H U L L F O R T H

R E F E R E N C E   M A N U A L

A N D

S E L F   T E A C H I N G   G U I D E

Rev. 2, Aug. 1981

A.F.T.Winfield and
P.S.Cain.
148, Goddard Avenue,
HULL. HU5 2BP

Preface - Hullforth and Standard FORTH

Hullforth is an implementation of the computer language FORTH (1), invented by Charles H. Moore of the National Radio Observatory (U.S.A.), around 1970 (2), I first came across Forth in a small article in Dr. Dobbs (3), and was instantly attracted by the elegance and versatility of the language, so much that I immediately began writing an implementation - based solely on this article, and a copy of the Forth Interest Group 'Forth Handy Reference' (4). It was not until after completing Hullforth that I had access to the excellent FORTH Inc. manual 'Using Forth' (5) - and in the interests of standardisation I altered some Hullforth words so that the final Hullforth conforms to standard Forth, with a few exceptions. Most standard Forth programs will therefore run on Hullforth with little, or no modification.

A.F.T.Winfield,
Hull.
June 1981

(1) FORTH is a registered trademark of FORTH Inc. (U.S.A.)

(2) Moore, C.H. and Rather, E.D., "The FORTH program for spectral line observing." Proc. IEEE, vol 61, September 1973, and
-- , "FORTH: A new way to program a mini-computer.", Astron. Astrophys. suppl. 15 1974.

(3) James, J.S., "FORTH on microcomputers", Dr. Dobbs, No 26.

(4) "Forth Handy Reference", Forth Interest Group, P.O. Box 1105, San Carlos, Ca 94070, and U.K.-Forth Interest Group, c/o B.Powell, 16 Vantorts Rd., Sawbridgeworth, Herts.

(5) "Using Forth", 1979, FORTH Inc., available from COMSOL Ltd., 87 Briar Rd., Shepperton, Mddx.

0 Contents

1 Introduction

Hullforth is a new compiler/interpreter written for the NASCOM microcomputer, it is based upon the language 'FORTH'. Forth is, we believe, an important new programming language - it provides a simple to use, interactive, programming environment allowing complex software to be developed 'at the keyboard', but without the speed penalty of BASIC since the final program is truly compiled.

The unique combination of stack orientated, reverse Polish, integer arithmetic (up to 32 bit as standard), with high level structuring (DO .. LOOP, IF .. ELSE .. ENDIF, BEGIN .. UNTIL), means that programs are well structured, and readable yet the compiled machine code is efficient and fast - often almost as fast as purpose written assembler.

In addition the use of a vocabulary which is indefinitely extendable means that the programmer is not limited to the standard set of reserved words, but may define special words to suit his particular application. These then become part of the standard vocabulary, and are treated just like all of the other words.

Indeed the 'definition' of new words is a natural part of programming in Forth, since programs of more than a few lines in length are best broken down into small modules. A complex program is thus built by defining the innermost operations first, as new words in the vocabulary - and then further words are defined which use the previously defined words, until the final result is achieved. Programs are written, and tested, in small segments with the result that develop--ment and debugging times tend to be very short.

Forth is thus a language suited to almost any programming task, but particularly those applications requiring fast execution, which would normally have to be written in assembler (e.g. fast interactive games, I/O interfacing etc.). Both BASIC and machine code programmers find Forth a nice alternative, for all but the ultra time critical problems.

This manual is intended both as a reference for the Hullforth system, and, (for those with no experience of Forth programming) as a self-teaching guide to the use of Hullforth. Chapter 4 is the self-teaching guide, and it is recommended that this is read in conjunction with your microcomputer running

Hullforth, so that examples may be tried out as they occur in the text. Before embarking on this self-teaching course read chapter 2, 'Overview' and, of course, chapter 3 on loading and running your system.

2 Overview

All input to Hullforth consists of sequences of 'words'. A word is any sequence of non-space characters, and different words must be separated by spaces. Whenever the Hullforth system finds a word in the input stream, the 'dictionary' is searched, for that word. If the word is not found in the dictionary then Hullforth assumes that the word is a number. All words must then be either previously defined dictionary entries, or numbers, otherwise an error message will result. The dictionary is a linked list of words together with a sub--routine for each word. When a word is found in the dictionary, Hullforth either executes the subroutine, or compiles a call to the subroutine if the word is part of a new definition. The Hullforth dictionary initially contains about 150 words, and writing a Hullforth program actually consists of defining a set of new words using the standard dictionary (and previously defined new words).

When Hullforth is ready for user input, the prompt character "*" will be printed. Hullforth input is buffered so that no action will be taken on a line of input until carriage-return (or newline, on NASCOM) is pressed. The input:

```
42 200 + .
```

followed by carriage return will cause Hullforth to print 242, the result of adding the two numbers 42 and 200 (assuming base 10). This example illustrates the use of the postfix, (or reverse polish) notation, and the last-in first-out stack. In the example first the number 42 is 'pushed' onto the stack, then the number 200 is pushed onto the stack. "+" is defined as a standard Hullforth word, whose action is to 'pop' the top two numbers off the stack, add them together, and push the result back onto the stack. The word "." , also a standard Hullforth word, has the effect of popping the top number off the stack, and printing it, in this case - the result 242.

New words may be defined and added to the dictionary in a number of ways, the most useful of which is the colon (":") definition. For example, the input stream:

```
: TIMESTEN 10 * ;
```

defines a new word called "TIMESTEN". When executed TIMESTEN causes the words 10 and * to be executed as if explicitly stated.

So, typing:

```
27 TIMESTEN .
```

will cause the result 270 to be printed out.

Most non-trivial programs, or 'colon definitions' will be longer than one line, so Hullforth has the facility of inputting from a block buffer, or 'screen' containing 16 lines by 64 characters. Screens may then be edited, and saved onto, or loaded from backing storage.

3 Loading and running Hullforth

Hullforth is approximately 8K bytes, including stack and system variables - but excluding dictionary space which grows upwards from the program towards high memory. Hullforth occupies memory from 1000HEX upwards*, but the first 1K bytes contains the stacks, some variables, and the line input buffer. The actual program starts at 1400HEX. This is the cold start 'GO' address into Hullforth.

As soon as you have loaded Hullforth successfully, you should make a back up copy onto another cassette, by saving 1400HEX to 330AHEX. The program is entered at 1400HEX and the message:

HULLFORTH - VERSION 1.1
COPYRIGHT AFTW/PSC 1981
*

will be printed on your display terminal - Hullforth is then ready to accept input. If you should at any time wish to return into your machine monitor the Hullforth word "BRK" will effect this. To re-enter Hullforth without losing any new definitions you may have created, GO at 1448HEX. Go at 1400HEX clears the dictionary down to the standard words only.

The remainder of this chapter details the loading and running of Hullforth on specific systems. General details of re-configuring system pointers in any implementation (for, say, machines with greater than 16K RAM), are given in appendix 1.

3.1 Nascom I running NAS-SYS

The cassette will be supplied in T4 format, for NAS-SYS. It is loaded using the R command, and run by typing E 1400.

3.2 Nascom II running NAS-SYS

The cassette will be supplied in CUTS format, at 300 baud, for NAS-SYS. It is loaded using the R command, and run by typing E 1400. No modifications are necessary to run in NAS-SYS 1 or 3.

3.3 Nascom I running NASBUG

The cassette will be supplied in the appropriate format, to run under NASBUG. It is loaded using the R command, and run by typing E 1400.

* see appendix 1 for a detailed memory map.

3.4 Other Z80 based systems

For non-NASCOM systems Hullforth will be provided on cassette in standard CUTS (or, KANSAS CITY) format, at 300 baud. The data is contained in the following format:

Data is contained in blocks, each of 256 bytes, except the last block which may have less: the format of each block is as follows,

00	Null (0).
FF FF FF FF	Four start of block characters (FFHEX).
SS SS	Start address, low order first.
LL	Length of data (00=256 bytes).
BB	Block number, this is one less for each block. The last block is block 00.
CC	Checksum for the header data (mod 256 sum of bytes).
DD DD	Data.
EE	Checksum for the data.
00 00	Ten nulls.

Before running Hullforth you must then alter the following to suit your own system.

3.4.1 Character Input

Hullforth requires two character input routines, one which waits for a keyboard entry, and returns the ascii for the character in the A register. The other character input routine must check the keyboard to see if a key has been pressed - without waiting. The A register must contain zero if no key was pressed, or the ascii if a key had been pressed.

Both routines must preserve all registers (except A, of course!). Once you have established that you have these routines, (or written them, if not), then insert the following code into Hullforth:

For character input with 'wait'

```
1BAB   CD aa aa    CALL aaaa
1BAE   C9          RET
```

and for character input, without wait,

```
1BB2   CD bb bb    CALL bbbb
1BB5   C9          RET
```

where aaaa and bbbb are the addresses of your own routines.

3.4.2 Character Output

One character output routine, which simply prints the character in A, is required. The routine must again preserve all registers, and is inserted into:

```
1BB9    CD cc cc    CALL cccc
1BBC    C9          RET
```

3.4.3 Return to Monitor

To be able to use the BRK word you must first insert the code to cause a jump into the entry point of your own monitor at:

```
2363    C3 dd dd    JP dddd
```

3.4.4 Cassette Save

In order to use the Hullforth cassette load and save commands, SSAVE, SLOAD, VSAVE, VLOAD, and CATALOG you must alter the following port and status bit references:

2E2B ee and 2E85 ee

where ee is your UART data port number,

2E34 ff and 2E61 ff

where ff is your UART status port number,

2E3D gg

where gg is a mask to suit your transmit buffer ready flag in the UART status byte. For example, if TBR is bit 6, gg = 40 HEX.

2E6A hh

where hh is another mask to suit your data ready flag, in the UART status byte.

Finally, to use the Hullforth command FORTH, requires a call to your own systems cassette save routine. (The FORTH command saves the entire system including your own new definitions, for re-loading by your own system monitor.) Insert a call to your routine at:

```
2DF3    CD ii ii    CALL iiii
```

Your routine must save the memory from 1400HEX to the address stored at 15C2HEX.

When you have made all of the above modifications you should, of course, save the entire new system, from 1400HEX to 330A HEX on another cassette. These are the only system dependant elements in Hullforth.

This chapter is not an exhaustive description of Hullforth, but instead attempts to demonstrate the main techniques of programming in Hullforth - using only a subset of Hullforth words. For a detailed and complete description of the terms (e.g. dictionary, word etc.) used in this and other chapters, see the Hullforth glossary of terms, appendix 2.

