BINAR™ Forth Reference

Preliminary - Version 0.0

March 1, 1990

CONTENTS

1.0 Introduction

This document is preliminary Version 0.0. This version is the working document used by Harris engineers, and is not necessarily complete and accurate in all respects. If you have any questions or doubts about the technical content of this document, please contact Harris for clarification. A more comprehensive and coherent document to supersede this version is currently in preparation.

Forth began as a solution to the problems facing Charles Moore in the late 1960's, and has grown from a language describing a virtual machine into real machines which can execute Forth very efficiently. The BINAR chip is a stack-based machine, and so is very efficient at executing Forth. Therefore, this Forth system has been created to provide an interactive software development environment for the BINAR evaluation board.

This document is not designed to teach Forth, but to give a user already acquainted with Forth directions in which to explore this particular implementation of Forth. Anyone not already familiar with the concepts on which Forth is based should consult Leo Brodie's book, Starting Forth. It should be noted that there are many differences between the Forth described by Brodie and this particular implementation -- please consult the Glossary for the exact definition of any Forth word.

2.0 Philosophy

This implementation of Forth is tailored to the BINAR hardware, but is based on the FORTH 83-STANDARD. Unfortunately, this standard was defined before any hardware implementations of stack machines were built. Consequently, many of its restrictions are arbitrary and far too limiting for a 32-bit stack machine to be forced to exist with, such as the required use of a 16-bit address space, 16-bit stack elements, and floored division. We have therefore modified our Forth where necessary to better fit our architecture while remaining reasonably compatible with the standard. These modifications are based on existing practice of other 32-bit Forth implementations, such as those for the 68020 processor.

The basic paradigm on which Forth is based is that of a stack: Forth uses a stack for virtually all computation and parameter passing. Its computation model closely resembles the Reverse Polish Notation (RPN) calculators built by the Hewlett-Packard Company. There are no explicit registers to use for intermediate storage.

The RTX family of processors by Harris Semiconductor are all stack-based machines. Each has two stacks; one for return addresses, the other for computation and parameter passing. The similarities between the virtual machine of Forth and the physical implementation of a stack machine result in a computer that can execute Forth (a high-level language) as its opcode set.

This document describes an implementation of Forth for the BINAR, a stack-based machine by Harris Semiconductor.

The BINAR is a stack-based, microcode processor. It has an opcode set consisting of 2048 words of microcode rom and 128 words of user-writable microcode ram. The 200+ opcodes in rom comprise a super set of the opcodes required by the Forth virtual machine -- many are included to support other languages, such as C. There are 16 opcodes mapped into ram, which the user may define and use as needed.

Note also that all 32-bit memory operations will occur on word-aligned memory addresses -- the least significant 2 bits of the address will be ignored and no exception will be generated. Likewise, half-word operations mask the least significant bit of the address.

3.0 Quick Start

3.1 Starting the System

To begin executing Forth on the BINAR, follow this script (allowing for customized directory names for any particular installation):

To compile the BINAR kernel:

ENV -c KERNEL.4

To extend the kernel into a full Forth system:

ENV -i KERNEL.IMG

Note that the initial execution of KERNEL.IMG will automatically load the files required for the system extension, save the extended image as FORTH.IMG, and leave the system in the "dumb terminal" mode. To re-enter this mode after having created the FORTH.IMG file once:

ENV -i FORTH.IMG -t

At this point, the PC will appear to be a very nice, dumb terminal for the BINAR hardware running Forth.

ENV.EXE has exactly 4 command line switches. They are:

-i Load an image file and begin terminal mode.

-t Begin terminal mode, assuming BINAR is already loaded.

-c Compile the specified file for the BINAR. Truly useful only for the KERNEL.4 file.

-" Parse the string up to but not including the following double-quote and send the string to the BINAR just as if typed on the keyboard.

All of the command line switches and filenames must be followed by a blank space (just like Forth interpretation).

The full filename (and path if necessary) must be specified. Multiple commands may be entered on one command line.

Also note that via the command line, input and output may be redirected as per DOS conventions, but the running application on the BINAR must end with the Forth command **BYE** in order to return to DOS.

```
ENV -i FORTH.IMG -" WORDS BYE " >GLOSSARY.LST
```

3.2 BINAR Files

ENV.EXE The host environment utility. Used to communicate with the BINAR hardware.

FORTH.BAT A batch file that will load the BINAR with FORTH.IMG and execute the terminal program.

KERNEL.BAT A batch file which will recompile all of the BINAR system extensions, assuming that KERNEL.IMG and all of the source code to the extensions exist.

COMPILE.BAT A batch file to recompile the KERNEL.IMG file from the KERNEL.4 file.

TERM.BAT A batch file to re-execute the terminal program without disturbing the current image on the BINAR.

KERNEL.4 Source code for the BINAR Forth kernel.

BUILD.4 Load file for extensions.

CURSOR.4 Cursor control words.
CUSTOM.4 User definable extensions.
DASM.4 Decompiler / disassembler.
DOS.4 Dos interface for BINAR / host.
EXCISE.4 Routines to relink words.
EXPECT.4 Command line editing.
EXTEND.4 Basic extensions to the kernel.
FLOAT.4 Floating point software extensions.
FORGET.4 Discard definitions.
OPCODREF.4 Opcode reference tables for compiler / decompiler.
OPTIMIZ.4 The optimizing compiler.

TIMER.4 Time of day, date, and timer routines.
UASM.4 Micro-code assembler.
VIEW.4 Source code location and editing.
VOCAB.4 Vocabularies.
WORDS.4 Dictionary display.
XOPS.4 Cross reference for the opcodes.

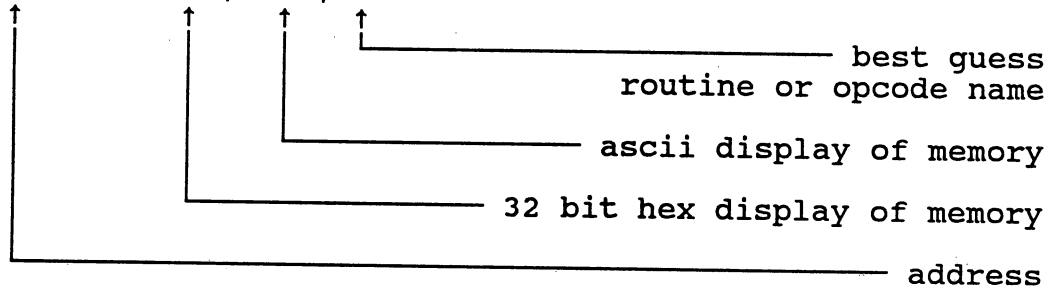
3.3 Decompiler

Any Forth word may be decompiled interactively. This is very useful for debugging programs, or for determining exactly what the optimizing compiler has done. There is one basic decompiler word called **UN**, which will decompile 20 instructions from an address given on the stack. Its typical use is:

```
' DECIMAL UN <cr>
```

which will display

```
1D20 | 0C140602 | .... | A NOP
1D24 | 00001990 | .... | BASE
1D28 | 06000001 | .... | ! exit
1D2C | 0000014E | N... | +_B@ <interrupt>
1D30 | 00001D14 | .... |    1D14 call
1D34 | 58454883 | .HEX | -EADE0 OR
```



UN will decompile starting at any address, even in the middle of a text string or data table, and may produce meaningless results. Typically, **UN** is used to decompile a particular word in the Forth dictionary, as in the example above. Since that syntax is bit cumbersome, the following higher-level word **SEE** is defined. Its typical use is:

```
SEE DECIMAL
```

which has the exact same effect as the previous example.

All decompilations may be stopped by hitting any key on the keyboard (see the glossary entry for **KEY?**)

3.4 Locator

During compilation of source code, the Forth compiler keeps a linked list of the files compiled and the line number where each word was defined. The source code locator **VIEW** takes advantage of this data to allow the interactive automated lookup of any word that was compiled from a loaded file. Its typical use is:

```
VIEW WORDS
```

which will invoke the system-defined editor on the file in which **WORDS** was defined at the proper line. See also FILES, DICTIONARY STRUCTURE.

3.5 Files

Application source code is maintained in normal Ascii files. These files may be edited by any text editor that does not embed special control characters in the edited text. The Forth system already knows about a particular editor, named **EDIT**, and can call the editor interactively (see VIEW). The editor may be invoked by typing:

```
EDIT MYFILE.XYZ
```

Files that contain source code may be loaded by the command **LOAD**. Its typical use is:

```
LOAD APPLIC.4
LOAD D:\SOURCES\MYFILE.XYZ
```

Note that the full filename and path (if necessary) must be specified -- the name given at load time is the name that will be "remembered" by the system for later **VIEW**ing.

