFIL (a b c -- c)

Read the requested number of bytes 'c', from the channel specified 'a', to the buffer specified 'b'. If the channel is from a terminal it will stop on the first carriage return found.

WRITEFIL (a b c -- c)

Write the specified number of bytes 'c', to the channel 'a', from the buffer 'b'. If the media becomes full the return count 'c' will be different from the requested count.

SEEKFIL (chan posit --)

Position the read/write pointer for a random access file. Any subsequent reads or writes will occur at that point in the file.

TRAP (a --)

This is a generic hook to all the K-OS ONE services. The top of the parameter stack 'a' contains the address of a parameter block as specified in the K-OS ONE programming manual. After the call, the status can be found in the parameter block.

PARBLK (-- a)

This returns the address of the parameter block used for the built in system calls. Check this after any file operation if more information is needed about the results of any file operation.

POINTER WORDS

For each of these words the address on the parameter stack is not the address of the data but is the address of a pointer to the data. The pointer is changed by each of these words. These are similar to the pointer operations available in the C language. In each case the pointer is modified by the size of the data loaded or stored.

    @B+ (adrs -- b)

Load byte using pointer then increment pointer.

    @W+ (adrs -- w)

Load word using pointer then increment pointer.

    @L+ (adrs -- L)

Load long using pointer then increment pointer.

    @-B (adrs -- b)

Decrement pointer and use it to load a byte.

    @-W (adrs -- w)

Decrement pointer and use it to load a word.

    @-L (adrs -- L)

Decrement pointer and use it to load a long.

    !B+ (b adrs --)

Use pointer to store a byte then increment pointer.

    !W+ (w adrs --)

Use pointer to store a word then increment pointer.

    !L+ (L adrs --)

Use pointer to store a long then increment pointer.

    !-B (b adrs --)

Decrement pointer and use it to store a byte.

    !-W (w adrs --)

Decrement pointer and use it to store a word.

    !-L (L adrs --)

Decrement pointer and use it to store a long.

    LNDX (adrs index -- adrs)

Calculate an address into a single dimensioned array of long items.

    WNDX (adrs index -- adrs)

Calculate an address into a single dimensioned array of word size items.

    BNDX (adrs index -- adrs)

Calculate an address into a single dimensioned array of byte sized items.

HT - FORTH
EDIT MANUAL

Table of Contents

Using the Editor 3-2

Edit Commands 3-3

Help Command 3-3

Move Cursor Commands 3-3

Delete and Restore 3-6

Find and Replace 3-7

File Operations 3-8

USING THE HT-FORTH EDITOR

To enter edit mode after loading the HT-FORTH editor, type EDIT.

```
     OK EDIT (return)
```

In the editor you will start with a clear screen. You can start a new file to be named when you save it or read in an existing file to work with.

To read in an existing file you use the ^KR command.

```
     ^KR  (prompts 'FILE NAME':)
     filename.ext
```

While using the editor, any time that you get the message: PRESS # TO CONTINUE, pressing any key will allow you to resume editing.

EDIT COMMAND SET :

```
-------- FOR HELP SCREEN ^J or Line Feed --------
 ^E - line up              | ^R - page up
 ^X - line down            | ^C - page down
 ^L - repeat last find     | ^M - new line
 ^D - char right           | ^G - del char
 ^S - char left            | ^Y - delete line
-------------------------------------------------
 ^QS - start of line       | ^QL - restore line
 ^QD - end of line         | ^QY - delete to
 ^QR - top of file         |         end of line
 ^QC - bottom of file      | ^KR - read file
 ^QA - find and replace    | ^KW - write file
 ^QF - find                | ^KD - exit editor
-------------------------------------------------
```

EDIT COMMANDS

The following is a description of each edit command that is supplied with the HT-FORTH editor in the edit toolkit.

In describing these commands, a ^ is used to indicate a control character, where the control key is held down while the character is pressed.

HELP COMMAND

^J HELP

This command causes a list of the edit commands to be desplayed on the console terminal. Pressing any key will restore the screen and allow you to resume editing.