4.1 Numbers and Arithmetic

After loading, and running Hullforth, the sign on message and the prompt character should be printed. Try typing a number, say, 42. (Your input is shown underlined and the carriage return, or newline at the end of each line is assumed).

```
*42
*
```

You will get another prompt, and apparently nothing has happened. But actually something has happened; the number 42 has been converted into 16 bit binary, and pushed onto the stack. The word "." has the effect of popping the top number off the stack and printing it,

```
*.
42*
```

So, numbers are pushed onto the stack whenever they appear in an input stream (except in a colon definition - but more about that later). All Hullforth arithmetic words (and indeed most others) pop their arguments off the stack, and push the result back onto the stack, so if you type,

```
*42 -23 + .     (don't forget the spaces)
```

you get the result, 19*

If you try typing "." with nothing in the stack you will get an error message,

```
NSU ERROR IN .
```

This is the most common error in Hullforth, and means 'normal stack underflow'. The error is produced whenever a Hullforth word needs more arguments than the stack actually contains. This and the rest of the Hullforth error messages are listed in detail in appendix 4.

If you should realise that you have made a typing error use the 'backspace' key to step back and correct the error. If you should wish to abort the whole line and start again, hit the 'escape' key (shift-newline on NASCOM), and you will return to the prompt, without executing any of your aborted line of input.

It is worth noting that all input to Hullforth is buffered - no action is taken until return or newline is hit.

Hullforth, as indicated earlier, uses a postfix notation for all arithmetic so a complex arithmetic expression must first be converted from 'infix', the normal arithmetic notation, into 'postfix', or reverse Polish. For example the expression,

((100*2)/4) + (100/2)

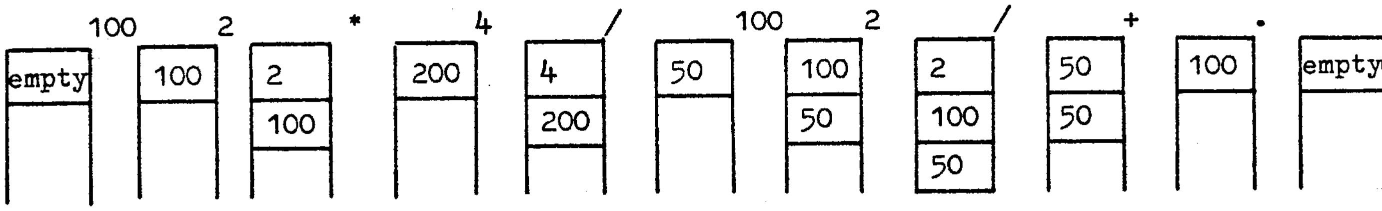
is, in postfix, 100 2 * 4 / 100 2 / +

and this is exactly how the expression is written in Hullforth, not forgetting the spaces between each number, and word. So,

*<u>100 2 * 4 / 100 2 / + .</u>

produces the result, 100*

Let us examine the stack after each word, to see how this works,

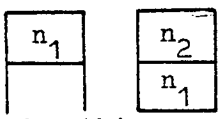


Note that the result of the first sub-expression, ((100*2)/4), is stored on the stack while the second is being evaluated.

This principle, of saving intermediate results on the stack may be extended to entire Hullforth programs, so that normally very few variables ever have to be used. Programs which use few variables will of course run much more quickly than those which use many variables - even if the programs do the same job.

The stack, then, is used for more than just arithmetic, and to make the stack as flexible as possible Hullforth provides a set of stack manipulation words. These do not change any of the numbers on the stack, but either duplicate numbers, or rearrange their order on the stack. One example is DUP, which duplicates the number on top of the stack, i.e.

DUP



where $n_2 = n_1$

A shorthand notation for this graphical picture of the stack before and after, is used throughout this manual, and for the word DUP, the shorthand is, $(n_1 \text{ -- } n_1\ n_2)$

This notation is so useful that all Hullforth words are described using it.

When writing Hullforth programs it is important to keep track of the stack contents after each word, (in the same way that writing programs in algebraic languages like BASIC or PASCAL requires keeping track of the contents of variables). The notation is again strongly recommended for this. For example, if we require 2 times, and 4 times a number, say 27, in Hullforth,

```
*27 DUP 2 * . SP 4 * . CR
54 108
*
```

This simple program may be explained as follows,

word	(stack before -- stack after)	
27	(-- 27)	
DUP	(27 -- 27 27)	
2	(27 27 -- 27 27 2)	
*	(27 27 2 -- 27 54)	
.	(27 54 -- 27)	output: 54
SP	no effect	output: space
4	(27 -- 27 4)	
*	(27 4 -- 108)	
.	(108 --)	output: 108
CR	no effect	output: newline

Notice the use of SP and CR to provide some rudimentary output formatting.

All of the examples I have shown so far have been in decimal. In fact Hullforth will accept numbers in any radix from 2 to 36, and print them in any of these radices also. The convention of using letters of the alphabet to represent digits above 9 is adopted (as in standard HEX notation), thus the maximum radix of 36 (0-9, A-Z). The usefulness of base 36 is doubtful! But certainly binary, decimal, octal, and hexadecimal are all useful. The standard Hullforth dictionary provides words to change the base into HEX, or DEC. So,

```
*255 HEX .
```

produces, FF*

You are now interpreting numbers as hex (base 16), so typing,

```
*FFFF DEC .
```

produces, -1*

Notice that the current base only affects input and output of numbers. Internally all arithmetic is binary.

Normally numbers will be converted into 16 bit binary, but if a number includes one or more of a set of punctuation characters (.,/:-), Hullforth will convert it into 32 bit binary, and push two 16 bit quantities onto the stack (upper half first). So, for example,

*HEX 4/06/81 DEC

has the effect of pushing 00040681HEX onto the stack, and typing,

*U.

prints, 40681*

But ideally number formatting would be used, to re-insert punctuation on output. See chapter 5.7.

4.2 Defining New Words

I have indicated already that, although variables are not often used in Hull-forth programming - the facility does nevertheless exist. Variables are not implicit, as in BASIC, but must be declared (or, more accurately - defined), before they can be used. If you wish to create a variable named FRED, say, with initial value zero, type,

*0 VARIABLE FRED

*

The effect of the Hullforth word VARIABLE is to create a dictionary entry named FRED within which two bytes are reserved. These two bytes are then set to the initial value (taken off the stack). When subsequently the word FRED is executed, the address of the variable will be pushed onto the stack. Try it,

*FRED .

will print the address of FRED. To print the value of FRED, use the Hullforth word @. This replaces an address by the (16 bit) number at that address. So,

*FRED @ .

prints, 0*

To set FRED to a new value use the word "!" ; So,

*-1 FRED !

is directly equivalent to LET FRED = -1 in algebraic languages. -1 pushes the value -1 onto the stack, FRED pushes the address of FRED, and ! stores the value at the address. Try printing the value of FRED again to check that it has changed.

Expressions involving variables may now be written, so that, for example, the algebraic expression, FRED = FRED * 10 would be written in Hullforth,

FRED @ 10 * FRED !

As in all new word definitions in Hullforth, there are no restrictions on either the number of characters, or the characters themselves in the variable name. All characters are significant, and checked when the dictionary is being searched so, for example, if

0 VARIABLE TESTVARIABLE0
1 VARIABLE TESTVARIABLE1

are defined, the whole of each variable name must be written to reference the variable, and the two variables shown in the example will not be confused by Hullforth.

Of course if you define a new variable with the same name as a previously defined variable, the new variable will supercede the old one, and since the dictionary is searched backwards, the newer definition will always be used. This is true not just of variables but of any defined word, since all are dictionary entries.

CONSTANTS may be defined in Hullforth, e.g.

```
*100 CONSTANT ONEHUNDRED
*
```

In this case the word ONEHUNDRED, when executed will push not an address, but the actual value, 100, onto the stack. CONSTANT's are then unchangeable.

The defining words VARIABLE and CONSTANT (and ARRAY, which is a special type of VARIABLE), have the effect of adding new words to the dictionary which will have a special action when executed, as described above. The 'colon definition', however, allows new words to be defined, but does not restrict the action of the words, when executed. When a new word is defined using a colon definition the action of the word is specified within the colon definition.

Consider the example given in chapter 2,

```
* : TIMESTEN  10 *  ;      (the extra spaces are
*                                 cosmetic only!)
```

This will define a new word TIMESTEN which has the effect of multiplying the number on top of the stack by 10. Try it,

```
*27 TIMESTEN .
270*
```

In all of the previous examples Hullforth has executed each word, in the input line as it found it - acting then as an interpreter. In the above 'colon definition' example the word ":" has the effect of switching Hullforth into a 'compile' mode. Hullforth remains in compile mode until finding the word ";", which ends the colon definition and switches Hullforth back into 'interpret' mode. A colon definition must then include the words : and ; which effectively start, and end the definition respectively. A missing semi-colon will result in an abandoned compilation and a 'COL' error message. In addition there must be a name following the colon - this will be the new word in the dictionary, and between the name and the semi-colon is the 'body', the actual Hullforth program to be compiled as the subroutine in the dictionary entry.

More useful than TIMESTEN are OCT and BIN,

```
*DEC
*: OCT  8 BASE !  ;
*: BIN  2 BASE !  ;
```

Notice the precaution of making sure that the current base was decimal before attempting the definitions! The word BASE is a system variable containing the base used by the input and output routines, so after OCT is executed Hullforth will expect all input numbers to be in octal. BIN similarly will cause the base to change to binary, when executed. Try this,

```
*DEC 255 DUP BIN . CR DUP OCT . CR HEX . CR DEC
11111111
377
FF
*
```

It is strongly recommended that more complex colon definitions are written as 'screens', so that errors may be edited out without retyping the whole definition and of course the (commented) source may be saved for future reference.

4.3 Branching and Looping

The colon definitions illustrated so far have all been very simple programs consisting of a sequence of operations. More complex programs require branching and looping. To be more specific, 'structured' programming requires,

i) The ability to execute a sequence of operations, one after the other.

ii) Conditional testing, and the execution of either one sequence, or another sequence depending on the result of the conditional test.

iii) Repetitive execution of a sequence of operations, until some condition is met.

Hullforth provides each of these requirements, the first has already been illus--trated, but consider now the second, conditionals.

