MPE Forth 5 for RTX2000

Glossary

Forth 5 for RTX2000

GLOSSARY

RTX2000/1A/10 Target

Version: 5.100

User Manual

Revision: 1.02

Date: 17 February 1994

Package No:	

For technical support:

Please contact your supplier

For further information:

MicroProcessor Engineering Limited
133 Hill Lane, Southampton
SO1 5AF, UK
Tel: 0703 631441
Fax: 0703 339691
Email: mpe@cix.compulink.co.uk

MPE Forth 5 for RTX2000

Acknowledgements

MPE would like to thank the following people for all their involvement in the production of this product:

Jon Lee, Stephen Pelc, Paul Gallienne, Gary Ellis

Microprocessor Engineering Limited
133 Hill Lane
Southampton
SO1 5AF, UK

Table of contents

Glossary Notation

The documentation of the glossaries uses a methodology based on that used for the Forth-83 Standard document. As this is not a standard document, but is supposed to be a user manual, we have taken some liberties to make this manual easier to look at.

Glossary word order

The glossary definitions are listed in the following order:-

, . : ; ! ? “ ‘ ‘ ( ) [ ] { } $ + - * / ^ < = > % # & @ \ _ | ~ 0..9 A..Z

Forth words

Forth word names are capitalized throughout, and are shown in bold in the text.

Stack Notation

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

before — after

where:

‘before’ means the stack parameters before execution

and

‘after’ means 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 execution time. If it applies at compile time, the line is followed by: (compiling) .

Stack Parameters

Unless otherwise stated all references to numbers apply to 16-bit signed integers. The implied range of values is shown as {from..to}. The content of an address is shown by double braces, particulary for the contents of variables, e.g., BASE {2..72}. 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.

Abbrv.	type	range & field size
flag	boolean	0=false, else=true 16
true	boolean	-1 (as a result) 16
false	boolean	0 16
b	bit	{0..1} 1
char	character	{0..127} 7
8b	8 bits	not applicable 8
16b	16 bits	not applicable 16
n	number	{-32,768..32,767} 16
+n	+ve int	{0..32,767} 16
u	unsigned	{0..65,535} 16
w	unspecified weighted number (n or u)	{32,768..65,535} 16
addr	address	{0..65,535} 16
32b	32 bits	not applicable 32
d	double number	{-2,147,483,683,648.. 2,147,483,647} 32
+d	positive double number	{0..2,147,483,647} 32

ud	unsigned double number	{0..4,294,967,295}	32
wd	unspecified weighted double number (d or ud)	{-2,147,483,648..4,294,967,295}	32
f	floating point number		
{			
sys	0, 1, or more system dependent entries	not applicable	

Any other symbol refers to an arbitrary signed 16-bit integer in the range {-32,768..32,767}, unless otherwise noted. Because of the use of two's complement arithmetic, the signed 16-bit number (n) -1 has the same bit representation as the unsigned number (u) 65,535. Both of these numbers are within the set of unspecified weighted numbers (w). On many occasions where the context is obvious, informal names are used to make the documentation easier to use.

Input Text

<name>
An arbitrary Forth word accepted from the input stream. This notation refers to text from the input stream, not to values on the data stack.

ccc
A sequence of arbitrary characters accepted from the input stream until the first occurrence of the specified delimiter character. The delimiter is accepted from the input stream, but it is not one of the characters ccc and is therefore not otherwise processed. This notation refers to text from the input stream, not to values on the data stack. Unless noted otherwise, the number of characters accepted may be from 0 to 255.

Other markers

The following markers may appear after a word's stack comment. These markers indicate certain features and peculiarities of the word.

C	The word may only be used during compilation.
I	The word is immediate. It will be executed even during compilation, unless preceded by the word **[COMPILE]**.

Compiler Directives

This glossary details the cross compiler directives.

[\\] —
"bracket-stop-opt"

Used during compilation to break the optimisation sequence at this point. Opcodes either side of **[\\]** will not be merged. See **NO-OPTIMIZE OPTIMIZE**

;P —

End Page. Causes **FROM** and **FROM-FILE** to stop using the current page. Has the same effect as **;S** in screen files.

```
;P
```

(— ;I
"open-paren"

The comment in a text file differs from the screen file version. The paranthesis comment works over multiple lines. The end of comment is marked by a white-space-delimited closing parenthesis, e.g.

```
... 2DUP ( save adr # for later )  INIT ...
    ...
CLOBBER  ( NOTE:
         use the operating system function
         here to shut off access
         )
        ...
```

)ELSE(— ; I
"paren-else-paren"

The words **IF( )ELSE( )ENDIF** permit conditional compilation and interpretation. They are an analogue of **IF ELSE ENDIF** and are used in the forms:

```
flag
IF( <if flag is true> )ENDIF
```

```
flag
IF( <if flag is true>
)ELSE( <if flag is false>
)ENDIF
```

If the flag is true the words after **IF(** are interpreted or compiled, but if the flag is false the words after **)ELSE(** are interpreted/compiled. The **)ELSE(** clause is optional.

)ENDIF — ; I
"paren-endif"

See **IF(** and **)ELSE(**.

ALIAS —
"alias"

A defining word used to create a second symbol name for a word. Used in the form:

```
ALIAS name1 name2
```

```
ALIAS (EMIT) SEND-BYTE
```

This feature is useful where a package is written using one word name, and the required function can be provided by a word of a different name elsewhere in the system. In the example above the CFA of **(EMIT)** will be compiled for a reference to **SEND-BYTE**.

ALIGN-ODD —
"align-odd"

Used to align CFAs onto odd addresses (so that PFAs are on even ones). The link fields are still forced to a even address.

```
ALIGN-ODD
```

ALL — 1 -1
"all"

Used before **FROM**, **DISPLAY**, or **CONTENTS**, **ALL** will put the first and last page numbers on the stack so that ALL the pages are specified. e.g.

```
ALL DISPLAY MYFILE.FTH
```

ALLOT n —
"allot"

Allots space in the current code page area of a ROM/RAM target.

```
CREATE AREA    \ — addr ; returns pointer to ROM area
 0100 ALLOT                \ reserve space in ROM
55 AREA !C                 \ which can be filled in later
66 AREA 2+ !C              \ Note the use of !C
```

ALLOT-RAM n —
"allot-ram"

Allots space in the current data page area of a ROM/RAM target. Its function is to allow uninitialised space to be reserved in RAM.

```
THERE CONSTANT POINTER  \ pointer to RAM area
 0100 ALLOT-RAM         \ reserve space in RAM
```

ALONE n — n n
"alone"

A modifier used before **FROM** or **FROM-FILE** or other text file words to describe a single page, for example to compile page 5 from the current text file use:

```
5 ALONE FROM
```

BASE-36 —
"base-thirty-six"

Sets the current base to 36 (decimal). This apparently strange operation allows upper-case characters and digits to be encoded as radix-36 numbers, reducing the memory required. This technique is inherited from earlier incarnations of the compiler, where it was often used to encode the processor type as a double number.

BOUNDING size mask end-gap start-skip —
"bounding"

This word is used in a ROM/RAM system to skip parts of memory that may be occupied by existing ROMS. Such a situation is sometimes found in domestic computers, or when modifying systems that use partial memory decoding.

```
n1 n2 n3 n4 BOUNDING
```

The first parameter n1 declares the size of the EPROM in bytes.