3.6 Input Line Editor

The interactive Forth interpreter depends heavily on user typing. To make life easier for the user, Forth includes a simple line editor. It allows the retrieval and editing of the last 8 command lines. Keystroke controls for the line editor are:

return	terminate the current line

escape	erase the current line
up-arrow	retrieve the last line for editing
down-arrow	retrieve the oldest line for editing
right-arrow	cursor right
left-arrow	cursor left
home	cursor to the left margin
end	cursor to right margin
insert	insert one space under the cursor
delete	delete character under the cursor
backspace	delete character to left of cursor

Note that the line editor is always in overstrike mode -- any inserting is done via the insert-space command and over typing the blanks.

3.7 Words

The basic unit of Forth programs is the "word". A Forth word is roughly equivalent to a subroutine in other languages, but it is a much broader concept. Every entity in Forth is just a word -- compiler directives, language primitives, application definitions, constants.

Central to Forths use of words is the memory-based dictionary. This data structure is a linked list of the names (in Ascii) of all the words that Forth knows. All compilation addresses come from the dictionary.

To view the dictionary, a utility called **WORDS** is provided. It has three modes of operation:

WORDS <cr> display all the names in the CONTEXT vocabulary

WORDS XYZ <cr> display all names in all vocabularies that contain the string "XYZ" (case ignored)

WORDS *.* <cr> display all names in all vocabularies

3.8 Turnkey Applications

BINAR Forth has the capability to build a turnkey program, that once loaded can execute completely independently of its host.

On reset and program load, BINAR Forth executes an initialization word, **COLD**. After performing the system specific initialization, it searches for the name **READY** in the dictionary and executes it.

To use **READY** to build a turnkey application, it (or one of its component words) must execute the system call **BYE**. This forces the host environment to terminate and return to DOS.

To create a very simple turnkey application, try:

(in Forth)

```
: READY   WORDS BYE ;     ( define the application.)
SAVE-SYSTEM TEST.IMG      ( save an executable image.)
BYE                        ( return to dos)
```

(in dos)

```
LOAD TEST                  ( load and execute the
turnkey)
```

READY may also be used to perform a specialized initialization sequence by simply executing a return to **COLD** which will continue the normal boot process.

3.9 Dos Shell Interface

Even though the BINAR is very distinct from a processor that can run MS-DOS, it is implemented as a co-processor card for a DOS machine. It becomes much more user-friendly if, while running a program on the BINAR, DOS services are still available. This is accomplished through a "command processor" built into the host environment.

The simplest DOS interface is **** . When BINAR Forth encounters the DOS-ESCAPE (double-backslash), it parses the remainder of the input line, formats the entire string as a DOS command, and sends the command to the host. The host then executes the specified DOS command and returns to the BINAR. For example:

```
( print a director listing)
\\ DIR <cr>

( execute an arbitrary utility)
\\ dbase2 \files\myfile <cr>
```

A full DOS shell may be spawned by typing

SHELL <cr>

and this may return to Forth by typing "exit" at any DOS prompt.

There is also a shorthand form of commonly used DOS commands available through a Forth defining word **DOS:** . Examples of some commands commonly used are:

```
DOS: DIR
DOS: COPY
DOS: RENAME
```

These commands, like the \\ commands, parse the remainder of the input line and pass the entire formatted string for DOS execution. Therefore, one may not mix a "dos" command and a Forth command on the same line.

3.10 Optimizing Compiler

The BINAR processor is capable of executing combinations of opcodes, subroutine calls, and subroutine exits in parallel. This is fully documented in the BINAR TECHNICAL REFERENCE.

The Forth compiler implemented for the BINAR performs as much optimization as possible during compilation. The basic algorithm that it employs is "greedy compilation," i.e. it packs best fit into 2ops format instead of trying to analyze and compile the absolute most optimal sequence based on opcode timing.

A diagram of the compiler's finite state machine may be found in the file OPTIMIZ.4.

3.11 Memory Dump

Memory is organized into bytes and may be displayed on the console in either byte- or word- organized format. The commands for this are:

```
start #bytes DUMP        ( for byte dump)
start #bytes WDUMP       ( for word dump)
```

For example:

```
HEX  100 80 WDUMP
```

will display 128 bytes of data, beginning at address 100H, formatted into 32-bit units.

3.12 Vocabularies

BINAR Forth supports only a very primitive vocabulary structure. All vocabularies are sealed entities, defined by (for instance):

VOCABULARY SYSTEM

Subsequent execution of the word **SYSTEM** will cause **SYSTEM** to become the **CONTEXT** vocabulary. If **DEFINITIONS** is executed, **SYSTEM** will also become the **CURRENT** vocabulary, and all new definitions will be linked into it.

Vocabularies are searched according to these rules:

1- search the context vocabulary
2- search the Forth vocabulary

Vocabularies are meta-structures: they contain links to each other as well as the normal dictionary links. This allows words like **FORGET** to trim all of the vocabularies when modifying the dictionary structure. It also permits definition of a word **VOCS**, which will display the names of all vocabularies known to the system.

4.0 Implementation Technical Reference

To be supplied.

5.0 Definition of Terms

These are the definitions of the terms used within this Document.

address, byte
: An unsigned 32-bit number that locates an 8-bit byte in a standard FORTH address space over the range {0..4,294,967,295}. It may be a native machine address or a representation on a virtual machine, locating the addr-th byte within the virtual byte address space. Addresses are treated as unsigned numbers. See: "arithmetic, two's complement"

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

address, native machine
: The natural address representation of the host computer.

address, parameter field
: The address of the first byte of memory associated with a word definition for the storage of compilation addresses (in a colon definition), numeric data, text characters, etc.

arithmetic, two's complement
: Arithmetic is performed using two's complement integers within a field of either 32 or 64 bits as indicated by the operation. Addition and subtraction of two's complement integers ignore any overflow condition. This allows numbers treated as unsigned to produce the same results as if the numbers had been treated as signed.

byte
: An assembly of 8 bits. In reference to memory, it is the storage capacity for 8 bits. When read from memory, the value is sign extended to 32 bits.

character
: A 7-bit number the significance of which is given by the ASCII standard. When contained in a larger field,

the higher order bits are zero. In some contexts this also refers to an 8-bit value treated as an unsigned integer and zero-extended when read from memory.

compilation
: The action of converting text words from the input stream into an internal form suitable for later execution. When in the compile state, the compilation addresses of FORTH words are compiled into the dictionary for later execution by the address interpreter. Numbers are compiled to be placed on the data stack when later executed. Numbers are accepted from the input stream unsigned or negatively signed and converted using the value of BASE . See: "number" "number conversion" "interpreter, text"

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

definition
: See: "word definition"

dictionary
: A structure of word definitions in computer memory which is extensible and grows toward higher memory addresses. Entries are organized in vocabularies to aid location by name. See: "search order"

display
: The process of sending one or more characters to the current output device. These characters are typically displayed or printed on a terminal. The selection of the current output device is system dependent.

error condition
: An exceptional condition which requires action by the system which may be other than the expected function.

false
: A zero number represents the false state of a flag.

flag
: A number that may have one of two logical states, false or true. See: "false" "true"

free field format
: Numbers are converted using the value of BASE and then displayed with no leading zeros. A trailing space is displayed. The number of characters displayed is the minimum number of characters, at least one, to uniquely represent the number. See: "number conversion"

glossary
: A set of explanations in natural language to describe the corresponding computer execution of word definitions.

immediate word
: A word which executes when encountered during compilation or interpretation. Immediate words handle special cases during compilation. See, for example, IF LITERAL ." etc.

input stream
: A sequence of characters available to the system, for processing by the text interpreter. The input stream conventionally may be taken from the current input device (via the text input buffer). >IN , TIB and #TIB specify the input stream. Words using or altering >IN , TIB and #TIB are responsible for maintaining and restoring control of the input stream. The input stream extends from the offset value of >IN to the size of the input stream.

interpreter, address
: The machine code instructions, routine or other facilities that execute compiled word definitions containing compilation addresses.