A word which tests for the sign of a number, and prints a message might be defined as follows,

```
*: SIGN?  0 < IF ." NEGATIVE" ENDIF CR  ;
*
```

To test this type a number followed by SIGN?,

```
*300 SIGN?
*
```

or,

```
*-327 SIGN?
NEGATIVE
*
```

A refinement of this would be the inclusion of an ELSE clause,

```
*: SIGN?  0 < IF ." NEGATIVE" ELSE ." POSITIVE" ENDIF CR  ;
*
```

The new definition of SIGN? now supercedes the old, so if you type,

```
*0 SIGN?  -1 SIGN?
POSITIVE
NEGATIVE
*
```

To examine this new definition of SIGN?; upon execution the value 0 is first pushed onto the stack. The word < then compares the top two values on the stack and tests for the second value less than the first (top) value, which is in this case zero. The two values will be popped off the stack and a single boolean value pushed onto the stack, 'true' if the second value was less than the top value, 'false' otherwise. In this case since the top value was zero, we are in effect testing for 'negative'. The word IF, on execution, pops the (boolean) value off the stack and causes the words immediately following to be executed if the boolean was true (non-zero), or those following the ELSE if the boolean was false (zero). In either case execution then continues after the ENDIF.

In this example both the 'true' words, and the 'false' words are print state--ments, ." MESSAGE" . Upon execution the MESSAGE terminated by " is printed. The space following the ." must be included otherwise ." would not be recog--nised as a word.

Two further points are worth noting, as asides,

i) The sequence of words 0 < , is so useful that a Hullforth word has been predefined to have the same effect (0<).

ii) You now have two definitions of SIGN? in the dictionary, although the most recent will always be used. However you can type FORGET SIGN? to get back to the earlier definition.

To summarise the overall structure of conditionals, the following diagram may be helpful,

```
conditional test
IF
   'true' words
ELSE
   'false' words
ENDIF
```

The conditional test must place a boolean value on the stack, and usually involves 'comparison' words (see section 5.5), but not necessarily, since to test, for example, for a number being non-zero requires no comparison. A non-zero number is, after all, 'true' if treated as a boolean.

The 'true' words, or 'false' words may of course be any sequence of Hullforth words, and include further conditionals. In this way IF statements, and indeed looping statements may be nested to (virtually) any depth.

Conditional structures (and looping structures) may only be used inside colon definitions, the reason for this is that such structures contain forward 'jumps' which are not known until the whole program has been compiled. To achieve 'nesting' the jump addresses must be pushed onto a stack during compilation, but the 'normal' stack, (refered to up to now as simply 'the stack') is not used for this.

A second stack called the 'return' stack is used for compilation of conditional and looping structures, and is also used by looping words during execution, as shall be explained shortly. The return stack is accessible by the Hullforth programmer and may be used for temporary storage of values (using the words >R and R>).

The third of the 'structured' programming devices may now be considered, repetition. Hullforth provides two different looping constructions, DO .. LOOP and BEGIN .. UNTIL.

The DO .. LOOP structure is used primarily when the number of loops is known, or calculated before the loop is entered. For example,

```
*: COUNT  DO  I . SP  LOOP  ;
*
```

The word DO when executed, pops two values off the normal stack, and pushes them onto the return stack. These values are used as the index and end values during the loop. So typing,

```
         *5 1 COUNT
gives,   1 2 3 4 5 *
```

The word I, in COUNT, has the effect of pushing a copy of the number on top of the return stack onto the normal stack, and since the top value on the return stack is the value incremented by LOOP, the words I . will print this value each time round the loop. LOOP causes a jump back to the words following the DO, until the value on top of the return stack, the 'index' becomes greater than the second value on the return stack, the 'end' value. When it does, LOOP finally clears the top two values off the return stack, and execution continues of the words following the LOOP.

Notice that the arguments for DO are supplied in reverse order; end, start, DO.

An example of nested DO loops is the definition of TIMESTABLE,

```
         *: TIMESTABLE 10 1 DO  10 1 DO  I J * 4 .R  LOOP CR  LOOP ;
So       *TABLE
prints,    1   2   3   4   5   6   7   8   9  10
           2   4   6   8  10  12  14  16  18  20
           3   6   9  12  15  ... etc.
         *
```

The word J has the effect of pushing a copy of the third number on the return stack onto the normal stack. This is then the index value for the next outer--most DO loop, in a nested program. I and J are then specially useful in DO loops.

In TIMESTABLE the inner DO loop is repeated 10 times, and the code I J * 4 .R repeated 100 times. I and J pick up the inner and outer loop indices, which are multiplied, and each result printed right justified in a field width of 4 characters (4 .R). Thus we have a ten times table!

DO loops are then very simple to use, and the action of the return stack during execution need not concern the Hullforth programmer - provided that he ensures that the return stack is not affected by the words inside the DO loop.

Do loops are always executed at least once, but they may be terminated pre-maturely using the word LEAVE. LEAVE, usually inserted in a conditional inside a DO loop, has the effect of setting the top value on the return stack equal to the second (end) value, so that on the next execution of LOOP, the loop will be ended. For example,

```
                *: COUNT  DO I . SP  ?TERMINAL IF LEAVE ENDIF  LOOP ;
                *1000 1 COUNT
will print,     1 2 3 4 5 6 7 8 9 10 11 ... etc.
```

until <u>either</u> the value 1000 is reached, <u>or</u> any key on the keyboard is pressed.

In this definition of COUNT, the word LEAVE is <u>only</u> executed if ?TERMINAL returns a value 'true'. ?TERMINAL checks the keyboard to see if a key has been pressed, and returns the non-zero value of the key - if one has, or zero otherwise, and non-zero is of course the same as 'true'.

An alternative COUNT to the one shown above might be,

```
                *: COUNT  BEGIN  DUP . SP 1+  ?TERMINAL UNTIL  DROP ;
and,            *1 COUNT
will print,     1 2 3 4 5 6 7 8 9 10 ... etc.
```

indefinitely, unless a key is pressed. (Actually if no key is pressed the count will eventually terminate with an AOV error).

In this example the word BEGIN has no effect upon execution (it is needed only for compilation). DUP duplicates the value on top of the stack (initially 1), . prints it, and 1+ increments the original value. The counting is thus done on the normal stack here. The word UNTIL, on execution, pops the top (boolean) value off the stack, and causes a jump back to the words following the BEGIN if the value is 'false', or continues execution after the UNTIL if the value is 'true'. The loop thus continues indefinitely <u>until</u> 'true' at UNTIL.

In COUNT this will only occur when a key has been pressed.

It is worth noting the inclusion of the word DROP, after the UNTIL, to clear the counting value off the stack at the end. It is generally good practise to 'clear out' stacks of any unwanted results, at the end of a program.

4.4 Screens and Editing

Although a maximum of 128 characters may be typed in on one line, re-typing lengthy definitions is somewhat tedious, and it is generally much better practise to enter (all but the simplest) definitions, into the system, as 'screens'. A screen is a 1024 character block buffer which may be written using the editing commands, and then passed through Hullforth as if it were an ordinary line of input. The screen may contain any Hullforth input, but generally screens are most useful for developing colon definitions.

Screens are organised, for editing, into 16 lines of 64 characters each, the lines numbered from 0 to 15. The word NEW will clear out the current screen, (set by SCRO or SCR1), and start a prompting sequence to enter new text. Try entering the TIMESTABLE example,

```
*SCRO NEW
0 ( TEST SCREEN )
1 DEC
2 : TIMESTABLE  ( PRINT TEN TIMES TABLE )
3   10 1 DO
4        10 1 DO
5                I J * 4 .R
6                LOOP
7        CR
8        LOOP ;
9*
```

The sequence is exited at line 9 by typing escape (shift-newline on NASCOM), otherwise the sequence would continue until line 15. Now type LI to list the whole screen, or 0 8 LI to list just the first 9 lines.

Note the following details in the example screen,

i) A title has been included as comment (between brackets), on line 0. It is a good idea to use this title as a file name when saving the screen to cassette.

ii) The word DEC, on line 1 will ensure that when TIMESTABLE is compiled the base will be decimal. This avoids confusing errors if you happen to have been working in another base.

iii) Nested structures have been indented for readability, the intervening spaces will have no effect when the screen is passed through Hullforth.

To pass the screen through Hullforth type EXEC, and DEC will be executed, TIMESTABLE will be compiled.

To edit the screen use the editing words RP, to replace a line, INS to insert a new line between two others, and DL to delete lines. Do not use the word NEW again unless you want to clear out the screen, and start again.

When you have completed the screen, try saving it onto cassette. Wind your cassette past the leader, set to record, and press the pause button. Now type,

*SSAVE TESTSCREEN

START THE CASSETTE, AND PRESS RET

Release the pause on the cassette, and then press the return, or newline key. In about 20 seconds the screen should have been saved. If you now rewind, type CATALOG, and playback the cassette the screen filename should be listed, together with the letter "S" indicating a screen file. On SLOAD (or VLOAD), Hullforth searches for a file with the specified filename, before loading, so there is no need to find the beginning of the file manually.

You can now start to write complete Hullforth programs, as successions of screens each containing colon definitions. It is best to start with the simplest definitions, and gradually build upon these until the final program is achieved (itself a simple colon definition), this is a 'modular', bottom up approach. Each definition is tested separately, in turn, before proceeding to the next. Development and debugging times are thus kept short.

For examples of screens, see appendix 3, which contains full listings of all screen defined Hullforth words (including the editing, and cassette words). These are not claimed to be definitive solutions, and you are invited to re-define them to suit your own requirements!

5 The Hullforth Standard Vocabulary

This chapter contains a full description of each word in the Hullforth standard dictionary. Words are grouped into classes of operation, and not in the same order that they occur in the dictionary, as listed by HELP. Each word description includes details using the following notation:

Stack	(normal stack before -- normal stack after)		
Operand Key	n, n_1	..	16 bit value.
	d, d_1	..	32 bit value, ($d = n_{upper}\ n_{lower}$).
	addr	..	16 bit address.
	c	..	16 bit value whose lower 8 bits only are set, or used by the operation, and may represent an ascii character.
	f	..	16 bit value representing a boolean flag, zero = 'false', non-zero = 'true'.
	u	..	The prefix denoting an unsigned number.
	s	..	The prefix denoting a signed number.
Type Key	0	..	Word available at any time, compiled if inside a colon definition, executed otherwise.
	1	..	Word is always executed, but is only available inside a colon definition.
	2	..	Word is always executed, but is not available inside a colon definition.
	3	..	Word is always executed, but available anywhere. (For further details see the 'type' entry in appendix 2).