LINE FEED HELP

This command acts the same as ^J

MOVE CURSOR

These commands are used to position the cursor on the screen.

```
                 ^E - line up
                 ^X - line down
                 ^D - char right
                 ^S - char left
                ^QS - start of line
                ^QD - end of line
                ^QR - top of file
                ^QC - bottom of file
```

^E Line Up

Moves the cursor to the line just above the line it was on. The cursor will remain in the same column regardless of the length of the line.

^X Line Down

Moves the cursor to the line just below the line it was on. The cursor will remain in the same column regardless of the length of the line.

^D Character Right

Moves the cursor to the right one character. The cursor will move past the last character and when it hits the right margin it will wrap around to column one on the same line.

^S Character Left

Moves the cursor to the left one character. When column one is reached the cursor will wrap around to the right margin.

^QS Start of Line

Moves the cursor to the first column of the line it is on.

^QD End of Line

Moves the cursor to the right of the last character on the line.

^QR Top of File

Moves the cursor to the first character at the top of the file. The screen will change to show the first screen of the file.

^QC Bottom of File

Moves the cursor to the last charactor at the bottom of the file. The screen will change to show the last screen of the file.

MOVE SCREEN - SCROLL

These commands are used to scroll through the text by displaying the next screen requested.

^R Scroll Up

Moves the display so the next screen up towards the beginning of the file is displayed.

^C Scroll Down

Moves the display so the next screen down towards the end of the file is displayed.

DELETE AND RESTORE

These commands are used to delete items from your file.

```
 ^G - delete character
 ^Y - delete line
^QL - restore line
^QY - delete to end of line
```

^G Delete Character

Deletes the character that the cursor is on.

BACKSPACE or DEL Delete Character Left

The Backspace or DEL (Delete) keys will cause the charactor to the left of the cursor to be deleted. If the last character on a line is deleted a blank line will remain. (^Y will remove a blank line).

^T Delete Word

Deletes the word to the right of the cursor, starting with the charactor the cursor is on. If the cursor is on a space between words, the space that the cursor is on and all of the spaces to the right will be deleted, up to the next word.

^Y Delete Line

Deletes the entire line that the cursor is on.

^QL Restore Line

Restores the line that the cursor is on to its prior state. A line that has had words or characters changed is restored to the way it was before any editing changes were made. (An entire line that has been deleted can not be restored because the cursor is no longer on that line.)

^QY Delete to End of Line

Deletes all characters from the cursor position to the end of the line.

FIND AND REPLACE

These commands are used for searching and search and replace.

```
^QF - Find
^QA - Find and Replace
^L  - Find / Find & Replace Again
```

^QF Find

Finds the specified text string and moves the cursor to that location. If requested string is not found, the cursor will remain at its original location.

^QA Find and Replace

Finds the specified text string and replaces it with the requested new text.

^L Find / Find & Replace Again

Find / replace text again. Repeats the previous find or find and replace command.

FILE OPERATIONS

The following commands are used for dealing with files.

```
^KR - read file
^KW - write file
^KD - exit editor
```

^KR Read a File

Read a file into memory. After you give the ^KR command, it prompts for the name of the file to read.

^KW

Write a file to disk. This saves the text are editing to a file name that you specify. After you give the ^KW command, it prompts for the file name.

^KD Exit Editor

Exit editor to operating system. The command will prompt: ** ARE YOU SURE ** ?. This is to give you a chance to save the file before exiting the editor. A 'Y' or 'Yes' response will cause you to exit the editor.

USING THE HT-FORTH ASSEMBLER

The word CODE selects the ASSEMBLER vocabulary and allows the user to define a word in assembly language. END-CODE or C; completes the definition and restores the FORTH vocabulary.

ADDRESSING MODE SYNTAX:

MOTOROLA	HT-FORTH
Dn	Dn
An	An
(An)	An)
(An)+	An)+
-(An)	An -)
d(An)	d An D)
d(An,Xi.W)	d Xi An DI)
d(An,Xi.L)	d Xi An LDl)
d(PC)	d DPC)
d(PC,Xi.W)	d Xi DIPC)
d(PC,Xi.L)	d Xi LDIPC)
Abs.W	n #)
Abs.L	n L#)
Imm	n #

WHERE...