interpreter, text
: The word definitions(s) that repeatedly accepts a word name from the input stream, locates the corresponding compilation address and starts the address interpreter to execute it. Text from the input stream interpreted as a number leaves the corresponding value on the data stack. Numbers are accepted from the input stream unsigned or negatively signed and converted using the value of BASE . See: "number" "number conversion"

load
: Redirection of the text interpreter's input stream to be from mass storage. This is the general method for compilation of new definitions into the dictionary.

mass storage
: Storage which might reside outside FORTH's address space. Mass storage data is made as standard files on the host system consisting of a stream of bytes (i.e., sequential files).

number
: When values exist within a larger field, the most-significant bits are zero. 32-bit numbers are represented in memory by addressing the first of four bytes at consecutive addresses. Double numbers are represented on the stack with the most-significant 32 bits (with sign) most accessible. Double numbers are represented in memory by two consecutive 32-bit numbers. The address of the least significant 32 bits is four greater than the address of the most significant 32 bits. See: "arithmetic, two's complement" "number types"

number conversion
: Numbers are maintained internally in binary and represented externally by using graphic characters within the ASCII character set. Conversion between the internal and external forms is performed using the current value of BASE to determine the digits of a number. A digit has a value ranging from zero to the value of BASE-1. The digit with the value zero is represented by the ASCII character "0" (position 3/0 with the decimal equivalent of 48). This representation of digits proceeds through the ASCII character set to the character "(" corresponding to the decimal value 9. For digits with a value exceeding 9, the ASCII graphic characters beginning with the character "A" (position 4/1 with the decimal equivalent 65) corresponding to the decimal value 10 are used. This sequence then continues up to and including the digit with the decimal value 71 which is represented by the ASCII character "" (position 7/14 with a decimal equivalent 126). A negative number may be represented by preceding the digits with a single leading minus sign, the character "-".

number types
: All number types consist of some number of bits. These bits are either arbitrary or are weighted. Signed and unsigned numbers use weighted bits. Weighted bits within a number have a value of a power of two beginning with the rightmost (least-significant) bit

having the value of two to the zero power. This weighting continues to the leftmost bit increasing the power by one for each bit. For an unsigned number this weighting pattern includes the leftmost bit; thus, for an unsigned 32-bit number the weight of the leftmost bit is 2,147,483,648. For a signed number this weighting pattern includes the leftmost bit but the weight of the leftmost bit is negated; thus, for a signed 32-bit number the weight of the leftmost bit is -2,147,483,648. This weighting pattern for signed numbers is called two's complement notation. Unspecified weighted numbers are either unsigned numbers or signed numbers; program context determines whether the number is signed or unsigned.

pictured numeric output
: The use of numeric output definitions which convert numerical values into text strings. These definitions are used in a sequence which resembles a symbolic 'picture' of the desired text format. Conversion proceeds from least-significant digit to most-significant digit, and converted characters are stored from higher memory addresses to lower.

program
: A complete specification of execution to achieve a specific function (application task) expressed in FORTH source code form.

receive
: The process of obtaining one character from the current input device. The selection of the current input device is system dependent.

recursion
: The process of self-reference, either directly or indirectly.

return
: The means of indicating the end of text by striking a key on an input device. The key used is system dependent. This key is typically called "RETURN", "CARRIAGE RETURN", or "ENTER".

search order
: A specification of the order in which selected vocabularies in the dictionary are searched. Execution of a vocabulary makes it the first vocabulary in the search order. The dictionary is searched whenever a

word is to be located by its name. This order applies to all dictionary searches unless otherwise noted. The search order begins with the last vocabulary executed and ends with FORTH , unless altered in a system dependent manner.

source definition
: Text consisting of word names suitable for compilation or execution by the text interpreter. Such text is usually arranged in screens and maintained on a mass storage device.

stack, data
: A last in, first out list consisting of 32-bit binary values. This stack is primarily used to hold intermediate values during execution of word definitions. Stack values may represent numbers, characters, addresses, boolean values, etc. When the name 'stack' is used alone, it implies the data stack.

stack, return
: A last in, first out list which contains the addresses of word definitions whose execution has not been completed by the address interpreter. As a word definition passes control to another definition, the return point is placed on the return stack. The return stack may cautiously be used for other values.

string, counted
: A sequence of consecutive 8-bit bytes located in memory by their low memory address. The byte at this address contains a count {0..255} of the number of bytes following which are part of the string. The count does not include the count byte itself. Counted strings usually contain ASCII characters.

string, text
: A sequence of consecutive 8-bit bytes located in memory by their low memory address and length in bytes. Strings usually, but not exclusively, contain ASCII characters. When the term 'string' is used alone or in conjunction with other words it refers to text strings.

structure, control
: A group of FORTH words which when executed alter the execution sequence. The group starts and terminates with compiler words. Examples of control structures:
DO ... LOOP DO ... +LOOP BEGIN ... WHILE ... REPEAT
BEGIN ... UNTIL IF ... THEN IF ... ELSE ... THEN
See: "9.9 Control Structures"

true
: A non-zero value represents the true state of a flag. Any non-zero value will be accepted by word as 'true'; all opcodes return a 32-bit value with all bits set to one when returning a 'true' flag.

vocabulary
: An ordered list of word definitions. Vocabularies are an advantage in separating different word definitions that may have the same name. More than one definition with the same name can exist in one vocabulary. The latter is called a redefinition. The most recently created redefinition will be found when the vocabulary is searched.

vocabulary, compilation
: The vocabulary into which new word definitions are appended.

word
: A sequence of characters terminated by one blank or the end of the input stream. Leading blanks are ignored. Words are usually obtained via the input stream.

word definition
: A named FORTH execution procedure compiled into the dictionary. Its execution may be defined in terms of machine code, as a sequence of compilation address, or other compiled words.

word name
: The name of a word definition. Word names are limited to 31 characters and may not contain an ASCII space. If two definitions have different word names in the same vocabulary they must be uniquely findable when this vocabulary is searched.

6.0 Glossary Conventions

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

before -- after

before	stack parameters before execution
after	stack parameters after execution

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

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

The Forth compiler supports all opcodes described in the BINAR Opcode Reference. Not all of these opcodes are included in this Forth Reference. The operation of the elided opcodes should be obvious from their description in the BINAR Opcode Reference.

6.1 Attributes

Capitalized symbols indicate attributes of the defined words:

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

6.2 Pronunciation

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

6.3 Stack Parameters

Unless otherwise stated, all references to numbers apply to 32-bit signed integers.

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

Stack Abbrv.	Number Type	Range in Decimal	Minimum # bits
char	character	{0..255}	8
byte	byte	{-128..127}	8
h	half-word	{-32768..32767}	16
n	number (weighted bits)	{-2G..2G}	32
+n	positive number	{0..2G}	32
u	unsigned number	{0..4G}	32
addr	address (same as u)	{0..4G}	32
d	double number	{-2**63..2**63}	64
+d	positive double number	{0..2**63}	64
ud	unsigned double number	{0..2**64}	64
flag	boolean	0=false, else=true	32
true	boolean	-1 (as a result)	32
false	boolean	0	32
sys	0, 1, or more system dependent stack entries	not applicable	32

Any other symbol refers to an arbitrary signed 32-bit integer unless otherwise noted.

7.0 Glossary

0< n -- flag "zero-less"

flag is true if n is less than zero (i.e. the sign bit is set).

0= n -- flag "zero-equals"

flag is true if n is zero.

0> n -- flag "zero-greater"

flag is true if n is greater than zero.

1+ n1 -- n2 "one-plus"

n2 is the result of adding 1 to n1.

1- n1 -- n2 "one-minus"

n2 is the result of subtracting 1 from n1.

16/ n1 -- n2 "sixteen-divide"

n2 is the result of dividing n1 by 16. This is true division using a series of 2/ operations, as opposed to arithmetic shift right operations.

2* n1 -- n2 "two-times"

n2 is the result of shifting n1 left one bit. A zero is shifted into the lowest order bit position.

2+ n1 -- n2 "two-plus"

n2 is the result of adding 2 to n1.

2- n1 -- n2 "two-minus"

n2 is the result of subtracting 2 from n1.

2/ n1 -- n2 "two-divide"

n2 is the result of dividing n1 by 2. This is true division, as opposed to an arithmetic shift right. Note that -1 2/ . produces 0 whereas -1 ASR . produces -1 .

4* n1 -- n2 "four-times"

n2 is the result of shifting n1 left two bits. Zeros are shifted into the lowest order bit positions.