Those words which were defined using Hullforth screens are denoted by an asterisk immediately preceeding the description. These screens are listed in appendix 3 both as examples of Hullforth programming, and to enable the user to redefine words to suit his own requirements. 'The stack' in the word descriptions always refers to the 'normal' stack. Whenever the return stack is referred to this will be explicitly stated.

5.1 Stack Operations

PUSHN	(-- n)	Push the contents of the HL register pair onto the normal stack. Preserves HL. type 0.
PUSHR		Push the contents of the HL register pair onto the return stack. Preserves HL. type 0.
POPR		Pop the top value off the return stack, and into HL. type 0.
DROP	(n --)	Pop the top value off the normal stack, and into HL. type 0.
DUP	(n -- n n)	Duplicate the top of the stack. type 0.
SWAP	(n_1 n_2 -- n_2 n_1)	Reverse the order of the top two items on the stack. type 0.
OVER	(n_1 n_2 -- n_1 n_2 n_1)	Duplicate the second item, on the top of the stack. type 0.
ROT	(n_1 n_2 n_3 -- n_2 n_3 n_1)	Rotate the order of the top three items on the stack, so that the third item becomes the top item. type 0.
-DUP	(n -- n ?) .	DUP only if the value is non-zero, 'true'. type 0.
>R	(n --)	Pop the top item off the normal stack, and push it onto the return stack. type 0.
R>	(-- n)	Pop the top item off the return stack, and push it onto the normal stack. type 0.
I	(-- n)	Make a copy of the item on top of the return stack, and push it onto the normal stack. Do not alter the return stack. type 0.

J	(-- n)	Similar to I, but takes the third item on the return stack. type 0.
2DROP	(d --)	*Drop the top 32 bit value off the stack. type 0.
2DUP	(d -- d d)	*Duplicate the top 32 bit value. type 0.
2SWAP	(d_1 d_2 -- d_2 d_1)	*Swap the top two 32 bit values. type 0.
2OVER	(d_1 d_2 -- d_1 d_2 d_1)	*Duplicate the second 32 bit value, on the top of the stack. type 0.

Notes:

1. The operations PUSHN, PUSHR, POPR are not normally used by the Hullforth programmer, but are useful when defining new words using machine code. For example, a fast unsigned increment could be defined as follows,

```
HEX : INC  DROP ↑ 23 C, PUSHN ;
```

23 is the code for INC HL. DROP is used here to get the value into HL, but would normally be used to simply lose the top item off the stack.

2. I and J are special operations normally used inside DO .. LOOP constructions, see chapter 4.3 for examples.

5.2 Bases

BASE	(-- addr)	System variable containing the current base, used by input and output routines for number conversion.	type 0.
DEC		Set BASE to decimal (base 10).	type 0.
HEX		Set BASE to hexadecimal (base 16).	type 0.

Notes:
For examples of defining other bases see chapter 4.2.

5.3 Arithmetic and Logical

+	(sn_1 sn_2 -- sn_{sum})	Add.	type 0.
-	(sn_1 sn_2 -- sn_{diff})	Subtract, $sn_1 - sn_2$.	type 0.
*	(sn_1 sn_2 -- sn_{prod})	Multiply.	type 0.
/	(sn_1 sn_2 -- sn_{quot})	Divide, sn_1 / sn_2 .	type 0.
/MOD	(sn_1 sn_2 -- sn_{quot} n_{rem})	Divide with remainder, remainder is always positive.	type 0.
1+	(sn -- sn+1)	Add one.	type 0.
MINUS	(sn -- -sn)	Negate (2's complement).	type 0.
ABS	(sn -- sn_{abs})	Change sign if negative only.	type 0.
NOT	(n -- $\overline{n}$)	1's complement.	type 0.
AND	(n_1 n_2 -- n_{and})	Logical and.	type 0.
OR	(n_1 n_2 -- n_{or})	Logical or.	type 0
XOR	(n_1 n_2 -- n_{xor})	Logical exclusive-or.	type 0.
ADD	(un_1 un_2 -- un_{carry} un_{sum})	Add unsigned with carry.	type 0.
MULT	(un_1 un_2 -- ud_{prod})	Unsigned multiply with 32 bit result.	type 0.
DIV	(ud un -- un_{quot} un_{rem})	Unsigned 32 bit / 16 bit divide with 16 bit result and remainder.	type 0
U+	(un_1 un_2 -- un_{sum})	*Unsigned 16 bit add. (The same as ADD but without the carry).	type 0
UD+	(ud_1 ud_2 -- un_{carry} ud_{sum})	*Unsigned 32 bit add, with carry.	type 0

U/MOD	(ud un -- ud_{quot} un_{rem})	*Unsigned 32 bit / 16 bit divide with 32 bit result and 16 bit remainder.	type 0.
D+	(sd_1 sd_2 -- sd_{sum})	*Signed double add.	type 0.
D-	(sd_1 sd_2 -- sd_{diff})	*Signed double subtract.	type 0.
DMINUS	(sd -- -sd)	*Double negate.	type 0.
DABS	(sd -- sd_{abs})	*Double absolute.	type 0.

Notes:

1. All arithmetic is integer only.
2. Single precision (16 bit) number ranges are,

   signed, -32768 to 32767

   unsigned, 0 to 65535.
3. Double precision (32 bit) number ranges are,

   signed, -2147483648 to 2147483647

   unsigned, 0 to 4294967295.
4. Arithmetic overflow errors occur in signed arithmetic only, when the result of an operation would be outside the signed integer ranges indicated above. For example,

   HEX 7FFF 1 +

   would result in AOV ERROR. When doing calculation involving, say, addresses which may go over the 7FFFH - 8000H boundary use U+, to avoid AOV ERRORs.
5. ADD, MULT and DIV are the unsigned binary operations upon which all Hullforth signed arithmetic is based, they are included so that longer precision operations may be developed. For examples see the definitions of the Hullforth mixed and double precision operations in appendix 3.

5.4 Memory

@	(addr -- n)	Replace addr with the 16 bit word pointed to by addr. Addr points to the lower half byte, addr+1 the upper half byte. type 0.
C@	(addr -- c)	Replace addr with the 8 bit value pointed to by addr. The upper half byte of c is set to zero. type 0.
!	(n addr --)	Store n at addr. Lower half byte first. type 0.
C!	(c addr --)	Store the lower half byte of c at addr. type 0.
+!	(sn addr --)	Add sn into the contents of addr. type 0.

Notes:

The above memory operations are only byte or word addressing, but double (or greater) precision, may easily be defined. For example D! and D@ could be defined, using a 2 word ARRAY as variable storage,

```
2 1 ARRAY TEST
: TEST  1 TEST ;
: D!  DUP ROT SWAP ! 2 + !  ;
: D@  DUP @ SWAP 2 + @ SWAP  ;
```

Note that TEST is redefined to always return the address of the first word in the array, so that we can write,

```
1,000,000 TEST D!
```

to store the value 1,000,000 in the (double) variable TEST.

5.5 Comparison

=	(n_1 n_2 -- f)	Replaces the top two values, by a boolean, true if $n_1 = n_2$, false otherwise. type 0.
>	(sn_1 sn_2 -- f)	True if sn_1 greater than sn_2 . type 0.
<	(sn_1 sn_2 -- f)	True if sn_1 less than sn_2 . type 0.
0=	(n -- f)	True if n is zero, (reverses the truth value). type 0.
0<	(sn -- f)	True if sn is negative. type 0.
MAX	(sn_1 sn_2 -- sn_{max})	Replaces the top two values by the greater of the two values. type 0.
MIN	(sn_1 sn_2 -- sn_{min})	Replaces the top two values by the lesser of the two values. type 0.
D=	(d_1 d_2 -- f)	*Replaces the top two 32 bit values by a boolean, true if $d_1 = d_2$. type 0.
D0=	(d -- f)	*True if d is zero. type 0.
D<	(sd_1 sd_2 -- f)	*True if sd_1 is less than sd_2 . type 0.
D0<	(sd -- f)	*True if sd negative. type 0.

Notes:
The signed comparisons involve signed subtractions which could then result in AOV errors, if the result of the subtraction is outside the number range.

5.6 Control Structures

BEGIN UNTIL	(f --)	Sets up an indefinite loop, which terminates when f is true. type 1.

IF
ELSE
ENDIF
	(f --)	Construction, IF ..true words.. ENDIF or, IF ..true words.. ELSE ..false words.. ENDIF The true words are executed if f is true, the false words if f is false. Execution then continues normally after ENDIF. type 1.

DO LOOP	(sn_{end} sn_{start} --)	Set up a loop given index range. DO transfers sn_{start} and sn_{end} onto the return stack, so that sn_{start} is on top of the return stack. LOOP increments the value on top of the return stack and compares it with the second number on the return stack (sn_{end}), and loops back to the word following DO if the index is less than, or equal to sn_{end}. When the loop ends the index, and end values are cleared off the return stack. type 1.
+LOOP	(sn --)	Similar to LOOP, except that sn is added to the index value, rather than 1. type 1.
LEAVE		Sets the index value on top of the return stack equal to the end value, thus forcing premature termination of a DO .. LOOP. type 1.

Notes:

1. During Compilation the above words utilise the return stack for temporary storage of jump addresses. The return stack is cleared to empty at the start of a colon definition, and if it is not empty at the end (;), then a COL ERROR results indicating more BEGINs than UNTILs, IFs than ENDIFs or DOs than LOOPs. If the mismatch is the other way round then RSU ERROR will result.
2. All of the control structures may be nested to a maximum depth of 128.
3. LOOP performs signed addition so attempting a DO .. LOOP with an index crossing the 7FFFH - 8000H boundary will result in AOV ERROR.

5.7 Input-Output and Number Formatting

.	(sn --)	Print sn, left justified, in the current base. Print signed only if the base is decimal, unsigned otherwise. type 0.
.R	(sn n_{width} --)	Similar to . but prints right justified in a field width n, with leading spaces. FOR ERROR results if n_{width} is outside the range 1 to 16. type 0.
."		Print the string following ." and terminated by ". If occuring within a colon definition the string is compiled with a call to a string printer. (Not ."). type 3.
EMIT	(c --)	Print the character whose ascii is c. type 0.
KEY	(-- c)	Wait for keyboard input, and push the ascii for the character onto the stack. The char is echoed to the display. type 0.
CR		Print newline, (carriage return, line feed). type 0.
SP		Print a space. type 0.
?TERMINAL	(-- c)	Check to see if a key has been pressed, if it has,push the character onto the stack, otherwise push the value zero. type 0.
EXPECT	(addr n --)	Get n characters, or until carriage return, from the keyboard into memory starting at addr. A null byte terminates the string in memory. The backspace key may be used in EXPECT. type 0.