Parameter n2 is a bit mask identifying which address lines select the EPROM. For example an 8k EPROM of size 02000 (hex) is controlled by address lines A15,14,13 and will have a mask value of 0E000 (hex).
Parameter n3 specifies how far before the end of the EPROM the compiler stops to ask the user whether it should step to the next EPROM. This limit is used to prevent the last word in an EPROM being compiled across an EPROM boundary. If the answer is no, the question will be repeated before each word is compiled until the answer is yes or the EPROM boundary is crossed. When the yes answer is given, the compiler steps to the start of the next EPROM, and repeats the question. If the answer is again yes, the complete EPROM is skipped. Note that specifying a value of say 0100 does not guarantee to reserve 0100 free bytes at the end of the EPROM, it only specifies at what point the compiler starts asking what needs to be done.
Parameter n4 specifies how many bytes are skipped at the start of each EPROM.
The example below is for an 8k byte EPROM such that the compiler starts checking 128 bytes (080h) before the end of the EPROM, and the first 256 bytes (0100h) are skipped at the start of each EPROM.

```
HEX 02000 0E000 080 0100 BOUNDING
```

CODE-PAGE start end page-id — ; <name>
"code-page"

Defines the start and end of a page of ROM in a system. The label <name> is the page name of the area of memory. The page-id is a unique identifier for the page and is used by the page switching word **PAGE-WORD.** The code generated for the page will be saved with the filename <name> and extension .IMG. For example,

```
$4000 $7FFF 2 CODE-PAGE PAGE2
```

defines an area of memory from 4000h to 7FFFh with page-switching code 2. The page's image is saved with the name PAGE2.IMG. See also **DATA-PAGE**, **KERNEL** and **KERNEL-RAM**.

DATA-PAGE start end page-id — ; <name>
“data-page”

Defines the start and end of a page of RAM in a system. The label <name> is the page name of the area of memory. The page-id is a unique identifier for the page and is used by the page switching word **PAGE-WORD.** For example,

```
$8000 $BFFF 5 DATA-PAGE PAGE5
```

defines an area of memory from 8000h to BFFFh with page-switching code 5. See also **CODE-PAGE**, **KERNEL** and **KERNEL-RAM**.

CON: —
“to-con”

Direct the symbol table log output to the screen. The default output device. If required this directive has to be used before the command **CROSS-COMPILE**.

```
CON:
```

CROSS-COMPILE —
“cross-compile”

This word tells the system to start cross compiling. Until this point the compiler is a conventional Forth system with an application (the cross compiler) loaded but not running. At this stage extensions and assembler macros can be loaded. After **CROSS-COMPILE** has executed the compiler ‘pulls down the shutters’ to seal itself off, and then treats all code as target code and compiler directives.

```
CROSS-COMPILE
```

EMU-BASE address —
“emu-base”

Sets the base address of the LePROM emulator for the cross compiler. The tsr emulator drivers must be installed to use this word.

EQU n — ; <name>
“equ”

Creates a compile-time equate of value n. When the equate is referred to in the target code the value assigned to the equate is used as a literal. An equate only exists during the run-time of the cross compiler. Equates may be redefined like macro-assembler

set-symbols. Equates may be used wherever numbers may be used, they are just a means of naming a number.

```
HEX
0FF80 EQU IO-PORT
```

EXTERNAL —
"external"

The following words are generated with headers containing up to 31 characters, provided that the directive **NO-HEADS** has not been used.

```
EXTERNAL
```

FILE: — ; <filename>
"to-file"

Direct the symbol table log output to the file <filename>. If required this directive has to be used before the command **CROSS-COMPILE**.

```
FILE: SYMBOLS.LOG
```

FINIS —
"finis"

This word stops the cross compiler. A cleaning-up operation is performed, final reports issued, the output file or EPROM emulator closed, and finally the compiler exits to host/target mode.

```
FINIS
```

FINIS-CODE-PAGE —
"finis-page"

This word is used to finish off the compilation of a code page, and return to cross compiling the kernel. This must not be used on the kernel page.

FROM first last —
"from"

Get text input from a range of pages in the current file. The first and last pages to be used are supplied on the stack. Files can be nested 16 deep; that is files can include input from other files. **FROM** is like **THRU** or **THRU-USING** with screen files, e.g.

```
4 10 FROM
```

FROM-FILE first last — ; <filename>
"from-file"

Get text input from a range of pages in the file <file-name> typed after **FROM-FILE**. The first and last pages to be used are supplied on the stack. Files can be nested; that is files can include input from other files. **FROM-FILE** is like **THRU** or **THRU-USING**, e.g.

```
4 10 FROM-FILE MYFILE.FTH
```

HEADS? — t/f ; I
"heads-query"

An immediate equate that returns false if the **NO-HEADS** directive has been used. The function of this equate is to return a value for conditional compilation of the interpreter and compiler layers if heads are needed, for example:

```
HEADS?
IF( 7 LOAD-USING INTERACTIVE )ENDIF
```

HOST&TARGET —
"host-and-target"

Used after **TARGET-ONLY** to allow defining words to be handled again. Some special cases of defining words cannot be handled by the cross compiler, but are required for target execution. **TARGET-ONLY** and **HOST&TARGET** handle this situation.

```
HOST&TARGET
```

HOST-COMPILATION—
"host-compilation"

Temporarily turns off the cross compiler so that the following code can be compiled into the cross compiler itself. Cross compilation is restarted by **TARGET-COMPILATION**.

I: — ; <wordname>
"i-colon"

Not really a directive, rather an auxiliary version of **:**. **I:** is used instead of **:** to create an immediate word that will exist in, and can be executed by, the cross compiler in the same way that defining words are handled. The same rules as for defining words apply to words created by **I:**.

```
I: <name>
......
;     IMMEDIATE
```

Note that **IMMEDIATE** only affects the target code, **I:** is needed to enable the analogue to be built into the cross compiler.

IF(n — ; I
"if-paren"

The words **IF( )ELSE( )ENDIF** permit conditional compilation and interpretation. They are an analogue of **IF ELSE ENDIF** and are used in the forms:

```
flag
IF(  <if flag is true>  )ENDIF
```

```
flag
IF(  <if flag is true>
)ELSE(  <if flag is false>
)ENDIF
```

If the flag is true the words after **IF(** are interpreted or compiled, but if the flag is false the words after **)ELSE(** are interpreted/compiled. The **)ELSE(** clause is optional.

IN-EMULATOR addr — ; <name>
"in-emulator"

The code generated for page <name> is redirected into an EPROM emulator. The address addr is the start of the image within the emulator. e.g.

```
$2000 IN-EMULATOR KERN
```

will send the page KERN to your emulator, starting at offset 2000h. This directive must be used befor the first use of

```
USE-CODE KERN
```

INTERNAL —
"internal"

This directive causes the following words to be generated without headers. The cross compiler still knows they are there, but they will not be visible to the interpreter on the target system.

```
INTERNAL
```

IS-FENCE —
"is-fence"

Marks the last word defined as being the last word in the protected dictionary. In a RAM system, the cross compiler places the NFA of this word at the location **INIT-FENCE**. The target start-up code

can then access this location to find the initial value of the variable **FENCE**.

```
L: INIT-FENCE  0 ,                          \ reserve space

.............

: FORTH-83  ;  IS-FENCE                     \ fills INIT-FENCE
```

L: — ; <word-name>
"ell-colon"

Creates a label with the address returned by **HERE**, i.e. the current location in the dictionary. Normally only used during interpretation and assembly, but can be used during compilation if surrounded by **[** and **]**.

```
L: DATA-SLOT  0 ,

: <word>
......
[ L: INSIDE-WORD ]
......
;
```

KERNEL start end — ; <name>
"kernel"

Defines the start and end of the fixed kernel ROM in a system. <name> is the page name of the area of memory. The generated image file will have the name <name> and the extension .IMG. For example,

```
$0000 $7FFF KERNEL KERNROM
```

defines an area of memory from 0h to 7FFFh. The kernel page's image is saved with the name KERNROM.IMG. See also **DATA-PAGE**, **CODE-PAGE** and **KERNEL-RAM**.