Dn	is a data register D0..D7
An	is an address register A0..A7
Xi	is a general register D0..D7, A0..A7
d	is a number
n	is a number

NOTE:

The mode words D) , DI) , DPC) and DIPC) all suggest the order of the arguments on the stack. The Displacement is always the first argument, an Index register may be the next argument, which may be followed by an address register, and the mode word is the last argument.

INSTRUCTION ARGUMENT ORDER:

TYPE	ARGUMENTS	MNEMONICS
imm	n # ea	ADDI ANDI CMPI EORI ORI SUBI
immsr	n #	ANDI>SR EORI>SR ORI>SR
immccr	n #	ANDI>CCR EORI>CCR ORI>CCR
iq	n #	ADDQ SUBQ
ieaa	ea An	ADDA CMPA LEA SUBA
isr	Dn Dn	ASL ASR LSL LSR ROL ROR ROXL ROXR
	n # Dn	
	ea	
ibit	Dn ea	BCHG BCLR BSET BTST
	n # ea	
ibra	label	Bcc
idbr	Dn label	DBcc
iset	ea	Scc
move	ea ea	MOVE
moveq	n # Dn	MOVEQ
move usp	An	MOVE<USP MOVE>USP
movem	n # ea	MOVEM< MOVEM>
movep	Dn d An D)	MOVEP
	d An D) Dn	
cmpm	An)+ An)+	CMPM
exg	Rn Rn	EXG
ext	Dn	EXT
swap	Dn	SWAP
stop	n #	STOP
trap	n #	TRAP
link	n # An	LINK
unlk	An	UNLK
eor	Dn ea	EOR
cmp	ea Dn	CMP
ibcdd	Dn Dn	ABCD ADDX SBCD SUBX
	An -) An -)	
idea	ea Dn	ADD AND OR SUB
	Dn ea	
iead	ea Dn	CHK DIVS DIVU MULS MULU
iea	ea	JMP JSR MOVE<SR MOVE>CCR MOVE>SR NBCD PEA TAS
ieas	ea	CLR NEG NEGX NOT TST
icon	(none)	NOP RESET RTE RTS

WHERE...
  Rn   is a general register
  ea   is an effective address
    (see addressing modes above)
  label is an address

NOTES:

Instructions which operate on more than one data size will be coded according to the state of the SIZE variable.

SIZE is set by the words BYTE WORD and LONG.

It is a good idea to set the size before every instruction for which size is relevant.

Beware, entering a DO loop sets SIZE to LONG.

STRUCTURED CONDITIONALS SYNTAX:

STRUCTURE	SYNTAX
if statement	condition IF true_code ELSE false_code ENDIF
while loop	BEGIN condition WHILE body REPEAT
until loop	BEGIN body condition UNTIL
forever loop	BEGIN body AGAIN
for loop	count Dn DO body LOOP

WHERE...

condition	is the code required to set the condition flags followed by one of these branch instructions: 0= 0<> 0< 0>= < >= <= >
true_code	is code to be excuted if the condition is true
ELSE	is an optional word
false_code	follows ELSE and is executed if the condition is false
body	is the body of the loop
count	is the number of times the body will be executed
Dn	is the data register used to hold the loop count

EXAMPLES:

```
BEGIN A6 ) LONG TST 0<> WHILE
  1 # A6 ) LONG SUBQ
REPEAT

( this loop decrements the top of stack while
  it is not zero )
```

EXAMPLES (continued)

```
 4 A6 D) D1 LONG MOVE
 D0 DO
   1 # D1 LONG LSL
 LOOP
 D1 A6 ) LONG MOVE

 ( this code multiplies the next item in the
   stack by 2 raised to the top-of-stack power
   and consumes the top of stack.  This can be
   done more efficiently without a loop as shown
   below: )

 A6 )+ D0 LONG MOVE
 A6 ) D1 LONG MOVE
 D0 D1 LONG LSL
 D1 A6 ) LONG MOVE
```