WORD	(c --)	Read the next word in the input stream, pointed to by IN, to HERE onwards. Use the character c as the delimiter. The last character of the word copied will have bit 7 set high. type 0.
NUMBER	(addr -- sd)	Convert the ascii byte string starting at addr into a double precision number in the current base, and push the number onto the stack. The last character in the string must have bit 7 set high. type 0.
PICT	(n -- addr)	*A 40 word, single dimensioned ARRAY, used to build formatted output. type 0.
PPNTR	(-- addr)	*Variable, points to current PICT element. type 0.
HOLD	(c --)	*Inserts, at the current position in the formatted output, the character c. HOLD (or a word using HOLD) must be used between <£ and £> . type 0.
<£		*Begin a formatted output. Initialises PPNTR. type 0.
£	(d -- d/base)	*Convert one digit of the double precision number d, into ascii in the current base, and insert it into the current position in the formatted output. £ will always produce a digit, if d is zero £ will produce the character zero. Must be used between <£ and £> . type 0.
£S	(d -- 0 0)	*Convert and HOLD all remaining significant digits, in the current base. d = zero is left on the stack. Must be used between <£ and £> . type 0.

£>	(d -- addr n)	*End the formatted conversion by leaving the character count, and address on the stack, for TYPE. type 0.
TYPE	(addr n --)	*Print n characters from an ARRAY starting at addr. type 0.
SIGN	(f d -- d)	*Insert the character "-", into the current position in the formatted output, only if f is true. Must be used within <£ and £> . type 0.
U.	(d --)	*Print the 32 bit unsigned number d, in the current base. Left justified. type 0.
D.	(sd --)	*Print the 32 bit signed number sd, in the current base. Left justified. type 0.
.2HEX	(c --)	*Print c as 2 hex digits, sets base to hex. type 0.
.4HEX	(n --)	*Print n as 4 hex digits, sets base to hex. type 0.

Notes:

1. On some microcomputers the character £, used in formatting words may be the character # (hash).
2. Formatted output is built backwards, the least significant digit is converted first. Although formatting words are type 0, new output words are most conveniently defined in colon definitions. As an example:

```
: .DATE <£ £ £ " / HOLD £ £ " / HOLD £ £ £> TYPE ;
```

would define a word .DATE, which prints a 32 bit number in the following format, 09/07/81, for the number 90781.

5.8 Defining Words

: xxx		Begin the colon definition of xxx. xxx may be any number of characters, and any characters except space. The word defined is of type 0. type 2.
;		End the colon definition, placing the RET machine instruction in the dictionary, updating dictionary pointers, and switching Hullforth back to interpret mode. type 1.
T: xxx	(n --)	Identical to ":", except that a new word of type n is defined. type 2.
CONSTANT xxx	**(n --)**	**Define a constant xxx with value n. xxx** will push the value n onto the stack when executed. type 2.
VARIABLE xxx	**(n --)**	**Define a variable xxx with initial value n. xxx will push the address of the variable** onto the stack when executed. type 2.
ARRAY xxx	**(n_i .. n_2 n_1 n_0 --)**	**Define a word array named xxx, with n_0 dimensions, and bounds n_1 .. n_i, where $i = n_0$. xxx when executed expects i indices on the stack, and returns the address** of the element indexed. type 2.

Notes:

1. If any error occurs during a word definition, then the definition is abandoned. The dictionary reverting back to its state before the abortive word definition.
2. As an example of an ARRAY definition, a 3 by 3 array TEST would be defined by,

   3 3 2 ARRAY TEST

   To select an element would require two indices, i.e. 3 3 TEST would return the address of the final word in the array.

5. 9 Editing and Cassette

SCRO		*Sets the current screen to screen 0. This is the default screen (SCR = 3800H for 16K configurations). (SCR= 6ØØØH). type 0.
SCR1		*Set the current screen to screen 1. (3C00H in 16K configurations). (SCR = 64ØØH). type 0.
NEW		*Clear the current screen to contain all spaces and enter a line number prompting sequence to enter new text into the screen. Exit either at the end of line 15, or when escape hit. type 0.
LI	(n_{from} n_{to} --) or, (n_{line} --) or, (--)	*List the current screen. If the stack contains 2 values take these as the start and end line numbers for the listing. If the stack contains 1 value list just that line. If the stack is empty list the whole screen. The n must be in the range 0 - 15. type 0.
RP	(n_{from} n_{to} --) or, (n_{line} --)	*Replace the lines, <u>or</u> line specified on the stack by new text. The line number prompts for the text. type 0.
DL	(n_{line} --) .	*Delete the line number specified, moving the rest of the screen up to fill the gap. type 0.
INS	(n_{line} --)	*Insert an extra line between two existing lines. I.e. <u>5 INS</u> will prompt for a new line, the old line 5 will become line 6, etc. type 0.
CL	(n_{line} --)	*Fill the line specified with spaces. type 0.
CY	(n_{from} n_{to} --)	*Copy line n_{from} into line n_{to}. Then fill line n_{from} with spaces. type 0.

SSAVE xxx — *Save the current screen, onto cassette, with filename xxx. type O

SLOAD xxx — *Search the cassette for a screen file named xxx, and if found, load the screen into the current screen. type O

VSAVE xxx — *Save the vocabulary, from (VSTART) to (HERE) onto cassette, with file name xxx. type O

VLOAD xxx — *Search for a vocabulary named xxx, and if foun load, and update dictionary pointers to include all words in the loaded vocabulary. type O

FORTH — *Update all system pointers so that a <u>cold</u> start into Hullforth will include all newly defined words, and then save the entire system onto cassette, in the microcomputers own forma This may then be loaded and run, from the system monitor. type O

CATALOG — *Read a cassette containing screens, and vocabularies, and list each file name. type O.

SCRN (n --) — * SET THE CURRENT SCREEN TO SCREEN n. n MUST BE IN THE RANGE Ø – 7. (SCR = 6ØØØH + n*4ØØH).

<u>Notes:</u>

1. Any of the Hullforth saves and loads (screens and vocabularies), may be aborted by hitting any key, before the operation is finished.
2. After a successful SLOAD or VLOAD the message "LOADED OK" will be printed. It is adviseable to stop the cassette as soon as possible after this message, to prevent any spurious input from the cassette.
3. The NASCOM 'motor drive LED' is not activated by screen or vocabulary saves and loads.

5.10 Miscellaneous and Utility

EXEC		Causes the contents of the current screen, (addressed by SCR), to be passed through Hullforth as if it was being input from the keyboard. The system reverts to normal keyboard input on finding a null (0) byte, which EXEC inserts as the last character of the screen, if it was not there already. type 0.
,	(n --)	Compile n into the dictionary. type 1.
C,	(c --)	Compile the byte c into the dictionary. type 1.
' xxx	(-- addr)	Pushes the address of the start of the executable code for the word xxx. NAM ERROR results if xxx is not found in the dictionary. type 0.
'H xxx	(-- addr)	Similar to "'", but returns the address of the start of the header for the word xxx. type 0.
" x	(-- c)	Pushes the ascii for the single character x onto the stack. type 3.
(		Begin comment, terminated by). A space must follow the (. type 3.
BRK		Exit Hullforth, into machine monitor. type 0.
ABORT		Terminate the current operation, clear all stacks, and print ERROR IN xxx. Where xxx is the word currently being executed. type 0.
EXIT		Terminate the current operation, but do not clear stacks, or print any message. type 0.

FORGET xxx		Forget all dictionary words back to, and including xxx. Use with care! type 2
ALLOT	(n --)	Leave a gap of n bytes in the dictionary. type 1.
↑		Execute the next word, even if within a colon definition. That is, make the word act like a type 3. type 1
SWAB	(n -- n_{swab})	Swap the upper and lower bytes of n. type 0.
RATE	(n --)	Set the printing speed, zero = fastest. type 0.
HCHECK		Check that HERE is not beyond END. Print END ERROR if it is. type 0.
BREAK	(-- f)	*Check the keyboard to see if a key has been pressed, if not return f = false. If a key had been pressed then test if it was the 'escape' key, (shift-newline on NASCOM), if so return with f = true. Otherwise suspend the program until another key has been pressed, giving a pause facility, and return f = false. type 0.
HEXLIST	(addr --)	*List the contents of memory, in hex and ascii from addr. Hit escape to break out or any other key to pause the listing. type 0.
NARGS	(-- n)	*Return the number of words currently on the normal stack. type 0.
INPUT	(c_{port} -- c_{input})	*Get a byte from the specified port number. type 0
OUTPUT	(c_{output} c_{port} --)	*Output the byte c_{output} to port c_{port}. type 0.

HELP		*List all words in the dictionary, from the most recently defined, backwards. Each word type is shown, and the address (in the current base) of the code. Hit escape to break out or any other key to pause. type 0.
TXT	(n -- addr)	*32 word ARRAY, used by TEXT, and cassette filenames. type 0.
TEXT xxx		*Get xxx into the array TXT. type 0.

5.11 System Variables

HERE	(-- addr)	HERE points to the next free space in the dictionary, the next word to be defined will start here. Hullforth also uses the space from HERE onwards as its temporary workspace, when interpreting. type 0.
IN	(-- addr)	IN points to the current parsing position of the input buffer, or screen during EXEC. type 0.
LINK	(-- addr)	LINK points to the header of the most recently defined word in the dictionary. type 0.
NSTK	(-- addr)	The normal stack pointer. type 0.
RSTK	(-- addr)	The return stack pointer. type 0.
SCR	(-- addr)	Start address of the current screen. type 0.
END	(-- addr)	The end of useable dictionary space. type 0.
VSTART	(- addr)	The start of the user vocabulary. A VSAVE saves the dictionary from VSTART to HERE. type 0.