KERNEL-RAM start end page-id —; <name>
"kernel-ram"

Defines the start and end of the fixed kernel RAM in a paged system. <name> is the page name of the area of memory. For example,

```
$8000 $FFFF 0 KERNEL-RAM KERNRAM
```

defines an area of memory from 0h to FFFFh with page-switching code of 0. See also **KERNEL**, **DATA-PAGE CODE-PAGE**

LABEL addr — ; <word-name>
"label"

Used during interpretation to create a label at an arbitrary location to satisfy a forward reference from a code definition or code fragment.

```
THERE LABEL MY-DATA
```

LOAD n —
"load"

The contents of screen n of the current screen file are compiled, e.g.

```
10 LOAD
```

LOAD-USING n — ; <pathname>
"load-using"

The contents of screen n of the given screen file are compiled. If no extension is given, the compiler will add the extension '.SCR', e.g.

```
10 LOAD-USING A:\ROM-IO
```

LOG —
"log"

Generate a full symbol table log.

```
LOG
```

MAKE-TURNKEY — ; <name>
"make-turnkey"

Used to make your application a turnkey or autostarting system. The word <name> is the name of your application that you want to be run at startup. For example,

```
MAKE-TURNKEY MY-APP
```

MEM-END addr —
"mem-end"

Used to define the end of memory in the target system. For a ROM-based system with ROM at 0000, and RAM from 08000 to 09FFF, the following sequence would be used:

```
HEX
0000 7FFF  KERNEL Rom-targ
8000 9FFF 0 KERNEL-RAM Rom-data
A000 MEM-END
```

NO-HEADS —
"no-heads"

Disables generation of heads, overriding **EXTERNAL** and **TARGET-WIDTH** completely. This directive can be used when the application is complete to remove ALL heads from the system without having to go through the source code removing all occurences of **EXTERNAL** and **TARGET-WIDTH**.

```
NO-HEADS
```

NO-LOG —
"no-log"

Generate a reduced symbol table log.

```
NO-LOG
```

NO-OPTIMIZE —
"no-optimise"

This directive disables the opcode optimiser, which can merge several opcodes into one, so producing smaller and faster code. The default state is **OPTIMIZE**. The usual reason to turn the optimiser off is in order to build a table of opcodes.

For example:

```
CREATE OPCODE-TABLE
 ]  @ ! + 0< - R> DROP >R R@  [
```

Without any optimisation control, several elements of this table would be merged, producing an incorrect result. The optimiser can be 'broken' using the **[\\]** function as follows:

```
CREATE OPCODE-TABLE
 ]  @ [\\] ! [\\] + [\\] 0< [\\] - [\\] R> [\\]
 DROP [\\] >R [\\] R@ [\\] [
```

The use of **[\\]** after each opcode forces the optimiser not to merge any opcodes, but makes the code longer and difficult to read or understand. A better solution is to use **NO-OPTIMIZE** and **OPTIMIZE** to turn the optimiser off for a short while.

```
NO-OPTIMIZE
```

```
CREATE OPCODE-TABLE
 ] @ ! + 0 - R DROP R R@ [

OPTIMIZE
```

NO-TAIL-OPT —
"no-tail-opt"

A procedure that ends in a call to another procedure, followed by a return can often by optimised by just branching to the other procedure. Because some Forth words can make assumptions about the use of the return stack, this optimisation is controlled separately from all other optimisations. The default is **NO-TAIL-OPT**. It is recommended that you develop the code without this optimisation, and then turn it on for final debugging. **NO-TAIL-OPT** and **TAIL-OPT** can be used around single words if required.

```
: A  ..... ;
: B  ..... A ;
```

If **TAIL-OPT** is active the last call to **A** will be converted to a jump, saving two bytes and one clock cycle. If this is undesirable use the code below.

```
: A  ..... ;
NO-TAIL-OPT
: B  ..... A ;
TAIL-OPT
```

ONWARDS n — n -1
"onwards"

A modifier used before **FROM** or **FROM-FILE** or other text file words to use the text from a specified page to the end, for example to compile page 5 to the end from the current text file use:

```
5 ONWARDS FROM
```

OPTIMIZE —
"optimise"

This directive enables the opcode optimiser. For details see **NO-OPTIMIZE**.

ORG addr —
"org"

A directive used to set the dictionary pointer to the given address.

```
HEX
2080 ORG
```

OUTPUT-EMULATOR eprom-type bus-width —
"output-emulator"

Defines the EPROM type and bus width of the EPROM emulator the compiler will use. This directive switches all target memory words to use the EPROM emulator drivers instead of the output file drivers. Predefined constants exist to define the EPROM type and bus width. The EPROM type is one of:

```
E2764 E27128 E27256 E27512
```

and the bus-width is one of:

```
8BIT 16BIT 32BIT
```

For example:

```
E27256 16BIT OUTPUT-EMULATOR
```

The emulator tsr driver must be installed to use this word.

PAGED-VOCABULARY<name> —
"paged-vocabulary"

Used to make pages interactive. When a page is defined, a page vocabulary should be created as well. The vocabulary must be created in the page <name> i.e.

```
USE-CODE <name>
PAGED-VOCABULARY <name>
```

PRN: —
"to-prin"

Direct the cross-compiler log output to the printer. If required this directive has to be used before the command **CROSS-COMPILE**.

```
PRN:
```

PTO —
"p-t-o"

Please Turn Over. This word causes **FROM** and **FROM-FILE** to stop using the current page and to start on the next. (The same effect as —> in screen files.)

```
PTO
```

RESTART — ; <FileName>
"restart"

Continue cross compilation from the position saved under the given file name. The cross compiler saved position files having been generated using the directive **SUSPEND** during an earlier cross compilation.

```
RESTART KERNEL
```

SUSPEND — ; <filename>
"suspend"

Stop the present cross compilation, saving cross compiler information to disc under the given filename, the cross compiler then returning to host/target mode. Note that the file name should **not** include an extension, the cross compiler supplies its own. The directive **RESTART** is used to resume cross compilation later.

```
SUSPEND KERNEL
```

TAIL-OPT —
"tail-opt"

Turns on branch optimisation. The default is OFF. See **NO-TAIL-OPT**

TARGET-COMPILATION—
"target-compilation"

Re-enable cross compilation after it has been turned off by **HOST-COMPILATION**.

TARGET-WIDTH n—
"target-width"

Sets the maximum number of characters in the name field to be n. A maximum of 31 characters is imposed by the compiler.

```
7 TARGET-WIDTH
```

THRU n1 n2 —
"through"

The contents of screens n1 to n2 inclusive of the current screen file are compiled, e.g.

```
DECIMAL 7 23 THRU
```

It is good practice to define the number base just before **LOAD** or **THRU** as 'other people' sometimes forget to restore the base at the

end of a screen, or it might be house policy only to define the base where it matters. In this instance it does.

THRU-USING n1 n2 — ; <pathname>
"through-using"

The contents of screens n1 to n2 of the given screen file are compiled. If no extension is given, the compiler will add the extension '.SCR'.