IMPORTANT WORDS IN THE ASSEMBLER VOCABULARY:

```
     #         A0        D)        ELSE
     #)        A1        D0        END-CODE
     )         A2        D1        ENDIF
     )+        A3        D2        IF
     -)        A4        D3        L#)
     0<        A5        D4        LDI)
     0<>       A6        D5        LDIPC)
     0=        A7        D6        LONG
     0>=       AGAIN     D7        LOOP
     <         BEGIN     DI)       REPEAT
     <=        BYTE      DIPC)     UNTIL
     >         C;        DO        WHILE
     >=        CODE      DPC)      WORD
```

HT-FORTH ASSEMBLER MNEMONICS:

```
ABCD        DBLS        ROL
ADD         DBLT        ROR
ADDA        DBMI        ROXL
ADDI        DBNE        ROXR
ADDQ        DBPL        RTE
ADDX        DBRA        RTS
AND         DBT         SBCD
ANDI        DBVC        SCC
ANDI>CCR    DBVS        SCS
ANDI>SR     DIVS        SEQ
ASL         DIVU        SF
ASR         EOR         SGE
BCC         EORI        SGT
BCHG        EORI>CCR    SHI
BCLR        EORI>SR     SLE
BCS         EXG         SLS
BEQ         EXT         SLT
BGE         JMP         SMI
BGT         JSR         SNE
BHI         LEA         SPL
BLE         LINK        ST
BLS         LSL         STOP
BLT         LSR         SUB
BMI         MOVE        SUBA
BNE         MOVE<SR     SUBI
BPL         MOVE<USP    SUBQ
BRA         MOVE>CCR    SUBX
BSET        MOVE>SR     SVC
BSR         MOVE>USP    SVS
BTST        MOVEM<      SWAP
BVC         MOVEM>      TAS
BVS         MOVEP       TRAP
CHK         MOVEQ       TST
CLR         MULS        UNLK
CMP         MULU
CMPA        NBCD
CMPI        NEG
CMPM        NEGX
DBCC        NOP
DBCS        NOT
DBEQ        OR
DBF         ORI
DBGE        ORI>CCR
DBGT        ORI>SR
DBHI        PEA
DBLE        RESET
```

DUMP AND OTHER UTILITIES

The DUMP utility is used to display blocks of memory.

```
    DUMP      ( a b -- )
```

DUMP 'b' bytes of memory starting at address 'a'.

```
    DU        ( adrs1 -- adrs2 )
```

DUMP 64 bytes of memory starting at 'adrs1'. 'adrs2' is 'adrs1'+64.

There are other useful utilities in the DUMP file. Reading the source code for this file will show you what these utilities are and how they can be used.

STRUCTURE OF HT-FORTH

On the surface, HT-FORTH is like any other FORTH. You need to be aware however that many FORTH programs make significant assumptions about how the language is implemented. In particular, many assume that FORTH will be a threaded language in a real address space and make modifications to the dictionary. HT-FORTH is compiled and it is position independent.

The detailed workings of each FORTH word are described by comments in the assembly listing and each word is very small. Only the parts that are not obvious after reading the comments will be explained in detail here. The main parts that need to be described will be the data structures.

The kernal of the HT-FORTH is written in conventional 68000 assembly code. To change the system you need to edit the assembly source using any text editor and then reassemble it using the standard K-OS 68000 assembler or any compatible assembler. After you have reassembled the kernal and converted it to binary (using HEX2BIN), load all of the definitions that were written in FORTH and save the system using SAVE.

Before adding new words, a backup copy should be made of the unmodified FORTH system. When adding words that are written in FORTH, it is possible to add new words and save the system in a live mode. This allows the FORTH to be restarted with the old definitions in place. Because standard .BIN executable files on K-OS ONE are assumed to be position independent it is important not to use certain features if you intend to move the result to different hardware running the K-OS ONE system.

The source code is in four major files. The first part has the main structure and the storage definitions. The other parts have the rest of the standard words that are written in assembly language. For some applications many of these words could be deleted. An effort was made to keep the words in a very regular format so that it would be easy to add additional words in a convenient manner or to delete unused words.

If new words are added be sure to change the starting word that is part of the FORTH vocabulary. Currently it is DLX200. This number sequence was chosen to make it easy to avoid label conflicts when adding words.