Appendix 1. The System Memory Map

Hullforth version 1.1 Release 1000

Hullforth uses no memory below 1000HEX. Above 1000HEX the memory is configured as follows,

1000H - 1100H	The machine code stack.
1100H	Machine stack top. SP set to this on hard start.
1101H - 1206H	The 'return' stack.
1206H	Return stack top.
1207H - 130CH	The 'normal' stack.
130CH	Normal stack top.
1312H - 1391H	The line input buffer.
1392H - 13FFH	Reserved for system pointers and variables.
1400H - 3309H	Hullforth, cold start = 1400H soft start = 1448H start of dictionary = 15B5H
330AH - END	Free dictionary space, END initially set to 3800H, but may be reconfigured. (END = 6,ØØØH).
END - END+3FFH	Screen buffer 0.
END+400H - END+7FFH	Screen buffer 1.
END+(n*4ØØH) — END+(n*4ØØH)+3FFH	Screen buffer n=(Ø-7)

The END value and screen buffers are normally set for a system with memory ending at 3FFFH so that END = 3800H, executing SCRO sets SCR to 3800H, and executing SCR1 sets SCR to 3C00H.

If your actual high memory is above 3FFFH then adjust Hullforth as follows;
First write down your high memory address. That is, the address of the highest byte in your system giving continuous memory from 330AH upwards. Add 1. Then subtract 400H for each 'screen' buffer you would like to define. The final address you arrive at is your END value.

Worked example, for a system with continuous memory from 330AH to 7FFFH;
If two screen buffers are required, set the second to start at,

8000H - 400H = 7C00H.

Set the first to start at,

7C00H - 400H = 7800H.

And set END to 7800H.

To make these adjustments type into Hullforth,

```
HEX
: SCR1  7C00 SCR !  ;
: SCR0  7800 SCR !  ;
7800 END !
SCR0
DEC
```

And then FORTH save your new system onto cassette.

It is recommended that you do not attempt these modifications until after gaining some familiarity with Hullforth.

Appendix 2. The Hullforth Glossary of terms

Boolean

A boolean variable may only take one of the two 'logical' values, 'true' or 'false'. In Hullforth any 16 bit number may be interpreted as a boolean, in which case non-zero values are 'true', zero is 'false'.

Byte

An 8 bit quantity. Hullforth normally handles 16 bit (double byte) quantities, thus single byte values (such as ascii characters) are represented in Hullforth as 16 bit numbers with the upper byte set to zero.

Character

The standard character set recognised by Hullforth is ascii, and letters A-Z are normally upper case. New words may however be defined, which include lower case letters.

Special control characters are,

escape = 1BH The 'break' character
carriage return = ODH The 'return' character
and backspace = 08H.

No other control characters have any special significance.

Compile

'Compile' means 'translate into machine code'. Hullforth is set to 'compile' by the word colon (:), initiating a colon definition. Each word in the body of the colon definition, (between the name, and the semi-colon), then normally generates some machine code into the dictionary.

Numbers generate the 6 bytes,

```
LD   HL,number   ;when executed number will be
CALL PUSHN       ;pushed onto the normal stack
```

Control structures (IF, DO etc.), generate test and jump sequences, i.e. UNTIL generates the code,

```
CALL O=          ;test boolean
JP   Z,BEGIN     ;jump if 'false'
```

Most other words compile into 3 byte calls to the corresponding dictionary subroutines.

Immediates, (types 1 and 3), are executed immediately, and thus generate no code.

Dictionary

The dictionary forms 95% of Hullforth and is a linked list of dictionary entries. Each entry consists of a 'header' and a subroutine. The header contains the 'word' which identifies the particular dictionary entry, type information and a link address - pointing to the previous dictionary entry header. The subroutine is the executable part of the dictionary entry.
A new dictionary entry may only be created by one of the 'defining' words.
As an example, the dictionary entry created by the colon definition,

```
: OCT 8 BASE ! ;
```

is, in Z80 assembler,

```
ZOCT    DEFB    "O","C","T"+128
        DEFB    0               ;type 0
        DEFW    LINK            ;points to last header
OCT     LD      HL,7            ;7
        CALL    PUSHN
        CALL    BASE            ;BASE
        CALL    !               ;!
        RET                     ;;
```

With the header starting at ZOCT, and the subroutine starting at OCT. Thus the very next word to be defined would have a link address = ZOCT, and any occurences of the word OCT in further colon definitions would generate the code for CALL OCT.

Execute

A Hullforth word is 'executed' when the subroutine associated with it in the dictionary is called. This is caused by simply typing the word, outside a colon definition.

Header

Each dictionary entry has a header containing,

i) A character string terminated by a character with bit 7 set high. This is the word naming the dictionary entry.

ii) A single byte identifying the 'type' of the word, 0,1,2 or 3.

iii) A 16 bit link address, pointing to the start of the previous header in the dictionary. The very last word in the dictionary (in the reverse order in which it is searched), has a link address = zero, thus terminating the search.

Interpret

To interpret means to pass over (or, parse) a sequence of instructions obeying each one as it occurs. Thus Hullforth input containing no colon definitions, is interpreted.

Screen

A block buffer of 1024 bytes, terminated by a zero byte, which may be passed through Hullforth by typing EXEC. The screen may therefore contain any valid Hullforth input, especially colon definitions.

Stack

'Stack' is the term used for a special buffer for storing numbers, such that the last one to be stored is the first one to be retrieved. Numbers may only be retrieved in the reverse order that they were stored. A stack is thus a last-in first-out store. Most microprocessors implement stacks in their basic instruction sets, mainly to facilitate subroutine calls.

Hullforth maintains three stacks,

i) A machine code stack, whose stack pointer is the register pair SP. This stack is not accessible to the Hullforth programmer.

ii) The 'normal' stack, which is used for most arithmetic and general purposes, and is therefore highly accessible.

iii) The 'return' stack, which is reserved for compilation, and execution of DO loops. It is also accessible .

The routines which push and pop numbers on the normal and return stacks, also perform basic checking for empty, or full stacks.

Type

The majority of Hullforth words are type 0. That is, when they occur inside a colon definition they are compiled, otherwise they are executed.

Some Hullforth words are however 'immediates', that is they must be executed whenever they occur. 'Immediate' words are then never compiled (but they may occur within colon definitions). Hullforth makes the distinction between three types of immediate,

type 1. Only available inside a colon definition.

2. Not available inside a colon definition.

3. Available at any time.

All type 0 words may be forced to act like type 3 words by preceeding them by ↑ (up arrow).

Vocabulary

The vocabulary is that part of the dictionary which the Hullforth programmer has defined for his particular application. A number of vocabularies may be stored on cassette, and loaded as required. Vocabularies are not relocatable, and are, for VSAVE all of the words in the dictionary back to and including the word whose header is pointed to by the system variable VSTART.

Word

A word is any sequence of non-space characters delimited by spaces, occurring in the input stream. The word must be one of,

i) A dictionary entry.

ii) A valid number in the current base.

iii) The name given for a new dictionary entry, in a defining sequence.

If you type a valid number which is also a dictionary entry, then since the dictionary is searched first, the number will be mistaken for a dictionary entry. It is therefore recommended that new definitions are not named with numbers, to avoid confusing errors.