4+ n1 -- n2 "four-plus"

n2 is the result of adding 4 to n1.

4/ n1 -- n2 "four-divide"

n2 is the result of dividing n1 by 4. This is true division using a series of 2/ operations, as opposed to arithmetic shift right operations.

8/ n1 -- n2 "eight-divide"

n2 is the result of dividing n1 by 8. This is true division using a series of 2/ operations, as opposed to arithmetic shift right operations.

`!` n addr -- "store"

n is stored at addr.

`"` -- addr +n "quote"
-- (compiling)

Used in the form: " ccc"

Compile a delimited string into the dictionary. Later execution will return the address and number of characters in the string, up to but not including the delimiting " (close-quote). The blank following " is not part of the string ccc.

`#` +d1 -- +d2 "sharp"

The remainder of +d1 divided by the value of BASE is converted to an ASCII character and prepended to the output string, moving from high to low memory addresses using HOLD. +d2 is the quotient and is maintained for further processing. This word has the effect of extracting one digit from +d1 and transferring it to an output string. It is typically used between <# and #> .

`#>` d -- addr +n "sharp-greater"

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

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

+d is converted, appending each resultant character into the pictured numeric output string until the quotient (see: #) is zero. A single zero is added to the output string if the number was initially zero. #S is typically used between <# and #> , and has the effect of transferring all remaining significant digits (one or more) to the output string.

#TIB -- addr "number-t-i-b"

The address of a variable containing the number of bytes in the text input buffer.

' -- addr "tick"

Used in the form: ' <name>

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

(-- I "paren"
-- (compiling)

Used in the form: (ccc)

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

`*` n1 n2 -- n3 "times"

n3 is the least-significant 32 bits of the arithmetic product of n1 times n2.

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

n1 is first multiplied by n2 producing an intermediate 64-bit result. n4 is the quotient of the intermediate 64-bit result divided by the divisor n3. The product of n1 times n2 is maintained as an intermediate 64-bit result for greater precision than the otherwise equivalent sequence: n1 n2 * n3 / . An error condition results if the divisor is zero or if the quotient falls outside of the range {-2,147,483,648 .. 2,147,483,647}.

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

n1 is first multiplied by n2 producing an intermediate 64-bit result. n4 is the remainder and n5 is the quotient of the intermediate 64-bit result divided by the divisor n3. A 64-bit intermediate product is used as for */ . An error condition results if the divisor is zero or if the quotient falls outside of the range {-2,147,483,648 .. 2,147,483,647}.

`+` n1 n2 -- n3 "plus"

n3 is the arithmetic sum of n1 plus n2.

`+!` n1 addr -- "plus-store"

n1 is added to the value at addr using the convention for + . This sum replaces the original value at addr.

+LOOP n -- "plus-loop"
sys -- (compiling)

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

, n -- "comma"

ALLOT space for n then store n at HERE 4- .

- n1 n2 -- n3 "minus"

n3 is the result of subtracting n2 from n1.

-ROT n1 n2 n3 -- n3 n1 n2 "dash-rote"

The top three stack entries are reverse rotated, bringing the second stack item to the top, the third item to second, and putting the first item to third.

-TRAILING addr1 +n1 -- addr2 +n2 "dash-trailing"

The character count +n1 of a text string beginning at addr1 is adjusted to exclude trailing spaces. If +n1 is zero, then +n2 is also zero. If the entire string consists of spaces, then +n2 is zero.

. n -- "dot"

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

```
."              --                              "dot-quote"
                --    (compiling)
```

Used in the form: ." ccc"

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

```
.(              --          I                   "dot-paren"
                --    (compiling)
```

Used in the form: .(ccc)

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

```
.FILES          --                              "dot-files"
```

Print the names of all files compiled by the system. These files are maintained in the linked list whose head is in the system variable FLINK .

```
.N              n +n --                         "dot-n"
```

n is converted using BASE and then displayed right aligned in a field +n characters wide. The display of n is unsigned and padded with leading zeros. If the number of characters required to display n is greater than +n, an error condition exists.

```
.S              n0 n1 ... nn --                 "dot-s"
```

Nondestructively display the contents of the data stack or the message "empty".

.R n +n -- "dot-r"

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

/ n1 n2 -- n3 "divide"

n3 is the quotient of n1 divided by the divisor n2. An error condition results if the divisor is zero or if the quotient falls outside of the range {-2,147,483,648 .. 2,147,483,647}.

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

n3 is the remainder and n4 the quotient of n1 divided by the divisor n2. An error condition results if the divisor is zero or if the quotient falls outside of the range {-2,147,483,648 .. 2,147,483,647}.

/STRING addr1 +n1 +n2 -- addr2 +n3 "slash-string"

Truncate the first +n2 characters of the string addr1 with length +n1 . addr2 = addr1 + n2; n3 = n1 - n2 . Used primarily by WORD .

: -- sys "colon"

A defining word executed in the form:

: <name> ... ;

Create a word definition for <name> in the compilation vocabulary and set compilation state. The search order is changed so that the first vocabulary in the search order is replaced by the compilation vocabulary. The compilation vocabulary is unchanged. The text from the input stream is subsequently compiled. <name> is called a "colon definition". The newly created word definition for <name> cannot be found in the dictionary until the corresponding ; or ;CODE is successfully processed. An error condition exists if a word is not found and cannot be converted to a number or if, during compilation from mass storage, the input stream is exhausted before encountering ; . sys is balanced with its corresponding ; .

; -- I "semi-colon"
sys -- (compiling)

Stops compilation of a colon definition, allows the <name> of this colon definition to be found in the dictionary, sets interpret state and compiles EXIT (or a system dependent word which performs an equivalent function). sys is balanced with its corresponding : .
See: EXIT :

< n1 n2 -- flag "less-than"

flag is true if n1 is less than n2.

<# -- "less-sharp"

Initialize pictured numeric output conversion. The words:

#> #S <# HOLD SIGN

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

<< -- "begin-microcode"

Initialize the micro assembler and set the CONTEXT vocabulary to MICROASM .

<> n1 n2 -- flag "not-equal"

flag is true if n1 is not equal to n2.

= n1 n2 -- flag "equals"

flag is true if n1 is equal to n2.

> n1 n2 -- flag "greater-than"

flag is true if n1 is greater than n2.

>IN -- addr "to-in"

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

>link addr1 -- addr2 "to-link"

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

>name addr1 -- addr2 "to-name"

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

>view addr1 -- addr2 "to-name"

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

>R n -- "to-r"

Transfers n to the return stack.

? addr -- "question"

Print the contents of addr.

?DNEGATE d1 n -- d2 "question-d-negate"

Apply the sign of n to the 64-bit number d1 on the stack. Equivalent to:

```
0< IF DNEGATE THEN
```

?DUP n -- n n "question-dupe"
or 0 -- 0

Duplicate n if it is non-zero.

?EXIT flag -- "question-exit"

Exit the current routine if the flag is zero.
Equivalent to:

```
0= IF EXIT THEN
```

?NEGATE n1 n -- n2 "question-negate"

Apply the sign of n to the 32-bit number on the stack.
Equivalent to:

```
0< IF NEGATE THEN
```

@ addr -- n "fetch"

n is the value at addr.

@EXECUTE addr -- "fetch-execute"

Execute the routine whose address is stored at addr.
Equivalent to: @ EXECUTE .

ABORT" flag -- "abort-quote"
-- (compiling)

Used in the form: flag ABORT" ccc"

At execution time, if flag is true the characters ccc, delimited by " (close-quote), are displayed and then a system dependent error abort sequence, including the function of ABORT , is performed. If flag is false, the flag is dropped and execution continues. The blank following ABORT" is not part of ccc.

ABORT n0 .. nn --

Clears the data stack and performs the function of QUIT . No message is displayed.

ABS n -- u "absolute"

u is the absolute value of n. If n is -2,147,483,648 then u is the same value.

ADC n1 n2 cin -- n3 cout "add-with-carry"

n3 is the result of adding n1 to n2 with the carry-in flag cin. The carry-out flag cout is determined by the addition.

AGAIN -- I
sys -- (compiling)

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

ALIGN --

Force the dictionary pointer to a word aligned value (i.e., a multiple of 4) by adding 0, 1, 2, or 3 to the variable H.

ALLOT n --

Allocates n bytes in the dictionary by adding the value n to the variable H.

AND n1 n2 -- n3

n3 is the bit-by-bit logical 'and' of n1 with n2.