USE — ; <text-file-name>
"use"

Set the default text file you wish to **USE** (like "**USING** xxx.scr" for screen files). The file defaults to NUL.FTH.

```
USE TEXTFILE.FTH
```

USE-CODE <name> —
"use-code"

Used to select which page the compiled code is generated for. Any code following **USE-CODE** will be compiled into the page <name>. This will continue until the compilation is finished, **FINIS-CODE-PAGE**, or another **USE-CODE** is found.

UCODE opcode — ; < opcode-name>
"u-code"

Used to define new opcodes in the form:

```
opcode UCODE <opcode-name>
```

For example to define the RTX instruction **SELDPR** (for fetches and stores in a data page) use the sequence

```
$B08D UCODE SELDPR
```

USE-DATA <name> —
"use-data"

Used to select which page the compiled data is generated for. Any data following **USE-DATA** will be allocated in the page <name>. This will continue until the compilation is finished or another **USE-DATA** is found.

Target glossary

The words are listed here in ASCII alphabetical order, with the standard pronunciation of the word under the name. The stack comments show the execution point as "—" with the parameters to the left being the input parameters, and those to the right are the results left (if any) by the word's execution. The top of the stack is to the right of the lists. The indicator I indicates that the word is immediate.

Note: Not all the words in this glossary will exist in the Umbilical Forth target.

Words before 'A'

! 16b addr —
"store"
A sixteen bit integer is stored at the given address.

!CSP —
"store-c-s-p"
A word used by compiling and structure words. The stack pointer is saved in user variable **CSP**.

!L 16b addr page —
"store-l"
A 16-bit integer is stored at the long address given in page and address form.

+d1 — +d2
"sharp"
The remainder of +d1 divided by the value of **BASE** is converted to an ASCII character and appended to the output string toward lower memory addresses. +d2 is the quotient and is maintained for further processing. Typically used between <# and #>.

#> 32b — addr +n
"sharp-greater"

Pictured numeric output conversion is ended dropping 32b. 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**.

#LITERAL n1..nn n —
"hash-literal"

Takes n words from the stack and compiles them as literals, n1 first nn last. If no words are to be compiled, n may be zero. This word is used with **NUMBER?** as part of a consistent numeric conversion system.

#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. Typically used between <# and #>.

‘ — compilation-addr
"tick"

Use in the form:

‘ ccccc

Searches the dictionary using the normal search order, returning the compilation address (cfa) of the word. If the word is not found an error message is given.

“” — addr (executing) ; I
"quotes-quotes" — (compiling)

use in the form:

“” <text>"

Compiles text into the dictionary as a string with a count byte, and when the word containing **“”** executes later, the address of the string's count byte is returned.

“, —
”quotes-comma"

Compiles the string following in the input stream into the dictionary as a counted string (count byte + text). Use in the form:

“, string”

The space before the string, and the trailing double quotes are not compiled.

(—
“paren”

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 from zero to the number of characters remaining in the input stream up to the closing parenthesis.

(”) — addr
“paren-quotes”

The run time action of **“”** compiled by **“”**. See **“”**

***** w1 w2 — w3
“times”

W3 is the least-significant 16 bits of the arithmetic product of w1 times w2.

***/** n1 n2 n3 — n4
“times-divide”

n1 is first multiplied by n2 producing an intermediate 32-bit result. n4 is the quotient of the intermediate 32-bit result divided by the divisor n3. The product of n1 times n2 is maintained as an intermediate 32-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 the range {-32,768..32,767}.

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

n1 is first multiplied by n2 producing an intermediate 32-bit result. n4 is the remainder and n5 is the quotient of the intermediate 32-bit result divided by the divisor n3. A 32-bit intermediate product is used as for ***/** . n4 has the same sign as n3 or is zero. An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768..32767}.

+ w1 w2 — w3
"plus"

w3 is the arithmetic sum of w1 plus w2

+! w1 addr —
"plus-store"

w1 is added to the w value at addr using the convention for **+**. This sum replaces the original value at addr.

+LOOP n1 —
"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 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 corresponding **DO**. See: **DO**.

NB: Avoid using **+LOOP** where the difference between the start and end limits is greater than 8000h as this will not produce the results you expect! Use the more efficient **FOR...NEXT** or **BEGIN...WHILE...REPEAT** structures instead.

, 16b —
"comma"

ALLOT space for 16b then store 16b at **HERE 2-**. This is the basic word used to compile 16-bit data into the dictionary. It places the data at the end of the dictionary and adds two to the dictionary pointer. The byte equivalent is called **C,**

,(R) 16b —
"comma-r"

ALLOT space for 16b in the DATA RAM area then store 16b at **THERE 2-**. This is the basic word used to compile 16-bit data into

RAM. It places the data at the end of the portion in use, and adds two to the pointer. The byte equivalent is called **C,(R)**

- w1 w2 — w3
“minus”
w3 is the result of subtracting w2 from w1

—> — I
“arrow”
Continues interpretation/compilation from the next screen. Only valid when interpreting/compiling from a screen file.

-1 — -1
“minus-one”
A constant

-ROT n1 n2 n3 — n3 n1 n2
“dash-rote”
Saves the top of the stack under the next two items. Equivalent to:

```
ROT ROT
```

-TRAILING addr +n1 — addr +n2
“dash-trailing”
The character count +n1 of a text string beginning at addr 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 absolute value of n is displayed in a free field format with a leading minus sign if n is negative.

." — I
“dot-quotes” — (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”

An equivalent of **."** to be used when interpreting, or for immediate display from within compilation, as **."** is intended by the standard to be used within a colon definition to compile a string for later execution. Use in the form:

.(string)

.BYTE n1 —
“dot-byte”

Prints n1 as an unsigned number in a format of two hex digits followed by a space. The current value of **BASE** is unaffected.

.NAME addr —
“dot-name”

Given the name field address of a word, its name is displayed.

.R n1 n2 —
“dot-r”

The number n1 is printed right aligned in a field of width n2 without a trailing space.

.S —
“dot-s”

The contents of the stack are printed out, leaving the contents of the stack unchanged.

.WORD n1 —
“dot-word”

The value of n1 is displayed unsigned as four hex digits and a space. The current value of **BASE** is unaffected.

/ 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 the range {-32,768..32767}.

/MOD n1 n2 — n3 n4
"divide-mod"

n3 is the remainder and n4 the quotient of n1 divided by the divisor n2. n3 has the same sign as n2 or is zero. An error condition results if the divisor is zero or if the quotient falls outside of the range {-32,768..32,767}.

/STRING addr len n — addr+n len-n
"slash-string"

Steps a distance through a string. Often used by text scanning operators, and is then followed by **SKIP** or **SCAN**. See **SKIP SCAN WORD**

0 1 2 — n

The numbers 0..2 occur so often that it is more economical to define them as constants.

0< n — flag
"zero-less"

The flag is true if n is less than zero (negative).

0<> n — flag
"zero-not-equals"

The flag is true if n is non-zero.

0= w — flag
"zero-equals"

flag is true if w is zero.

0> n — flag
"zero-greater"

flag is true if n is greater then zero.

1+ w1 — w2
"one-plus"

w2 is the result of adding one to w1 according to the operation of **+**.

1- w1 — w2
"one-minus"

w2 is the result of subtracting one from w1 according to the operation of **-**.

2! d1 addr —
"two-store"

The double number d1 is stored at addr. Forth stores double precision numbers with the most significant of the two words on the top of the stack. The word **2!** preserves the memory order so that the number configuration in memory is the same as on the stack. See **2@**.