The term word is also used sometimes, to describe a 2 byte (16 bit) number, this will be clear from the context.

```
( UNSIGNED SINGLE AND MIXED PRECISION ARITHMETIC )
: U+     ADD SWAP DROP ;
: UD+    ( UNSIGNED DOUBLE ADD )
         ROT ADD  ( LOWER )  >R >R ADD  ( UPPER )
         R> ADD >R U+ R> R>  ( ADD CARRYS ) ;
: U/MOD ( UNNSIGNED MIXED DIVIDE )
         ROT OVER 0 ROT ROT DIV SWAP
                 >R ROT ROT DIV R> ROT ROT ;
( DOUBLE PRECISION STACK )
: 2DROP DROP DROP ;
: 2DUP  OVER OVER ;
: 2SWAP >R ROT ROT R> ROT ROT ;
: 2OVER 2SWAP 2DUP >R >R 2SWAP R> R> ;
```

```
( SIGNED DOUBLE PRECISION ARITHMETIC ) DEC
: DO<    DROP O< ;
: D+     2DUP DO< >R   ( SIGN OF ARG 1 )
  2SWAP  2DUP DO< >R   ( SIGN OF ARG 2 )
         UD+           ( ADD )
         ROT >R        ( CARRY )
         2DUP DO<      ( SIGN OF RESULT )
          2 * R> +
          2 * R> +
          2 * R> +     ( SIGNR*8 + CARRY*4 + SIGNA2*2 + SIGNA1 )
  DUP 8 = SWAP 7 = OR IF ." DAOV" ABORT ENDIF ( OVFL CHECK ) ;
: DMINUS SWAP NOT SWAP NOT  ( 1'S COMPLEMENT )
          1 ADD >R          ( ADD 1 TO LOWER )
          U+ R>             ( ADD CARRY INTO UPPER ) ;
: DABS  2DUP DO< IF DMINUS ENDIF ;  : D-  DMINUS D+ ;
: D<    D- DO< ;  : DO= OR O= ;  : D= D- DO= ;
```

```
( PICTURE FORMATTING )  DEC
40 1 ARRAY PICT  0 VARIABLE PPNTR
: HOLD PPNTR @ PICT.C!  -1 PPNTR +! ;
: <#    40 1 DO  32 I PICT C!  LOOP  ( CLEAR TO SPACES )
        40 PPNTR !                   ( INITIALISE PPNTR ) ;
: #     BASE @ U/MOD                 ( DIVIDE BY BASE )
        DUP 10 < IF 48 + ELSE 55 + ENDIF  ( ASCII )  HOLD ;
: #S    BEGIN #  OVER OVER OR 0=  UNTIL  ( QUOTIENT ZERO ) ;
: #>    DROP DROP  PPNTR @  1+ DUP PICT SWAP 41 SWAP - ;
: TYPE 1 DO  DUP C@ EMIT 2 + LOOP  DROP ;
: U.    <# #S #> TYPE ;
: SIGN ROT IF " - HOLD ENDIF ;
: D.    2DUP DO< ROT ROT  ( GET SIGN )  DABS  ( UNSIGN )
        <# #S SIGN #> TYPE ;
: .2HEX O SWAP HEX <# # # #> TYPE ;
: .4HEX O SWAP HEX <# # # # # #> TYPE ;
```

```
( UTILITIES 1 )  HEX
: BREAK  ( RETURNS TRUE IF ESC HIT, FALSE OTHERWISE )
         ?TERMINAL -DUP IF ( KEY PRESS )
                                 1B = IF ( ESC ) CR ." BREAK IN" CR 1
                                      ELSE KEY DROP 8 EMIT 0
                                      ENDIF
                              ELSE 0 ENDIF ;
: HEXLIST BASE @ SWAP ( SAVE CURRENT BASE ) HEX ( GO HEX )
          BEGIN  DUP DUP .4HEX SP SP                 ( ADDRESS )
             8 1 DO DUP C@ SP .2HEX 1 U+ LOOP        ( DATA )
             SP SP SP DROP
             8 1 DO DUP C@ 7F AND DUP
                    1F > IF ( PRINTABLE? ) EMIT
                         ELSE DROP SP ENDIF
                    1 U+ LOOP CR
    BREAK UNTIL DROP BASE ! ( RESTORE OLD BASE ) ;
```

```
( UTILITIES 2 )  HEX
: HELP LINK @  ( LIST ALL WORDS IN DICTIONARY )
       BEGIN BEGIN DUP C@               ( GET CHAR )
                   DUP 7F AND EMIT      ( PRINT IT )
                   80 AND               ( BIT 7 HIGH? )
                   SWAP 1+ SWAP         ( INCR POINTER )
             UNTIL                      ( END OF STRING )
             SP SP SP SP
             DUP C@ . SP ( PRINT TYPE )
             1+ DUP @    ( GET LINK )  SWAP 2 + . ( CODE ) CR
             BREAK IF DROP 0 ( FORCE LINK TO 0 ) ENDIF
       DUP 0= UNTIL ( LINK IS ZERO ) DROP ;
130C CONSTANT NSTACK  ( TOP OF NORMAL STACK )
: NARGS ( WORDS ON NORMAL STACK )
        NSTACK NSTK @ - 2 / 1 - ;
```

```
( EDITOR 1 )  HEX  3800 END !                        ( RESET END )
: SCR0 3800 SCR ! ;   : SCR1 3C00 SCR ! ;           ( SCREEN OPTIONS )
DEC  : LSTART  64 * SCR @ U+ ;                       ( START ADDRESS )
: LCHECK  DUP DUP 16 > SWAP 0< OR IF ." LINE" ABORT ENDIF ;
: LI  NARGS 0= IF 0 15                               ( LIST ALL )
               ELSE NARGS 1 = IF DUP                 ( LIST 1 LINE )
                              ENDIF
               ENDIF
      SWAP DO  I 2 .R  I DUP LCHECK LSTART DUP ROT
               15 = IF 63 U+ ELSE 64 U+ ENDIF     ( END OF LINE )
               BEGIN
                      -1 U+ DUP C@ 32 = 0= >R     ( STEP BACK )
                      OVER OVER = R> OR           ( TO NON SPS )
               UNTIL
               SWAP DO I C@ EMIT LOOP CR          ( PRINT LINE )
          LOOP ;
```

```
( EDITOR 2 ) DEC
: RP  NARGS 1 = IF DUP ENDIF                         ( REPLACE )
      SWAP DO
             I DUP 2 .R
             LSTART DUP 64 EXPECT PUSHN DUP 32 SWAP C!
             1+ SWAP 63 U+ SWAP DO 32 I C! LOOP ( SPACE FILL )
           LOOP ;
: CL  LCHECK LSTART DUP 63 U+ SWAP DO 32 I C! LOOP ; ( CLEAR )
: NEW 15 0 DO  I CL  LOOP                            ( NEW SCREEN )
      15 0 DO  I RP  LOOP ;
: CY  LCHECK SWAP LCHECK SWAP                        ( CHECK LINES )
      LSTART OVER LSTART
      63 0 DO
             OVER OVER C@ SWAP C!                    ( COPY CHARS )
             1 U+ SWAP 1 U+ SWAP                     ( INC ADDRS )
           LOOP DROP DROP CL ;                       ( ERASE OLD )
```

```
( EDITOR 3 ) DEC
: DL  LCHECK DUP 15 =                   ( DELETE )
      IF CL                             ( LINE 15 )
      ELSE 14 SWAP DO                   ( ELSE SHUNT ALL UP )
                     I DUP 1+ SWAP CY
                   LOOP
      ENDIF ;
: INS LCHECK DUP 15 =                   ( INSERT )
      IF ELSE
          14 BEGIN
               DUP DUP 1+ CY            ( MOVE LINE N TO N+1 )
               1 - OVER OVER >          ( FROM THE REAR )
             UNTIL DROP
      ENDIF RP ;                        ( AND ENTER NEW TEXT )
```

```
( UTILITY 3 )
DEC  32 1 ARRAY TXT   .                 ( TEXT BUFFER )
HEX  : TEXT  20 WORD                    ( GET WORD INTO TXT )
             20 1 DO  20 I TXT !  LOOP  ( FILL TXT WITH SPACES )
             HERE @  1 >R               ( SET UP PNTRS )
             BEGIN
                     DUP C@ DUP 7F AND I TXT C!  ( COPY CHAR )
                     80 AND  SWAP 1+ SWAP        ( BIT 7 HIGH? )
                     R> 1+ DUP >R 21 = OR        ( OR >32 CHRS? )
             UNTIL DROP R> DROP ;                ( EXIT IF SO )
: INPUT DROP                            ( PORT TO L )
        ↑ 4D C,   ↑ ED C,   ↑ 68 C,     ( LD C,L IN L,C )
        ↑ 26 C,   ↑ 00 C,   PUSHN ;     ( LD H,0 AND PUSH )
: OUTPUT DROP                           ( PORT TO L )
         ↑ 4D C,   DROP                 ( LD C,L DATA TO L )
         ↑ ED C,   ↑ 69 C, ;            ( OUT C,L )         DEC
```

```
( CASSETTE 1 ) HEX                     ( NASCOM PORTS, AND STATUS BITS )
: FORTH                                ( SAVE WHOLE SYSTEM )
  LINK @ 1401 !   HERE @ 1407 !                  ( UPDATE POINTERS )
  HERE @ 0C0E !   1400   0C0C !   1400 DROP      ( SET ARG1,ARG2,HL )
       ↑ DF C,   ↑ 57 C,   ↑ 0 C,   ;            ( WRITE CASS )
: SRLOUT 1 OUTPUT                     ( SERIAL OUTPUT )
         BEGIN  2 INPUT 40 AND                            ( TBR? )
                ?TERMINAL IF EXIT ENDIF  UNTIL ;          ( OR BRK? )
: SRLIN  BEGIN  2 INPUT 80 AND                            ( RDY? )
                ?TERMINAL IF EXIT ENDIF  UNTIL            ( OR BRK? )
         1 INPUT ;                    ( GET SERIAL INPUT )
: WRDAT DO   I C@ DUP  SRLOUT   + FF AND  LOOP ;  ( WRITE DATA )
: RDDAT DO   SRLIN  DUP  I C!   + FF AND  LOOP ;  ( READ DATA )
: CHK+  DUP SWAB  FF AND  SWAP  FF AND  + + FF AND ;    DEC
: WRN   DUP DUP SRLOUT SWAB SRLOUT CHK+ ;         ( WRITE WORD )
: RDN   SRLIN SRLIN SWAB OR DUP >R CHK+ R> ;      ( READ WORD )
```

```
( CASSETTE 2 ) HEX
: WRHEAD ( WRITE HEADER: 64 NULLS, FF, 16 CHAR FILENAME, TYPE )
         TEXT  ." START TAPE, THEN PRESS RET" KEY DROP
         40 1 DO  0 SRLOUT        LOOP FF SRLOUT
         10 1 DO  I TXT C@ SRLOUT LOOP SRLOUT  100 1 DO LOOP ;
: WRTAIL ( WRITE TAIL: 16 NULLS )
         10 1 DO  0 SRLOUT  LOOP  ." STOP TAPE" CR ;
: SSAVE ( SCREEN SAVE )
        " S WRHEAD                               ( TYPE "S" )
        O SCR @ DUP 3FF + SWAP WRDAT             ( CURRENT SCREEN )
        SRLOUT ( CHKSUM ) WRTAIL ;
: VSAVE ( VOCAB ) HERE @ VSTART @ = IF ." EMPTY" ABORT ENDIF
        " V WRHEAD                                     ( TYPE V )
    0 LINK @ WRN   HERE @ WRN   VSTART @ WRN  SRLOUT ( ADDRESS )
    0              HERE @       VSTART @ WRDAT SRLOUT ( DATA )
      WRTAIL ; DEC
```

```
( CASSETTE 3 ) HEX
: RDHEAD ( SEARCH FOR 00,FF HEADER )
         1 BEGIN  SRLIN DUP ROT XOR FF =  UNTIL DROP
         20 11 DO  SRLIN I TXT C!  LOOP  SRLIN ;
: PRFN   ( PRINT FILE NAME, FILE TYPE )
         11 TXT 10 TYPE EMIT CR ;
: CATALOG ." PRESS RET, THEN START TAPE" KEY DROP CR
          ." FILENAME      TYPE" CR
          BEGIN  RDHEAD PRFN  0 UNTIL ( USER BREAKS IN ) ;
: GETFILE 0 BEGIN  DROP RDHEAD          ( SEARCH FOR A FILE )
                   0 10 1 DO
                             I TXT C@ XOR      ( 0-10 TXT = )
                             I 10 + TXT C@ XOR ( 11-20 TXT ? )
                           LOOP 0=             ( MATCH IF SO )
            UNTIL  - IF ." FILE" ABORT ENDIF ;
```

```
( CASSETTE 4 ) HEX          ( SCREEN AND VOCAB LOADS )
: LOSTRT  ." PRESS RET, THEN START TAPE" KEY DROP CR ;
: LOEND     SRLIN XOR IF ." CHECKSUM" ABORT ( CHECKSUM OK? )
                      ELSE  20 PRFN ." LOADED OK" CR
                      ENDIF ;
: VLOAD TEXT  LOSTRT " V GETFILE                  ( GET FILE )
        O RDN SWAP RDN SWAP RDN SWAP              ( GET ADDRS )
        SRLIN XOR IF ." CHECKSUM" ABORT ENDIF ( CHECK OK? )
        OVER O ROT ROT SWAP RDDAT LOEND
        HERE ! LINK ! ;                           ( RESTORE )
: SLOAD TEXT  LOSTRT " S GETFILE
        O SCR @ DUP 3FF U+ SWAP RDDAT
        LOEND ;
```

Appendix 4. Error Messages

Whenever an error occurs in Hullforth the current operation is aborted, and the message printed:

xxx ERROR IN yyy

where xxx is a mnemonic indicating which error has occured, and yyy is the word currently being executed. Also all stacks are reset, and the system is set to normal keyboard input mode. The following is a list of system defined error mnemonics, with a description of the possible cause of the error,

NSU	Normal Stack Underflow	An attempt has been made to 'pop' a value off an empty normal stack.
NSO	Normal Stack Overflow	An attempt has been made to 'push' a value onto a full normal stack.
RSU	Return Stack Underflow	Similar to NSO but for the return stack.
RSO	Return Stack Overflow	Similar to RSO but for the return stack.
AOV	Arithmetic Overflow	A signed arithmetic operation has resulted in a number outside the number range.
BAS	Base Error	A number is invalid in the current base, or, word not found in dictionary.
TYP	Type Error	A word is being used in the wrong context for its 'type'.
IP	Input Error	The text required by WORD, or ." etc. is missing or incorrectly terminated.
FOR	Format Error	Number formatting error.
DZ	Division by Zero	An attempt has been made to divide by zero.
COL	Colon Error	Control structure mismatch inside a colon definition.
NAM	No Name Error	The name required by a defining word, or dictionary searching word is missing.
END	End Error	Array too big for the dictionary space left.
AOB	Array Out of Bounds	An attempt has been made to index an array element outside the defined bounds.