An important feature of HT-FORTH is that it is position independent. As new words are added to the basic system, they stay position independent also. As a result, it is important to distinguish between real addresses and relative addresses. When data is being accessed, a real address is usually used. When a routine is being accessed, a relative address is usually used. This results in a change in the way several of the words are defined. The base address for HT-FORTH is A5. It is set when FORTH is loaded.

[']	- compiles a relative address
'	- return relative address of code field of next word
COMPILE	- compile the next word when executed
[COMPILE]	- makes the next word normal, not immediate
EXECUTE	- execute the word whose relative address is on parameter stack
FIND	- find relative code address
REL	- make an absolute adrs into a rel adrs
UNREL	- make a rel adrs into an absolute adrs

DICTIONARY

The dictionary of HT-FORTH is similar to that of other FORTHs. The major difference is that it is compiled to JSR/BSR or macros and is not indirect threaded. The result of this is that the fields are not the same. In particular there is no parameter field. And any information is direct code and not just a refrence to code routines. Because of the macros and the way words are compiled the same word may compile to different code each time it is used.

LINK	- 4 bytes, relative to A5
NAME	- 1 byte size, first 7 characters
TAG	- 2 bytes, has info about macro, immediate
CODE	- the code to execute

The dictionary is a linked list. The variable LAST has the real address of the last directory entry. Each link points to the prior entry except the last link that is 0 to indicate the end of the dictionary. The links are 32 bits long so there is no limit on the size of defined item. This could easily be changed to 16 bits to save space.

The name is encoded like FIG-FORTH with a length byte and the first seven letters of the word. It would be easy to change this to use the first 3 or 5 letters.

TAG WORD

After the link and name, there is a tag word. Bit 14 is the macro flag. If this bit is a 1 it means that when it is compiled, the definition is copied into the word that is being compiled, not just referenced with a BSR or JSR. The low byte of the tag word is a count of how many words to copy when the word is compiled. This results in faster code because of the reduction of overhead by not having a BSR and RTS. In many cases it also results in slightly larger code. Do not use BSR or other relative branches in any macro definitions if the destination is not an integral part of the definition.

All macros end with an RTS so they will operate properly in interactive mode. The RTS must not be copied as part of the macro when it is compiled. If the RTS is copied it will act like the word EXIT.

Bit 13 is the immediate flag. If it is a 1 then the word is immediate. In general this means that it does something while it is being compiled.

The FORTH kernal gets its input by calling an internal routine GETCX. This uses a source flag to determine if input will come from the terminal or from a disk file. If it is coming from a disk file it is handled slightly differently than if it is coming from the terminal. When input comes from the disk blocks of 512 bytes are read in and then processed directly, a special text input buffer is not used. Any linefeed characters that are found are converted to spaces. If a ^Z is found then it is assumed that the end of the file has been reached and the loading is terminated. Currently the LOAD from disk echos all characters to the console. This can be easily disabled.

When a line is input from the terminal to be interpreted a special string of characters is appended to the end of the line. The purpose of these characters is to make sure that WORD and (find something to terminate with even if an error is made.

When an application is locked it is fixed so that the FORTH language used to create it cannot be reached by the user of the program. This makes it a simple .BIN executable file like an application program done in any other language. Unless a hex dump is made of the resulting program it will not be possible for the user to tell that it was written in FORTH rather than some other language. If an application is locked then it can be sold without violating the copyright of Hawthorne Technology. If it is possible to define any new words then it cannot be sold without a licience.

EXAMPLE of LOCK:

```
          LOCK  VOCX  STRTW  PPP.BIN
```

LOCK	- the word that locks the application
VOCX	- the only vocabulary active in the locked application
STRTW	- the word that will execute when the locked program is started
PPP.BIN	- the binary file for the new program

When an error is found in the FORTH code being interpreted or compiled some question marks will be displayed. (???). When this occurs press a key on the console to continue. If you press return then the rest of that line will be ignored. If you press control C then FORTH will terminate and you will be back in the operating system. If you press any other key then the error will be ignored.