2+ w1 — w2
"two-plus"

w2 is the result of adding two to w1 according to the operation of **+**. This operation is performed so often (like **1+ 1-** and **2-**) that it is worth having fast machine code routines.

2- w1 — w2
"two-minus"

w2 is the result of subtracting two from w1 according to the operation of **-**.

2@ addr — d
"two-fetch"

The double number at addr is returned to the stack. Forth stores double precision numbers with the most significant of the two words on the top of the stack. The double number memory operators preserve the memory order so that the number configuration in memory is the same as on the stack. See **2!**

2* n — 2n
"two-times"

A fast machine code multiply by 2.

2/ n — 2n
"two-slash" or "two-divide"

A fast machine code divide by two. Uses floored division.

2DROP d1 —
"two-drop"

The top two items on the stack are removed.

2DUP d1 — d1 d1
"two-dupe"

The top two items on the stack are duplicated.

2OVER d1 d2 — d1 d2 d1
"two-over"

Copies the second pair of words to the top of the stack.

2SWAP d1 d2 — d2 d1
"two-swap"

The top two pairs of items on the stack are interchanged.

: — sys I
"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 **;** or **;CODE**. sys is balanced by its corresponding **;**.

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

Stops the compilation of a colon definition, allows the <name> of this colon definition to be found in the dictionary, sets interpret state and compiles **;S**. sys is balanced by its corresponding **:**.

;S —
"semi-s"

The word compiled by **;** to perform the return from a high level word. Most often seen as the word at the end of a source screen to stop interpretation. Its action is to leave the word being executed. May be replaced by **EXIT**

< n1 n2 — flag
"less-than"
flag is true if n1 is less than n2

<<8 n — n'
"shift-left-eight"
Shifts the top of stack eight places left, filling with zeros

<<N n 1 n2— n1'
"shift-left-n"
Shifts n1 left n2 places, filling with zeros

<= n1 n2 — flag
"less-than-or-equals"
The flag is true if n1 is less than or equal to n2.

<> n1 n2 — flag
"not-equal"
The flag is true if n1 is not equal to n2.

<# —
"less-sharp"
Initialise pictured numeric output conversion. The words: **# #S HOLD SIGN** can be used to specify the conversion of a double number into ASCII text string stored in right-to-left order. See also **#>**

<MARK — addr
"back-mark"
Marks the entry point of a backward jump, which will later be resolved by **<RESOLVE**.

<RESOLVE addr —
"back-resolve"
Resolves a backward jump whose destination was earlier marked by **<MARK**

= w1 w2 — flag
"equals"
flag is true if w1 is equal to w2

> n1 n2 — flag
"greater-than"
flag is true if n1 is greater than n2

>>8 n — n'
"shift-right-eight"
Shifts the top of stack eight places right, filling with zeros

>>N n 1 n2— n1'
"shift-right-n"
Shifts n1 right n2 places, filling with zeros

>= n1 n2 — flag
"greater-than-or-equal"
The flag is true if n1 is greater than or equal to n2.

>BODY cfa — pfa
"to-body"
Converts a compilation address of a word (in this case, the address of the cfa) to the parameter field address.

>IN — addr
"to-in"
The address of a variable which contains the present character offset within the current input stream buffer. See: **WORD BLK**

>MARK — addr
"forward-mark"
Marks the start of a forward branch which will be later resolved by **RESOLVE>**

>NAME cfa — nfa
"to-name"
Converts a word's compilation address to its name field address.

>R 16b —
"to-r"
Transfers 16b to the return stack. The return stack is a handy place to use for storing data temporarily while other data stack operations take place.

>RESOLVE addr —
“forward-resolve”
Resolves the branch address of a forward branch previously marked by **MARK**

? addr —
“query”
Displays the contents of the address.

?BRANCH flag —
“query-branch”
Consumes a flag and branches to the address given in-line after **?BRANCH** if the flag is true. See **BRANCH**

?COMP —
“query-comp”
Causes the error handler to operate if not compiling

?CSP —
“query-c-s-p”
Causes the error handler to operate if the stack is unbalanced after last **!CSP**

?EVENT —
“query-event”
If the current task’s event flag is set, the flag is reset and the event handler is executed.

?EXEC —
“query-exec”
Causes the error handler to operate if not interpreting

?LOADING —
“query-loading”
Causes the error handler to operate if not loading

?STACK —
“query-stack”
Causes the error handler to operate if the stack is out of limits.

?DNEGATE d1 n — d2
"query-d-negate"

If n is negative, d1 is negated, otherwise it is left alone.

?DO w1 w2 —
"query-do" — sys (compiling)

Used in the forms:

```
?DO ... LOOP
?DO ... +LOOP
```

Begins a loop which terminates based on control parameters. The loop index begins at w2, and terminates based on the limit w1. See **LOOP** and **+LOOP** for details on how the loop is terminated. The loop will not execute if the index and limit are the same. For example the words inside the loop formed by:

```
w DUP ?DO ... LOOP
```

will not be executed. See **DO**

?DUP 16b — 16b 16b
"query-dupe" 0 — 0

Duplicate 16b if it is non-zero. This word is very useful when testing error conditions. Often zero is returned for successful completion, a non-zero value being an error code.

?ERROR flag n —
"query-error"

If flag is true error message n is displayed. When loading, the error message is taken n lines from line 0 of screen 4 in the current error file.

?LEAVE flag —
"query-leave"

If the flag is true (non-zero), the current **DO** ... **LOOP** structure is terminated immediately, execution resumed after the **LOOP** or **+LOOP**.

?NEGATE n1 n2 — n3
"query-negate"

If n2 is negative, n1 is negated to give n3, otherwise n3 is n1.

?PAIRS n1 n2 —
“query-pairs”
An error condition is reported if n1 is not equal to n2.

@ addr — 16b
“fetch”
16b is the value at addr in data space.

@L addr page— 16b
“fetch-l”
A 16-bit word is fetched from a page and address.

Words beginning with ‘A’

ABORT —
“abort”
Clear the stacks and enter the execution state. Return control to the operator via **QUIT**, printing an appropriate message. In this implementation **ABORT** calls **QUIT**. A sealed application will usually replace **QUIT** with a word containing the endless loop that forms the application.

ABORT" — (compiling)I
“abort-quotes” flag — (executing)
Used in the form:

ABORT" string"

If the flag is true the following string is displayed, and **ABORT** then executed.

ABS n — |n|
“abs”
Leave the absolute value of n.

ACTIVATE task# —
“activate”
Initialises and starts the given task number. Task 0 is Forth itself and was activated when Forth started. Note that **ACTIVATE** causes the task to start from the very beginning. If the task was

halted, and execution should resume where it left off, use **RESTART** instead.

AGAIN addr n — (compiling) I
"again" — (execution)

Used in a colon definition in the form:

BEGIN ... AGAIN

At run-time, **AGAIN** forces execution to resume at the corresponding **BEGIN**. There is no effect on the stack. This is an endless loop unless an exit is forced by other means. At compile time, **AGAIN** forces the compilation of **BRANCH**, followed by the address (addr) of the word after the corresponding **BEGIN**. The value n is used for compile time error checking.

ALIGN —
"align"

In cross compiler: Forces the parameter field addresses (PFA) to be on even byte boundaries
In target: Forces **HERE** to an even boundary.

ALLOT n —
"allot"

Reserves n bytes in the dictionary, from the current location. It adds the signed n to the current value of **DP**.

ALLOT-RAM n—
"allot-ram"

Reserves n bytes of RAM from the current RAM pointer given by THERE.