DAOV	Double Arithmetic Overflow	A signed double precision arithmetic operation has resulted in a number outside the double number range.
LINE	Line Number Error	The line number specified for the editing operation is outside the range 0 - 15.
FILE	Cassette File Error	An attempt has been made to load a cassette screen when a vocabulary was required, or vice-versa.
CHECKSUM	Checksum Error	A cassette load has resulted in a checksum error.
EMPTY	Empty Vocabulary Error	A VSAVE has been attempted of an empty vocabulary.

Note:

The Hullforth programmer may define his own error messages, to produce the same message format as the above, using the word ABORT. For example, including

IF ." message" ABORT ENDIF

in a colon definition, would cause 'message ERROR IN yyy' to be printed, and execution of yyy to be terminated, whenever the condition test at IF was true.

Appendix 5. The Hullforth Handy Reference

Stack notation: (normal stack before -- normal stack after)

Operand Key:

n,n1	..	16 bit value
d,d1	..	32 bit value, (d=nupper nlower)
addr	..	16 bit address
c	..	16 bit value whose lower 8 bits only are set, or used by the operation, and may represent an ascii character.
f	..	16 bit value representing a boolean flag, zero = 'false' non-zero = 'true'
u	..	the prefix denoting an unsigned number
s	..	the prefix denoting a signed number

Stack Operations

PUSHN	(-- n)	push the HL* pair onto the normal stack
PUSHR		push the HL pair onto the return stack
POPR		pop return stack into HL
DROP	(n --)	pop normal stack into HL
DUP	(n -- n n)	duplicate top of stack
SWAP	(n1 n2 -- n2 n1)	reverse top two stack items
OVER	(n1 n2 -- n1 n2 n1)	duplicate second item on top
ROT	(n1 n2 n3 -- n2 n3 n1)	rotate third item to top
-DUP	(n -- n ?)	duplicate only if non-zero
>R	(n --)	move top item to return stack
R>	(-- n)	move top of return stack to normal stack
I	(-- n)	copy top of return stack onto normal stack
J	(-- n)	copy third item on return stack on normal stack
2DROP	(d --)	drop the top 32 bit value off the stack
2DUP	(d -- d d)	duplicate the top 32 bit value
2SWAP	(d1 d2 -- d2 d1)	swap the top two 32 bit values
2OVER	(d1 d2 -- d1 d2 d1)	duplicate second 32 bit value on top of stack

* HL refers to the Z80 register pair

Bases

BASE	(-- addr)	system variable containing current base
DEC		sets BASE to decimal
HEX		sets BASE to hexadecimal

Arithmetic and Logical

+	(sn1 sn2 -- snsum)	add
-	(sn1 sn2 -- sndiff)	subtract, sn1-sn2
*	(sn1 sn2 -- snprod)	multiply
/	(sn1 sn2 -- snquot)	divide, sn1/sn2
/MOD	(sn1 sn2 -- snquot snrem)	divide with remainder
1+	(sn -- sn+1)	add one
MINUS	(sn -- -sn)	change sign (2's complement)
ABS	(sn -- snabs)	change sign if negative only
NOT	(n -- n)	1's complement
AND	(n1 n2 -- nand)	logical and
OR	(n1 n2 -- nor)	logical or
XOR	(n1 n2 -- nxor)	logical exclusive or
ADD	(un1 un2 -- uncarry unsum)	add unsigned with carry
MULT	(un1 un2 -- udprod)	unsigned multiply with 32 bit result
DIV	(ud un -- unquot unrem)	unsigned 32/16 bit divide with 16 bit remainder

U+	(un1 un2 -- unsum)	unsigned 16 bit add
UD+	(ud1 ud2 -- uncarry udsum)	unsigned 32 bit add, with carry
U/MOD	(ud un -- udquot unrem)	unsigned 32/16 bit divide with 32 bit result
D+	(sd1 sd2 -- sdsum)	signed double add
D-	(sd1 sd2 -- sddiff)	signed double subtract
DMINUS	(sd -- -sd)	double negate
DABS	(sd -- sdabs)	double absolute

Memory

@	(addr -- n)	replace addr by 16 bit value at addr
C@	(addr -- c)	replace addr by 8 bit value at addr (upper set zero)
!	(n addr --)	store n at addr
C!	(c addr --)	store c at addr
+!	(sn addr --)	add sn into the contents of addr

Comparison

=	(n1 n2 -- f)	true if top two numbers equal
>	(sn1 sn2 -- f)	true if sn1 greater than sn2
<	(sn1 sn2 -- f)	true if sn1 less than sn2
0=	(n -- f)	true if n is zero
0<	(sn -- f)	true if sn negative
MAX	(sn1 sn2 -- snmax)	maximum
MIN	(sn1 sn2 -- snmin)	minimum
D=	(d1 d2 -- f)	true if d1 = d2
D0=	(d -- f)	true if d is zero
D<	(sd1 sd2 -- f)	true if sd1 less than sd2
D0<	(sd -- f)	true if sd negative

Control Structures

BEGIN		loop back to BEGIN until true at UNTIL
UNTIL	(f --)	
IF	(f --)	construction IF ..true words.. ENDIF
ELSE		or IF ..true words.. ELSE ..false words.. ENDIF
ENDIF		
DO	(snend snstart --)	set up loop given index range
LOOP		add 1 to index, exit when index greater than end
+LOOP	(sn --)	add top stack number to index, and test for end
LEAVE		force DO..LOOP termination

Input-Output and Number Formatting

.	(sn --)	print n, left justified in current base
.R	(sn nwidth --)	print n, right justified in field width nwidth
."		print string terminated by "
EMIT	(c --)	print c
KEY	(-- c)	wait for keyboard input, put char on stack
CR		print carriage return, line feed
SP		print space
?TERMINAL		read keyboard without waiting, c=zero if no char
EXPECT	(addr n --)	read n chars, or until CR, from keyboard into addr
WORD	(c --)	read word delimited by c, from input buffer to HERE
NUMBER	(addr -- sd)	convert byte string at addr to double prec. number
PICT	(n -- addr)	ARRAY, used to build formatted output
PPNTR	(-- addr)	VARIABLE, pointer into PICT
HOLD	(c --)	insert c into formatted output
<£		begin a formatted output
£	(d -- d/base)	convert next digit and HOLD it

£S	(d -- 0 0)	convert and HOLD all remaining significant digits
£>	(d -- addr n)	end format, and prepare for TYPE
TYPE	(addr n --)	print n chars from an ARRAY, from addr
SIGN	(f d -- d)	HOLD "-" char only if f is true
U.	(d --)	print 32 bit number, unsigned
D.	(sd --)	print 32 bit number, signed
.2HEX	(c --)	print c as 2 hex digits
.4HEX	(n --)	print n as 4 hex digits

<u>Defining Words</u>

: xxx		begin colon definition of xxx
;		end colon definition
T: xxx	(n --)	begin colon definition of word type n
CONSTANT xxx	(n --)	define a constant xxx with value n
VARIABLE xxx	(n --)	define a variable xxx with initial value n
ARRAY xxx	(ni .. n2 n1 n0 --)	define a word array named xxx, with n0 dimensions where n0=i, with bounds n1 .. ni

<u>Editing and Cassette</u>

SCR0		set current screen 0
SCR1		set current screen 1
NEW		input a new screen
LI	(n1 n2 --)	list the current screen from line n1 to n2
RP	(n1 n2 --)	replace lines n1 to n2 in current screen
DL	(n --)	delete line n - shuffle screen up
INS	(n --)	insert a new line n - move screen down
CL	(n --)	fill line n with spaces
CY	(n1 n2 --)	copy line n1 into line n2, then CL line n1
SSAVE xxx		save the current screen, filename xxx
SLOAD xxx		load the screen named xxx
VSAVE xxx		save (compiled) user vocabulary, filename xxx
VLOAD xxx		load the vocabulary named xxx
FORTH		save whole system including new definitions
CATALOG		catalog cassette containing screens and vocabularies

<u>Miscellaneous and Utility</u>

EXEC		pass the current screen through Hullforth
,	(n --)	compile n into the dictionary
C,	(c --)	compile c into the dictionary
' xxx	(-- addr)	return the start of executable code for xxx
'H xxx	(-- addr)	return the header address for xxx
" x	(-- c)	push the ascii for the char x
(		begin comment, terminated by)
BRK		exit Hullforth into machine monitor
ABORT		error termination of operation, clears stacks
EXIT		terminate current operation
FORGET xxx		forget all dictionary words back to and including xxx
ALLOT	(n --)	leave a gap of n bytes in the dictionary
↑		execute the next word immediately
SWAB	(n -- nswab)	swap the upper and lower bytes of n
RATE	(n --)	set the printing speed, zero=fastest
HCHECK		check that HERE is not beyond END
BREAK	(-- f)	returns true if escape has been hit
HEXLIST	(n --)	lists memory in hex, and ascii from n
NARGS	(-- n)	returns the number of words on normal stack

INPUT	(cport -- cinput)	get byte from specified port
OUTPUT	(coutput cport --)	output byte to port
HELP		list all words in the dictionary
TXT	(n -- addr)	32 word ARRAY, used by TEXT
TEXT xxx		get the word xxx into array TXT

System Variables

HERE	(-- addr)	current free space in dictionary
IN	(-- addr)	pointer to current input stream
LINK	(-- addr)	address of last header in dictionary
NSTK	(-- addr)	normal stack pointer
RSTK	(-- addr)	return stack pointer
SCR	(-- addr)	start address of current screen
END	(-- addr)	end of dictionary space
VSTART	(-- addr)	start of user vocabulary