AND! n1 addr -- "and-store"

n1 is ANDed with the value at addr using the convention for AND . This result replaces the original value at addr.

ASCII -- char "as-key"
-- (compiling)

Used in the form: ASCII ccc

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

ASIC! n1 asic:addr -- "a-sic-store"

Write n1 to the asic device whose address is asic:addr.

ASIC@ asic:addr -- n1 "a-sic-fetch"

Read n1 from the asic device whose address is asic:addr.

ASR n1 -- n2 "a-s-r"

n2 is the result of arithmetically shifting n1 right one bit. The highest order bit of n1 is duplicated and placed in the highest order bit of n2.

ASRN n1 n -- n2 "a-s-r-n"

n2 is the result of arithmetically shifting n1 right n bits. The highest order bit of n1 is replicated and placed in the highest order bits of n2.

B@ addr -- byte "b-fetch"

byte is the sign-extended contents of the byte at addr.

BASE -- addr

The address of a variable containing the current numeric conversion radix.

BEGIN -- I
-- sys (compiling)

Used in the form: BEGIN ... flag UNTIL
or BEGIN ... flag WHILE ... REPEAT

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

BL -- 32 "b-l"

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

BLANK addr u --

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

BYE --

Command the Binar host environment to terminate execution and return to DOS.

C! n addr -- "c-store"

The least-significant 8 bits of n are stored into the byte at addr.

C+! n addr -- "c-plus-store"

The lowest 8 bits of n are added to the byte value at addr. This sum replaces the original byte value at addr.

C, n -- "c-comma"

ALLOT one byte then store the least-significant 8 bits of n at HERE 1- .

C@ addr -- char "c-fetch"

char is the non-sign-extended contents of the byte at addr.

CELL -- 4

A system constant returning the number of bytes in one compilation cell.

CELLS n1 -- n2

n2 is the number of bytes used by n1 cells. In the Binar, n2 is n1 times 4. This word is used for transportability to 16 bit systems, where CELLS would multiply by 2 instead of 4.

CMOVE addr1 addr2 u -- "c-move"

Move u bytes beginning at address addr1 to addr2. The byte at addr1 is moved first, proceeding toward high memory. If u is zero nothing is moved.

CMOVE> addr1 addr2 u -- "c-move-up"

Move the u bytes at address addr1 to addr2. The move begins by moving the byte at (addr1 plus u minus 1) to (addr2 plus u minus 1) and proceeds to successively lower addresses for u bytes. If u is zero nothing is moved. (Useful for sliding a string towards higher addresses).

COLD --

System routine that is executed on reset. Performs all initialization, evaluates the word READY , and enters the Forth interpreter.

COMPILE --

Used in the form: : <name> ... COMPILE <namex> ... ;

When <name> is executed, the compilation address compiled for <namex> is compiled and not executed. <name> is typically immediate and <namex> is typically not immediate.

CONFIG! n1 -- "config-store"

Write n1 to the Binar configuration register.

CONFIG@ -- n1 "config-fetch"

Return the contents of the Binar configuration register as n1.

CONSTANT n --

A defining word executed in the form:

n CONSTANT <name>

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

CONTEXT -- addr

Returns the address of a variable which determines the dictionary search order.

CONVERT +d1 addr1 -- +d2 addr2

+d2 is the result of converting the characters within the text beginning at addr1+1 into digits, using the value of BASE , and accumulating each into +d1 after multiplying +d1 by the value of BASE . Conversion continues until an unconvertible character is encountered. addr2 is the location of the first unconvertible character.

COUNT addr1 -- addr2 +n

addr2 is addr1+1 and +n is the length of the counted string at addr1. The byte at addr1 contains the byte count +n. Range of +n is {0.255}

CR -- "c-r"

Displays a carriage-return and line-feed or equivalent operation.

CREATE --

A defining word executed in the form:

CREATE <name>

Creates a dictionary entry for <name>. After <name> is created, the next available dictionary location is the first byte of <name>'s parameter field. When <name> is subsequently executed, the address of the first byte of <name>'s parameter field is left on the stack. CREATE does not allocate space in <name>'s parameter field.

CURRENT -- addr

Returns the address of a variable specifying the vocabulary in which new word definitions are appended.

C_OR! n1 addr -- "c-or-store"

The lowest 8 bits of n1 are logically ORed with the byte value at addr using the convention for OR . This result replaces the original byte value at addr.

D0= d -- flag "d-zero-equals"

flag is true if d is zero.

D! d addr -- "d-store"

d is stored at the double-word beginning at addr.

D+ d1 d2 -- d3 "d-plus"

d3 is the arithmetic sum of d1 plus d2.

D- d1 d2 -- d3 "d-minus"

d3 is the result of subtracting d2 from d1.

D. d -- "d-dot"

The value of d is displayed in a free field format. A leading negative sign is displayed if d is negative.

D>R d -- "d-to-r"

Transfer d from the data stack to the return stack. Equivalent to:

```
>R >R
```

D>S d -- n "d-to-s"

Convert the signed double number d into a signed single number n.

D< d1 d2 -- flag "d-less-than"

flag is true if d1 is less than d2 according to the operation of < except extended to 64 bits.

D= d1 d2 -- flag "d-equal"

flag is true if d1 equals d2.

D@ addr -- d "d-fetch"

d is the value at the double-word starting at addr.

DABS d -- ud "d-absolute"

ud is the absolute value of d. If d is -2,147,4,648 then ud is the same value.

DASR d1 -- d2 "d-a-s-r"

d2 is the result of arithmetically shifting d1 right one bit. The highest order bit of d1 is duplicated and placed in the highest order bit of d2.

DBASE! addr -- "d-base-store"

Set the processor's DBASE register to the value addr.

DBASE+! n -- "d-base-plus-store"

Add n to the value in the DBASE register.

DBASE+_! n1 n2 -- "d-base-indexed-store"

Store n1 at the address generated by adding n2 to the contents of the dbase register.

DBASE+_@ n1 -- n2 "d-base-indexed-fetch"

Retrieve the value n1 at the address generated by adding n1 to the contents of the dbase register.

DBASE@ -- addr "d-base-fetch"

Retrieve the value in the DBASE register.

DCONSTANT d -- "d-constant"

A defining word executed in the form:

d DCONSTANT <name>

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

DDROP d -- "d-drop"

d is removed from the stack.

DDUP d -- d d "d-dupe"

Duplicate d.

DECIMAL --

Set the input-output numeric conversion base to ten.

DECOMPILER --

The vocabulary containing the component words of the Binar Forth decompiler utility.

DEFINITIONS --

The compilation vocabulary CURRENT is changed to be the same as the CONTEXT vocabulary.

DEPTH -- +n

+n is the number of 32-bit values contained in the data stack before +n was placed on the stack.

DISABLE --

Disable the processor interrupts.

DLSL d1 -- d2 "d-l-s-l"

d2 is the result of the logical shift of d1 one bit left. A zero is shifted into the lowest order bit position.

DLSR d1 -- d2 "d-l-s-r"

d2 is the result of the logical shift of d1 one bit right. A zero is shifted into the highest order bit position.

DMAX d1 d2 -- d3 "d-max"

d3 is the greater of d1 and d2.

DMIN d1 d2 -- d3 "d-min"

d3 is the lesser of d1 and d2.

DNEGATE d1 -- d2 "d-negate"

d2 is the two's complement of d1.

DO n1 n2 -- I
-- sys (compiling)

Used in the form: DO ... LOOP
or DO ... +LOOP

Begins a loop which terminates based on control parameters. The loop index begins at n2, and terminates based on the limit n1. See LOOP and +LOOP for details on how the loop is terminated. The loop is always executed at least once. For example: n DUP DO ... LOOP executes 4,294,967,296 times. sys is balanced with its corresponding LOOP or +LOOP .

DOES> -- addr "does"
-- (compiling)

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

: <namex> ... <create> ... DOES> ... ;

and then

<namex> <name>

where <create> is CREATE or any user defined word which executes CREATE . Marks the termination of the defining part of the defining word <namex> and then begins the definition of the execution-time action for words that will later be defined by <namex>. When <name> is later executed, the address of <name>'s parameter field is placed on the stack and then the sequence of words between DOES> and ; are executed.

DOS --

A vocabulary which contains the components of the Binar environment's MS-DOS interface.

DOVER d1 d2 -- d1 d2 d1 "d-over"

Copy d1 to the top of the stack.