ALSO —
"also"

Room is made for another vocabulary to be added to the start of the vocabulary search list. Space is made by duplicating the top entry. This duplicate entry will be overwritten by the new vocabulary when it is executed. If the order is just:

FORTH ROOT

then after executing **ALSO** it will be:

FORTH FORTH ROOT

and after executing another vocabulary name (say **TOOLS**) it will become:

TOOLS FORTH ROOT

AND w1 w2 — w3
“and”

Leaves the bitwise logical and of w1 and w2 as w3.

ASCII — char ; I
“ascii” — (compiling)

use in the form:

ASCII A

Used to generate the value of the character entered. The example above will return the code for the letter A. If **ASCII** is used in a colon definition the value of the character is compiled as a literal which is returned when the word is executed.

ASSIGN — cfa (executing) ; I
“assign” — (compiling)

Used to assign the action for a deferred word, interrupt, or timer. Used in the form:

ASSIGN action-word TO-DO word

See **TO-DO**

Words beginning with ‘B’

BASE — addr
“base”

The address of a user variable containing the current numeric conversion radix. {{2..72}}

BEGIN — addr n (compiling) ; I
“begin”

Used in the forms:

BEGIN ... flag UNTIL
BEGIN ... AGAIN
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** ... **AGAIN** loop executes forever unless otherwise

left, and a **BEGIN** ... **WHILE** ... **REPEAT** loop will be repeated until flag is false. The words after **UNTIL** or **REPEAT** will be executed when either loop is finished.

BINARY —
“binary”

Switches the current number conversion base to two, by setting user variable **BASE** to two.

BL — char
“b-l”

A constant that returns the ASCII code for a space character.

BLANK addr count —
“blank”

Fills count bytes starting at addr with space characters.

BLK — addr
“b-l-k”

The address of a user variable containing the number of the mass storage block being interpreted as the input stream. If the value of **BLK** is zero the input stream is taken from the text input buffer. {{0..the number of blocks available -1}} See: **TIB**

BLOCK u — addr
“block”

addr is the address of the assigned buffer of the first byte of block u. If the block occupying that buffer is not block u and has been **UPDATE**d it is transferred to mass storage before assigning the buffer. If block u is not already in memory, it is transferred from mass storage into an assigned block buffer. A block may not be assigned to more than one buffer. If u is not an available block number, an error condition exists. Only data within the last buffer referenced by **BLOCK** or **BUFFER** is valid.

BODY> pfa — cfa
“body-to”

Converts a word’s parameter field address to its compilation address (cfa).

BOUNDS addr len — limit start
"bounds"

Converts an address and length into the end-address+1 and start-address, suitable for use with **DO** ... **LOOP**. **BOUNDS** is designed specifically for this purpose.

BRANCH —
"branch"

The Forth goto instruction, normally only used by structure words. Branches to an address given in-line.

BS —
"b-s"

Performs a destructive backspace operation if the variable **OUT** is non-zero. The destructive backspace is performed by the phrase:

```
8 EMIT  SPACE  8 EMIT
```

Words beginning with 'C'

C! 8b addr —
"c-store"

The least-significant 8 bits of 16b are stored into the byte at addr in data space.

C!L 8b addr page —
"c-store-l"

Stores the byte (least significant part of word on stack) into the page and address.

C@ addr — 8b
"c-fetch"

8b is the contents of the byte at addr in data space.

C@L addr page — 8b
"c-fetch-l"

8b is the contents of the byte at the page and address.

C, b —
"c-comma"

Compiles a byte into the next available dictionary location, and advances the dictionary pointer by one. The basic word for compiling byte wide data into the dictionary. See **,**

C/L — n
"c-slash-l"

Returns the number of characters per line. Conventionally 64 even on 80 character terminals. Don't ask why.

CASE — addr n (compiling) ; I
"case"

The word used to mark the start of the **CASE** .... **OF** .... **ENDOF** .... **ENDCASE** structure. For more details see the tutorial section of the PC PowerForth manual on control structures.

CLR-EVENT-RUN —
"clear-event-run"

Clears the event run flag for the current task. This is bit 4 in the task status byte.

CMOVE addr1 addr2 u —
"c-move"

Move u bytes beginning at address addr1 to addr2. The byte at addr2 is moved first, proceeding towards high memory. If u is zero nothing is moved. See **CMOVE> MOVE**

CMOVE> addr1 addr2 u —
"c-move-up"

Move u bytes beginning at address addr1 to addr2. The bytes in high memory are moved first. If u is zero nothing is moved. This word is provided so that blocks of memory can be moved if overlapping. See **CMOVE MOVE**

COLD —
"cold"

Restarts the Forth system as if from scratch. Stacks are reset, the dictionary is cleared out, and **ABORT** is executed.

COMPILE —
“compile”

Typically 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. Most often used to build new compiling or defining words.

CONSTANT 16b —
“constant”

A defining word executed in the form:

16b CONSTANT <name>

Creates a dictionary entry for <name> so that when <name> is later executed, 16b will be left on the stack.

CONTEXT — addr
“context”

A variable array holding the vocabularies which are searched to find a word.

COUNT addr1 — addr2 +n
“count”

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}. Forth strings are often stored as a count byte followed by the text. **COUNT** is used to convert the string address to the address of the text, and the number of characters in that text. For reasons of portability, do not use **COUNT** for any other purpose. For example:

“” Hello" COUNT TYPE

is the same as:

." Hello"

CR —
“c-r”

Displays a carriage-return and line-feed or equivalent operation. The user variable **OUT** is reset to 0.

CRASH —
"crash"

The default action of a deferred word as assigned by **DEFER**. On execution it gives an error message and performs **ABORT**. See **DEFER ABORT**.

CREATE — [parent]
"create" — addr [child]

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. Execution of <name> returns the parameter field address of <name>. **CREATE** is also often used within a colon definition:

```
: cccc  CREATE compile-time words
      DOES> run-time words ;
```

When 'cccc' is executed **CREATE** builds a new dictionary header. **DOES>** is immediate and compiles code that causes the words from the run-time portion of **DOES>** onwards to be executed. The phrase:

```
cccc nnnn
```

causes a new word 'nnnn' to be created. When 'nnnn' executes, **DOES>** returns the address of 'nnnn's parameter area, and the code following **DOES>** is then executed. To illustrate this, we will define **VARIABLE** and **CONSTANT** using **CREATE** and **DOES>**. The action is identical to, but slower than, the usual implementation because the new defining words execute high level code. For example, compiling interactively on the target, the definitions of **CONSTANT** and **VARIABLE** are:

```
: VARIABLE
  CREATE  0 ,  DOES> ;

: CONSTANT
  CREATE  ,  DOES> @  ;
```

CURRENT — addr
"current"

A variable holding the vocabulary into which new definitions are compiled

Words beginning with ‘D’

D>F d — f
“d-to-f”

Converts a 32 bit double integer to a normalized f.p. number.

D+ wd1 wd2 — wd3
“d-plus”

wd1 is the arithmetic sum of wd1 plus wd2.

D- d1 d2 — d3
“d-sub”

The result d3 is d1-d2.

D. d1 —
“d-dot”

Print a signed double number in the current base followed by a space.

D.R d1 n —
“d-dot-r”

Print a signed double number in the current base in a field n characters wide. The output is right aligned with leading zeros suppressed and no trailing space.

DABS d1 — d2
“d-abs”

Take the absolute value of d1 i.e. if d1 is negative, make it positive.

DECIMAL —
“decimal”

Set the input-output numeric conversion base to ten.