DR> -- d "d-r-from"

Retrieve d from the return stack. Equivalent to:

R> R>

DR@ -- d "d-r-fetch"

Copy d from the return stack. DR@ is designed for correct operation when used with D>R and DR> .

DROP n --

n is removed from the stack.

DROT d1 d2 d3 -- d2 d3 d1 "d-rote"

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

DS! n addr -- "d-s-store"

Store n at the given address in the data stack RAM. Similar to PICK, but the address is not top-of-stack relative.

DS@ addr -- n "d-s-fetch"

Retrieve the value n from the given address in the data stack ram. Similar to PICK, but the address is not top-of-stack relative.

DSWAP d1 d2 -- d2 d1 "d-swap"

The top two double numbers are exchanged.

DUMP addr u --

List the contents of u bytes starting at addr. Each line of values is preceded by the address of the first value.

DUP n -- n n "dupe"

Duplicate n.

DVARIABLE -- "d-variable"

A defining word executed in the form:

DVARIABLE <name>

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

ELSE -- I
sys1 -- sys2 (compiling)

Used in the form: flag IF ... ELSE ... THEN

ELSE executes after the true part following IF . ELSE forces execution to continue at just after THEN . sys1 is balanced with its corresponding IF . sys2 is balanced with its corresponding THEN . See: IF THEN

EMIT n --

The least-significant 7-bits of n are displayed as an ASCII character.

EMPTY --

Truncate the entire dictionary to the GUARD point. Similar to FORGET, but no particular name is needed. See: GUARD FORGET

ENABLE --

Enable the processor interrupts.

ERASE addr u --

u bytes of memory beginning at addr are set to zero. No action is taken if u is zero.

err addr n --

Causes the system to return to the PRESET routine and print the specified error message.

EVALUATE addr +n --

The string from addr of length +n will be evaluated by the Forth interpreter. The program calling EVALUATE is responsible for not mucking around with the state of the system in a non-recoverable manner. EVALUATE preserves its input stream, and restores its values on exit. See: INTERPRET

EXECUTE addr --

The word definition indicated by addr is executed. An error condition exists if addr is not a compilation address (which will probably result in a system crash).

EXIT --

EXIT is used to compile a subroutine return instruction within a colon definition. An error condition exists if the top of the return stack does not contain a valid return point. May not be used within a do-loop.

EXPECT addr +n --

Receive characters and store each into memory. The transfer begins at addr proceeding towards higher addresses one byte per character until either a "return" is received or until +n characters have been transferred. No more than +n characters will be stored. The "return" is not stored into memory. No characters are received or transferred if +n is zero. All characters actually received and stored into memory will be displayed, with the "return" displaying as a space. See: SPAN

FALSE -- 0

Returns a constant value of zero.

FILES --

A vocabulary containing the names of the files that have been loaded or compiled by the system. See: .FILES

FILL addr u byte --

u bytes of memory beginning at addr are set to byte. No action is taken if u is zero.

FIND addr1 -- addr2 n

addr1 is the address of a counted string. The string contains a word name to be located in the currently active search order. If the word is not found, addr2 is the string address addr1, and n is zero. If the word is found, addr2 is the compilation address and n is set to one of two non-zero values. If the word found has the immediate attribute, n is set to one. If the word is non-immediate, n is set to minus one (true).

FOR n1 -- I
-- sys (compiling)

Used in the form: FOR ... NEXT

Begins a loop which terminates based on control parameters. The loop index begins at n1, and terminates when it changes from 0 to -1. See NEXT for details on how the loop is terminated. The loop is always executed at least once. For example: -1 FOR ... NEXT executes 4,294,967,296 times. sys is balanced with its corresponding NEXT .

FORGET --

Used in the form: FORGET <name>

If <name> is found in the compilation vocabulary, delete <name> from the dictionary and all words added to the dictionary after <name> regardless of their vocabulary. Failure to find <name> is an error condition. An error condition also exists if the compilation vocabulary is deleted.

FORTH --

The name of the primary vocabulary. Execution replaces the first vocabulary in the search order with FORTH . FORTH is initially the compilation vocabulary and the first vocabulary in the search order. New definitions become part of the FORTH vocabulary until a different compilation vocabulary is established. See: VOCABULARY

FP! addr -- "f-p-store"

Set the processor's FP register to the value addr. See: LOC_! LOC_+! LOC_@ etc.

FP+! n -- "f-p-plus-store"

Add n to the value in the FP register. This word is used as one way to allocate and deallocate memory resident activation records. See: LOC_! LOC_+! LOC_@ etc.

FP@ -- addr "f-p-fetch"

Retrieve the value in the FP register. See: LOC_! LOC_+! LOC_@ etc.

GUARD --

Marks the current dictionary state for later EMPTYing.

H! h addr -- "h-store"

Store the 16-bit value n at addr.

H, h -- "h-comma"

Allocate 2 bytes for a half-word, then store h at HERE 2- .

H@ addr -- h "h-fetch"

Retrieve the signed 16-bit value from addr.

HALT --

Stop processor execution. The only ways to leave the HALT state is by a processor reset.

HERE -- addr

The address of the next available dictionary location.

HEX --

Set the numeric input-output conversion base to sixteen.

HOLD char --

char is inserted into a pictured numeric output string. Typically used between <# and #>.

huh? flag -- "huh"

If the flag is zero, execute err with a message of " ? ". Otherwise return to the calling routine.

I -- w

w is a copy of the loop index. May only be used in the form:

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

I' -- w "i-prime"

Used within a colon definition executed only from within a do-loop to return the corresponding loop index.

IF flag -- I
-- sys (compiling)

Used in the form: flag IF ... ELSE ... THEN
or flag IF ... THEN

If flag is true, the words following IF are executed and the words following ELSE until just after THEN are skipped. The ELSE part is optional. If flag is false, the words from IF through ELSE , or from IF through THEN (when no ELSE is used), are skipped. sys is balanced with its corresponding ELSE or THEN .

IMMEDIATE --

Marks the most recently created dictionary entry as a word which will be executed when encountered during compilation rather than compiled.

IN? n1 n2 .. nn #n x -- flag "in-set"

Assuming a set of #n data items on the stack {n1 n2 .. nn} and a value x, determine if x is in the set.

INTERPRET --

Begin text interpretation at the character indexed by the contents of >IN relative to the start of TIB.

J -- w

w is a copy of the index of the next outer loop. May only be used within a nested DO-LOOP or DO-+LOOP in the form, for example:

DO ... DO ... J ... LOOP ... +LOOP

KEY? -- flag "key-query"

Flag is true if a character is available for KEY. Executing KEY? does not consume any pending keystrokes.

KEY -- n

The least-significant 7 bits of n is the next ASCII character received. All valid ASCII characters can be received. Control characters are not processed by the system for any editing purpose. Characters received by KEY will not be displayed.

LAST -- addr

A variable containing the address of the beginning of the last dictionary entry made, which may not yet be a complete or valid entry.

LEAVE --
-- (compiling)

Terminates execution at the next occurance of LOOP or +LOOP . The loop is terminated and loop control parameters are discarded. May only be used in the form:

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

LEAVE may appear within other control structures which are nested within the do-loop structure. More than one LEAVE may appear within a do-loop.

link> addr1 -- addr2 "from-link"

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

LITERAL -- n I
n -- (compiling)

Used in the form: [n] LITERAL

Compiles a system dependent operation so that when later executed, n will be left on the stack.

LOAD -- <name>

Used in the form: LOAD EXTENSIONS.4

Load the specified file. The filename is compiled into the FILES vocabulary. The filename may include a full path specification. See: FILES .FILES

LOC_! n1 [n2] -- "local-store"

Assuming n2 to be a numeric literal compiled immediately before the LOC_! opcode, store n1 into the address generated by adding n2 to the contents of the FP register. See: FP! FP+! FP@

Example of use: : ... 8 LOC_! ... ;
This example stores the top of stack value at the local address 8. The "8" must be some literal value or an error condition is reported by the compiler.

LOC_+! n1 [n2] -- "local-plus-store"

Assuming n2 to be a numeric literal compiled immediately before the LOC_+! opcode, add n1 to the value at the address generated by adding n2 to the contents of the FP register. See: LOC_!

LOC_@ [n1] -- "local-fetch"

Assuming n1 to be a numeric literal compiled immediately before the LOC_@ opcode, retrieve the value at the address generated by adding n1 to the contents of the FP register. See LOC_!

LOC_@_! n1 [n2] -- "local-indirect-store"

Assuming n2 to be a numeric literal compiled immediately before the LOC_@_! opcode, retrieve the address at the address generated by adding n2 to the contents of the FP register, then store n1 at this indirected address. See LOC_!

LOC_@_+ n1 [n2] -- n3 "local-fetch-plus"

Assuming n2 to be a numeric literal compiled immediately before the LOC_@_+ opcode, retrieve the value at the address generated by adding n2 to the contents of the FP register. Add the value fetched from this address to n1, returning n3. See LOC_!

LOC_@_@ [n1] -- n2 "local-indirect-fetch"

Assuming n1 to be a numeric literal compiled immediately before the LOC_@_@ opcode, retrieve the value at the address generated by adding n1 to the contents of the FP register, then use this value as an address for retrieving n2. See LOC_!

LOC_B@ [n1] -- byte "local-b-fetch"

Assuming n1 to be a numeric literal compiled immediately before the LOC_B@ opcode, retrieve the sign extended byte value at the address generated by adding n1 to the contents of the FP register. See LOC_!

LOC_C@ [n1] -- char "local-c-fetch"

Assuming n1 to be a numeric literal compiled immediately before the LOC_C@ opcode, retrieve the character value at the address generated by adding n1 to the contents of the FP register. See LOC_!

LOOP -- I
sys -- (compiling)

Increments the DO-LOOP index by one. If the new index was incremented across the boundary between limit-1 and limit on the return stack, the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding DO . sys is balanced with its corresponding DO . See: DO

LSLN n1 n2 -- n3 "l-s-l-n"

n3 is the result of a logical shift left of n1 by n2 bits. The lowest order bits of n3 are filled with zeros.

LSR n1 -- n2 "l-s-r"

n2 is the result of a logical shift right of n1 by 1 bit. The highest order bit of n2 is filled with a zero.

LSRN n1 n2 -- n3 "l-s-r"

n3 is the result of a logical shift right of n1 by n2 bits. The highest order bits of n3 are filled with zeros.

M+ n1 d1 -- d2 "m-plus"

d2 is the result of sign extending n1 to a double precision value and adding it to d1.

MAX n1 n2 -- n3

n3 is the greater of n1 and n2 according to the operation of > .

MICROASM -- "micro-a-s-m"

A vocabulary containing all of the components of the microcode assembler for the BINAR. See: << >>

MIN n1 n2 -- n3

n3 is the lesser of n1 and n2 according to the operation of < .

MOD n1 n2 -- n3

n3 is the remainder after dividing n1 by the divisor n2. An error condition results if the divisor is zero or if the quotient falls outside of the range {-2,147,483,648 .. 2,147,483,647}.

MOVE addr1 addr2 n --

Move n words from addr1 to addr2. This is much quicker than CMOVE, but can move data only on aligned addresses.

MRAM! n addr -- "m-ram-store"

Store the value n at addr in the micro-code ram.

MRAM@ addr - n "m-ram-fetch"

Retrieve the value from addr in the micro-code ram.

NEGATE n1 -- n2

n2 is the two's complement of n1, i.e., the difference of zero less n1.

NEXT -- I
sys -- (compiling)

Decrements the FOR-NEXT index by one. If the new index was decremented across the boundary between 0 and -1 on the return stack, the loop is terminated and loop control parameter is discarded. When the loop is not terminated, execution continues to just after the corresponding FOR . sys is balanced with its corresponding FOR . See: FOR

NIP n1 n2 -- n2

Drop the second item from the stack. Equivalent to: SWAP DROP .

NOP -- "no-op"

Stall the processor for 1 or 2 clock cycles (1 clock cycle for the 2OPS instruction format, 2 clock cycles for the CALL/EXIT/JNEXT instruction formats). Do nothing.

NOT n1 -- n2

n2 is the one's complement of n1.

ONE -- 1

The constant 1 . Defined so access is available to the opcode, but spelled so that typical access uses the literal form.

OPTIMIZER --

The vocabulary containing the components of the optimizing compiler.

OR n1 n2 -- n3

n3 is the bit-by-bit inclusive-or of n1 with n2.

OR! n1 addr -- "or-store"

n1 is logically ORed with the value at addr using the convention for OR . This result replaces the original value at addr.

ORDER --

Display the vocabulary names forming the search order in their present search order sequence. Then show the vocabulary into which new definitions will be placed.

OVER n1 n2 -- n1 n2 n1

Copy the second stack element n1 to the top of the stack.

PAD -- addr

The lower address of a scratch area used to hold data for intermediate processing. The address or contents of PAD may change and the data lost if the address of the next available dictionary location is changed.

PICK +n -- n

n is a copy of the +nth stack value, not counting +n itself. +n is valid for {0..the number of elements on stack-1}. Extreme care should be used with this word if some stack elements are be spilled out to memory, since it does not check for this condition.

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

QUERY --

Characters are received and transferred into the memory area addressed by TIB . The transfer terminates when either a "return" is received or the number of characters transferred reaches the size of the area addressed by TIB . The value of >IN is set to zero and the value of #TIB is set to the value of SPAN . WORD may be used to accept text from this buffer. See: EXPECT

QUIT --

Clears the return stack, sets interpret state, accepts new input from the current input device, and begins text interpretation. No message is displayed. This may be thought of as a "warm-start" of the kernel.

R+! [n] -- "r-plus-store"

Assuming that the literal N was compiled immediately before R+! (required for literal field content of the opcode), add N to the value on top of the return stack. See: >R R@ R>

R> -- n "r-from"

n is removed from the return stack and transferred to the data stack.

R@ -- n "r-fetch"

n is a copy of the top of the return stack.

READY --

The application initialization word. COLD always executes the most recent version of READY. See: COLD

REALIGN addr1 -- addr2 "re-align"

addr2 is addr1 incremented to the next highest multiple of the processor cell size, (i.e., a multiple of 4) by adding 0, 1, 2, or 3 to addr1 as appropriate.

RECURSE -- I
         -- (compiling)

Compile the compilation address of the definition being compiled to cause the definition to later be executed recursively.

REPEAT -- I
        sys -- (compiling)

Used in the form: BEGIN ... flag WHILE ... REPEAT

At execution time, REPEAT continues execution to just after the corresponding BEGIN . sys is balanced with its corresponding WHILE . See: BEGIN

REPLACED --

The vocabulary containing the original definitions of words that have been replaced. Words only appear here when they are unneeded for future compilation, but must still be present for static references.

ROLL +n --

The +nth stack value, not counting +n itself is first removed and then transferred to the top of the stack, moving the remaining values into the vacated position. +n is valid for {0..the number of elements on the stack-1}. Extreme care should be used with this word if some stack elements are be spilled out to memory, since it does not check for this condition.

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

ROT n1 n2 n3 -- n2 n3 n1 "rote"

The top three stack entries are rotated, bringing the deepest to the top.

RP! addr -- "r-p-store"

addr becomes the current return stack pointer.

RP@ -- addr "r-p-fetch"

addr is the address value of the return stack pointer.

RPLIM! n -- "r-p-limit-store"

n is written to the return stack limit registers. The upper 16 bits of n are the upper limit; the lower 16 bits of n are the lower limit. The limit is ignored unless the interrupts are ENABLEd. See: ENABLE DISABLE CONFIG@ CONFIG!

RPLIM@ -- n "r-p-limit-fetch"

n is the current value of the return stack limit register. The upper 16 bits of n are the upper limit; the lower 16 bits of n are the lower limit. See: RPLIM!
.

RTI config -- "return-from-interrupt"

config is the configuration register value, pushed on the data stack by the interrupt opcode executed when the interrupt procedure was entered. RTI restores the last state of the config register before returning to the interrupted routine.

S>D n -- d "s-to-d"

d is the sign-extended 32-bit value n.

SAVE-SYSTEM -- <name>

Used in the form: SAVE-SYSTEM FORTH.IMG

The current executable image of the Forth environment is saved as <name>. The preferred name extension is .IMG. This image is re-loadable by the host environment, but is not ROM-able.

SBASE! addr -- "s-base-store"

Set the processor's SBASE register to the value addr.

SBASE+! n -- "s-base-plus-store"

Add n to the value in the SBASE register.

SBASE+_! n1 n2 -- "s-base-indexed-store"

Store n1 at the address generated by adding n2 to the contents of the SBase register.

SBASE+_@ n1 -- n2 "s-base-indexed-fetch"

Retrieve the value n1 at the address generated by adding n1 to the contents of the SBase register.