DEFER — (parent)
“defer” — (child)

A defining word used in the form:

DEFER <name>

When <name> executes it executes the action assigned to it. The action assigned by **DEFER** is that of **CRASH**, but other actions are assigned by the phrasing:

ASSIGN action TO-DO <name>
e.g:

ASSIGN (EMIT) TO-DO EMIT
will assign the word **(EMIT)** to be the action of **EMIT**, which is a deferred word.

DEFINITIONS —
"definitions"

The vocabulary that words are compiled into (defined by **CURRENT**) is changed to be the same as the first vocabulary in the search order (defined by **CONTEXT**).

DEG>RAD f1 — f2
"deg-to-rad"

Convert f1 degrees to its corresponding number of radians.

DEPTH — n
"depth"

Returns the current stack depth in cells, that is 1 represents one word on the stack, 2 represents two words, and so on. The returned value does not include n, so 0 represents an empty stack.

DIGIT char n1 — n2 true
"digit" char n1 — false

Converts the character char using base n1. If conversion is successful the result is returned with the flag, otherwise only the false flag is returned.

DINT f — d
"dint"

Leave the integer part of f as a double number on the stack.

DLITERAL d — ; I
"d-literal"

Compiles a double number into the dictionary as a literal. Unlike its fig-Forth counterpart, the Forth-83 version is not state-smart. See **LITERAL LIT**.

DNEGATE d1 — d2
"d-minus"

d2 is the two's complement of d1.

DNORM d n — f
"d-norm"

Normalize double number d by n left shifts. Leaves a f.p. number on the stack.

DO w1 w2 —
"do" — sys (compiling)

Used in the forms:

```
DO ... LOOP
DO ... +LOOP
```

Begins a loop which terminates based on control parameters. The loop index begins at w2, and terminates based on the limit w1. See **LOOP** and **+LOOP** for details on how the loop is terminated. The loop is always executed at least once. For example the words inside the loop:

```
w DUP DO ... LOOP
```

are executed 65,536 times. See **?DO**

DOES> — addr
"does" — (compiling)

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

```
: <namex>
  CREATE ... DOES> ... ;
```

and then:

```
<namex> <name>
```

DOES> 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.

DROP 16b —
"drop"

16b is removed from the stack.

DUMP addr n —
“dump”

The n bytes in memory starting at address addr are displayed (in hexadecimal). Very useful when debugging. **DUMP** also shows the ASCII characters formed by the memory.

DUP 16b — 16b 16b
“dupe”

Duplicate 16b.

Words beginning with ‘E’

E. f —
“e-dot”

Print the f.p. number on the stack in exponential form.

ELSE sys — sys I
“else”

Used in the form:

flag IF ... ELSE ... ENDIF

ELSE executes after the true part following **IF** and forces execution to continue at just after **ENDIF**. See: **IF**, **ENDIF** and **THEN**.

EMIT 8b —
“emit”

The ASCII character in the low byte is displayed.

EMPTY-BUFFERS —
“empty-buffers”

All the mass storage buffers are marked as unused, regardless of their current contents or status.

ENDCASE sys — (compiling) ; I
"end-case" n — (executing)

A word used to mark the end of the **CASE** .... **OF** .... **ENDOF** .... **ENDCASE** structure. If entered from the default action a word is dropped from the stack. See **?OF OF ENDOF CASE**

ENDIF sys — (compiling) ; I
"end-if"

The word used to mark the end of the **IF** .... **ENDIF** or **IF** .... **ELSE** .... **ENDIF** structures. See **IF ELSE THEN**

ENDOF sys — sys (compiling) ; I
"end-of"

The word used to mark the end of a selection procedure in the **CASE** .... **OF** .... **ENDOF** .... **ENDCASE** structure. The code between **OF** and **ENDOF** or **?OF** and **ENDOF** is executed if the test at **OF** or **?OF** is passed. After **ENDOF** execution continues immediately after the **ENDCASE**. See **CASE OF ?OF ENDCASE**

ERASE addr n —
"erase"

At address addr, a count of n bytes is zeroed.

ERROR n —
"error"

The standard error handler reports error n. If the system is loading the offending line will be displayed.

EVENT? — t/f
"event-query"

Returns true if the event triggered bit has been set in the current task's status byte.

EXECUTE addr —
"execute"

The word definition indicated by addr is executed. The application will most probably crash if addr is not a compilation address. Useful for executing an action (cfa) pulled out of a table.

EXIT —
"exit"

Compiled within a colon definition such that when executed, that colon definition returns control to the definition that passed control to it by returning control to the return point on top of the return stack. An error condition exists if the top of the return stack does not contain a valid return point, and so **EXIT** will not work within a **DO** ... **LOOP** structure.

EXPECT addr +n —
"expect"

Defined by the standard to receive characters and store each into memory. The transfer begins at addr proceeding towards higher addresses one byte per character until either a <CR> is received or until +n characters have been transferred. No more than +n characters will be stored. The <CR> is not stored in memory.
All characters actually received and stored into memory will be displayed, with <CR> displaying as space. The number of characters collected (excluding any "return") is stored in the user variable **SPAN**. Note that because of this the contents of **SPAN** interrogated directly from the keyboard may not reflect what you intended.

Words beginning with 'F'

F, f —
"f-comma"

Compile the f.p. number on the top of the stack.

F. f —
"f-dot"

Print the top f.p. number on the stack in free format.

F! f addr —
"f-store"

Store the f.p. number f at address addr.

F+ f1 f2 — f3
"f-plus"
Add together the top two f.p. numbers on the stack and put the f.p. result on the stack.

F- f1 f2 — f3
"f-minus"
Subtract the top f.p. number on the stack from the second f.p. number on the stack, and put the f.p. result on the stack.

F* f1 f2 — f3
"f-star"
Take the top two f.p. numbers off the stack, multiply them together, and leave the f.p. result on the stack.

F/ f1 f2 — f3
"f-slash"
Divide the second f.p. number on the stack by the top f.p. number and leave the f.p. result on the stack.

F< f1 f2 — flag
"f-less-than"
Leave true flag if f1<f2. Otherwise, leave a false flag.

F<0 f — flag
"f-less-than-0"
Leave a true flag if f<0. Otherwise, leave a false flag.

F= f1 f2 — flag
"f-equals"
Leave a true flag if the top two f.p. numbers on the stack are equal. Otherwise leave a false flag.

F=0 f — flag
"f-0-equals"
Leave a true flag if the f.p. number on the top of the stack is zero.

F> f1 f2 — flag
"f-greater-than"
Leave a true flag if f1>f2. Otherwise, leave a false flag.

F>0 f — flag
"f-greater-than-zero"

Leave a true flag if the f.p. number on the top of the stack is greater than zero.

F# — f [executing]
"f-hash" — [compiling]

If interpreting, takes text from the input stream and, if possible, converts it to a f.p. number on the stack. Numbers in integer format will be converted to floating point. If compiling, the converted number is compiled.

F#IN — f 3 | 0
"f-hash-in"

Attempts to convert a token from the input stream to a floating point number. Numbers in integer format will be converted to floating point. An indicator (0 or 3) is returned in the same way as an indicator is returned by **FNUMBER?**.

F@ addr — f
"f-fetch"

Fetch the f.p. number from address addr and put it on the stack.

F10^X f1 — f2
"f-10-to-the-x"

Raise 10 to the power f1 and put the result on the stack.

FABS f — |f|
"f-abs"

Take the modulus of the f.p. number on the top of the stack.