SBASE@ -- addr "s-base-fetch"

Retrieve the value in the SBASE register.

SEE --

Used in the form: SEE <name>

where <name> is a word in the dictionary. Decompile the given name.

SHELL --

Invoke a DOS shell. All available memory is returned to DOS prior to the invocation. Return to the Binar environment by typing EXIT.

SIGN n --

If n is negative, an ASCII "-" (minus sign) is appended to the pictured numeric output string. Typically used between <# and #> .

SP! addr -- "s-p-store"

addr becomes the current data stack pointer.

SP@ -- addr "s-p-fetch"

addr is the address value of the data stack pointer SP@ was executed.

SPLIM! n -- "s-p-limit-store"

n is written to the data stack limit registers. The upper 16-bits of n are the upper limit; the lower 16-bits of n are the lower limit. The limit is ignored unless the interrupts are ENABLEd. See: ENABLE DISABLE CONFIG@ CONFIG!

SPLIM@ -- n "s-p-limit-fetch"

n is the current value of the data stack limit register. The upper 16-bits of n are the upper limit; the lower 16-bits of n are the lower limit. See: SPLIM!

SPACE --

Displays an ASCII space.

SPACES +n --

Displays +n ASCII spaces. Nothing is displayed if +n is zero.

SPAN -- addr

The address of a variable containing the count of characters actually received and stored by the last execution of EXPECT . See: EXPECT

STATE -- addr

The address of a variable containing the compilation state. A non-zero content indicates compilation is occurring, but the value itself is system dependent. A Standard Program may not modify this variable.

STRING char --

Used in the form: ASCII \ STRING THIS IS A TEST\

Compile ascii text as a counted string. The delimiter is not included.

SWAP n1 n2 -- n2 n1

The top two stack entries are exchanged.

SYSTEM --

The vocabulary containing a large number of extra pieces of the Binar Forth system. These pieces are necessary for the operation of the system, but are not typically used in application programs. They are in this vocabulary to unclutter the main Forth vocabulary.

THEN -- I
sys -- (compiling)

Used in the form: flag IF ... ELSE ... THEN
or flag IF ... THEN

THEN is the point where execution continues after ELSE , or IF when no ELSE is present. sys is balanced with its corresponding IF or ELSE . See: IF ELSE

TIB -- addr "t-i-b"

The address of the text input buffer. This buffer is used to hold characters when the input stream is coming from the current input device.

TRUE -- -1

A constant composed of all ones.

TUCK n1 n2 -- n2 n1 n2

Insert a copy of n2 under n1 on the data stack. Equivalent to: SWAP OVER .

TYPE addr +n --

+n characters are displayed from memory beginning with the character at addr and continuing through consecutive addresses. Nothing is displayed if +n is zero.

U. u -- "u-dot"

u is displayed as an unsigned number in a free-field format.

U.R u +n -- "u-dot-r"

u is converted using the value of BASE and then displayed as an unsigned number right aligned in a field +n characters wide. If the number of characters required to display u is greater than +n, an error condition exists.

U< u1 u2 -- flag "u-less-than"

flag is true if u1 is less than u2.

U<= u1 u2 -- flag "u-less-or-equal"

flag is true if u1 is less than or equal to u2.

U> u1 u2 -- flag "u-greater-than"

flag is true if u1 is greater than u2.

UM* u1 u2 -- ud "u-m-times"

ud is the unsigned product of u1 times u2. All values and arithmetic are unsigned.

UM/MOD ud u1 -- u2 u3 "u-m-divide-mod"

u2 is the remainder and u3 is the quotient after dividing ud by the divisor u1. All values and arithmetic are unsigned. An error condition results if the divisor is zero or if the quotient lies outside the range {0 .. 4,294,967,295}.

UN addr -- "uncompile"

addr is the start of a disassembly of a Binar program. Dis-assembly continues for 20 opcodes or until a key is hit.

UNTIL flag -- I
sys -- (compiling)

Used in the form: BEGIN ... flag UNTIL

Marks the end of a BEGIN-UNTIL loop which will terminate based on flag. If flag is true, the loop is terminated. If flag is false, execution continues to just after the corresponding BEGIN . sys is balanced with its corresponding BEGIN . See: BEGIN

VAL, .. n3 n2 n1 #n -- "val-comma"

Compile #n words from the stack into the dictionary as a series of literals. The words are placed into the dictionary in reverse order, so that at run time the words will be on the stack in the same order as at compile time.

VAL? addr -- ... 0 "val-query"
or addr -- ... n 1
addr -- ... d 2

Convert the count and character string at addr to a binary number using the value of BASE . If numeric conversion is not possible, return a false flag. If conversion is successful, return the value as a sequence of 32-bit stack items (most-significant part highest) and the number of 32-bit stack items required for the number. The string may contain a preceding minus sign.

VARIABLE --

A defining word executed in the form:

VARIABLE <name>

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

VOCABULARY --

A defining word executed in the form:

VOCABULARY <name>

A dictionary entry for <name> is created which specifies a new ordered list of word definitions. Subsequent execution of <name> replaces the first vocabulary in the search order with <name>. When <name> becomes the compilation vocabulary new definitions will be appended to <name>'s list. See: DEFINITIONS

VOCS --

Display the names of all vocabularies known to the system.

WDUMP addr n -- "word-dump"

List the contents of u addresses starting at addr. Each line of values is preceded by the address of the first value. The display is organized into 32-bit units.

WHILE flag --
sys1 -- sys2 (compiling)

Used in the form: BEGIN ... flag WHILE ... REPEAT

Selects conditional execution based on flag. When flag is true, execution continues to just after the WHILE through to the REPEAT which then continues execution back to just after the BEGIN . When flag is false, execution continues to just after the REPEAT , exiting the control structure. sys1 is balanced with its corresponding BEGIN . sys2 is balanced with its corresponding REPEAT . See: BEGIN

WORD char -- addr

Generates a counted string by non-destructively accepting characters from the input stream until the delimiting character char is encountered or the input stream is exhausted. Leading delimiters are ignored. The entire character string is stored in memory beginning at addr as a sequence of bytes. The string is followed by a blank which is not included in the count. The first byte of the string is the number of characters {0..255}. If the string is longer than 255 characters, the count is unspecified. If the input stream is already exhausted as WORD is called, then a zero length character string will result. If the delimiter is not found, the value of >IN is the size of the input stream. If the delimiter is found >IN is adjusted to indicate the offset to the character following the delimiter. #TIB is unmodified. The counted string returned by WORD may reside in the "free" dictionary area at HERE or above. Note that the text interpreter may also use this area.

WORDS --

Used in the form: WORDS
or WORDS XYZ
or WORDS *.*

If WORDS is not followed on its command line by any text, list the word names in the first vocabulary of the currently active search order. Otherwise the text following is used as a pattern. WORDS will then display all dictionary entries, regardless of vocabulary, that contain the pattern string. A special string is *.* , which causes WORDS to display all words in all vocabularies.

XOR n1 n2 -- n3 "x-or"

n3 is the bit-by-bit exclusive-or of n1 with n2.

[-- I "left-bracket"
-- (compiling)

Sets interpret state. The text from the input stream is subsequently interpreted. For typical usage see LITERAL . See:]

['] -- addr "bracket-tick"
-- (compiling)

Used in the form: : ... ['] <name> ... ;

Compiles the compilation address addr of <name> as a literal. When the colon definition is later executed addr is left on the stack. An error condition exists if <name> is not found in the currently active search order. See: LITERAL

[COMPILE] -- I "bracket-compile"
-- (compiling)

Used in the form: [COMPILE] <name>

Forces compilation of the following word <name>. This allows compilation of an immediate word when it would otherwise have been executed.

] -- "right-bracket"

Sets compilation state. The text from the input stream is subsequently compiled. For typical usage see LITERAL . See: [

\ -- I "back-slash"

Comment to end of line. A space must be used after the \ for correct operation.

\OPT -- "break-optimization"

Causes the optimizing compiler's finite state machine to be reset, disallowing opcode, literal, call, and exit compression.

\\ -- "double-back-slash"

Used in the form: \\ dir c:\xyz*.dat
\\ <any valid dos command line>

Used to execute a single dos command line without shelling out to dos.

{ -- "begin-set"

Mark the beginning of a set of data to be evaluated by IN? .

} -- # "end-set"

Terminate the set, leaving the count of the number of items in the set for IN? .