FACOS f1 — f2
"f-a-cos"

Leave, on the stack, the angle (in degrees) whose cosine is f1, such that 0<=f2<=180.

FARRAY fn-1..f0 n — [parent]
"f-array" n — fn [child]

When generating the array, take n f.p. numbers and n, and compile them into the array. When executing the child word, take n and place f.p. number n from the array onto the stack. Note that the numbering in the array goes 0,1,..n-1.

FASIN f1 — f2
"f-a-sine"

Leave, on the stack, the angle (in degrees) whose sine is f1, such that -90<=f2<=90.

FATAN f1 — f2
"f-a-tan"

Leave, on the stack, the angle (in degrees) whose tangent is f1, such that -90<f2<90.

FCONSTANT f — [parent]
"f-constant" — f [child]

Floating point equivalent of **CONSTANT**. Use in the form:

<f.p. number on stack> FCONSTANT <name>

FCOS f1 — f2
"f-cos"

Take the cosine of f1 (degrees) and put it on the stack.

FDROP f —
"f-drop"

Drop the f.p. number on the top of the stack.

FDUP f — f f
"f-dup"

Duplicate the f.p. number on the top of the stack.

FE^X f1 — f2
"f-e-to-the-x"

Raise e, the exponential number, to the power f1 and put the result on the stack.

FFRAC f1 f2 — f3
"f-frac"

Leave the fractional remainder from the division f1/f2. The remainder takes the sign of the dividend.

FILL addr u 8b —
"fill"

u bytes of memory beginning at addr are set to 8b. No action is taken if u is zero.

FIND addr1 — addr2 +/-1
"find" addr1 — addr1 0

A counted string is at addr1. It is a name to be looked up in the dictionary. If the name cannot be found addr1 and a false flag are returned, so that **NUMBER?** can later check to see if the string is a valid number. If the name is found, its compilation address (cfa) is returned, together with a non-zero flag. If the word is immediate the flag is 1, otherwise it is -1. The search is through the currently specified search order.

FINT f1 — f2
"fint"

Place the f.p. integer value of f1 on the stack.

FLITERAL f —
"f-literal"

When compiling, compile f as a literal. For example,

```
: ABCD  [ calculate f ] FLITERAL ;
```

Compilation is suspended for the compile-time calculation of f. Execution of **ABCD** leaves f on the stack.

FLN f1 — f2
"f-log-base-e"

Take the logarithm of f1 to base e and put the result on the stack.

FLOATS —
"floats"

Switches the action of **NUMBER?** to be **FNUMBER?**. This action can be reversed by **INTEGERS**. Both **FLOATS** and **INTEGERS** are in the **FORTH** vocabulary.

FLOG f1 — f2
"f-log-base-10"

Take the logarithm of f1 to base 10 and put the result on the stack.

FLUSH —
"flush"

Performs the function of **SAVE-BUFFERS** then unassigns all block buffers. This word was originally intended to be useful for mounting or changing mass storage media but is now used to

ensure that data is passed from Forth to the operating system. The phrase:

```
USING xxx
```

where xxx is an invalid pathname will do this. See **SAVE-BUFFERS EMPTY-BUFFERS**

FMAX f1 f2 — max{f1,f2}
"f-max"

Put the greater of the top two f.p. numbers onto the stack.

FMIN f1 f2 — min{f1,f2}
"f-min"

Put the lesser of the top two f.p. numbers onto the stack.

FNEGATE f — -f
"f-negate"

Negate the f.p. number on the top of the stack.

FNUMBER? addr — 0 | n 1 | d 2 | f 3
"f-number-query"

Converts string at address addr to either a single, double or floating point number (see section 4.2) along with 1, 2, or 3 respectively. If a 0 is left on the stack then **FNUMBER?** was unable to convert the string.

FOR — sys (compiling) I
"for" n — (executing)

Marks the start of the **FOR ... NEXT** structure peculiar to the RTX family of processors. The structure is used in the form:

```
n FOR <words> NEXT
```

The words between **FOR** and **NEXT** are executed n+1 times. The value n is transferred to the top of the return stack, and is decremented on each iteration. If the value is 0 on ENTRY to **NEXT**, the loop terminates and the return stack is popped. For example a simple version of **TYPE** might be:

```
: TYPE          \ addr len —
  1-            \ loop executes n+1 times
  for
    dup c@ emit \ display character
    1+                      \ next address
  next                      \ until done
```

```
  drop                          \ clean up
;
```

FORGET —
“forget”

Used in the form:

FORGET <name>

If <name> is found in the compilation vocabulary (defined by **CURRENT**), delete <name> from the dictionary, and also delete 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 —
“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**

FOVER f1 f2 — f1 f2 f1
“f-over”

Floating point equivalent of **OVER**.

FROT f1 f2 f3 — f2 f3 f1
“f-rote”

Floating point equivalent of **ROT**.

FSEPARATE f1 f2 — f3 f4
“f-separate”

Leave the signed integer quotient f4 and remainder f3 when f1 is divided by f2. The remainder has the same sign as the dividend.

FSIGN f — f flag
“f-sign”

Leave the f.p. number and a flag on the stack. Leaves a true flag if f is negative, else leaves a false flag.

FSIN f1 — f2
"f-sine"
Leave the floating point sine of f1 (degrees) and put it on the stack.

FSQR f1 — f2
"f-s-q-r"
Take the square root of the floating point number on the top of the stack and put the result onto the stack.

FSWAP f1 f2 — f2 f1
"f-swap"
Floating point equivalent of **SWAP**.

FTAN f1 — f2
"f-tan"
Take the tangent of f1 (degrees) and put the result on the stack.

FVARIABLE —
"f-variable"
Floating point equivalent of **VARIABLE**. Set up an **FVARIABLE** by typing:

FVARIABLE <name>

FX^N f1 n — f2
"f-x-to-the-n"
Raise f1 to the power n (n integer), and put result on the stack.

FX^Y f1 f2 — f3
"f-x-to-the-y"
Raise f1 to the power f2 and put the result on the stack.

Words beginning with 'G'

GET-MESSAGE — message task#
"get-message"
Waits for a message to be received and returns the message and the sending task.

Words beginning with ‘H’

HALT task# —
“halt”

Halts the task whose number is given. Do not halt task 0. Halting a task prevents it responding to messages or events.

HALT? — flag
“halt-query”

Tests the keyboard using **KEY?** to see if a key has been pressed. If no key has been pressed, a zero flag is returned. If a key has been pressed it is read. If the key is not a space, a true flag is returned. If the key is a space, another key is read. If the second key is a space, a false flag is returned, otherwise a true flag is returned.
This word is very useful to control output displays, as it pauses on the space bar, and any other key returns a true flag, usually used to terminate the display.

HERE — addr
“here”

The address of the next available dictionary location.

HEX —
“hex”

Changes the base for numeric conversion to hexadecimal. The contents of **BASE** will be changed to decimal 16.

HOLD char —
“hold”

Char is inserted into a pictured numeric output string. Typically used between <# and #> to embed a character into numeric output.

Words beginning with 'I'

I — w
"i"

w is a copy of the loop index. Unlike older fig-Forth implementations, in Forth-83 **I** is not a synonym of **R@** which should not be used. May only be used in the forms:

```
DO ... I ... LOOP
DO ... I ... n +LOOP
```

IF flag — (executing)
"if" — sys (compiling)

Used in the forms:

```
flag IF ... ELSE ... ENDIF
flag IF ... ENDIF
```

If flag is true, the words following **IF** are executed and the words following **ELSE** until just after the **ENDIF** are skipped. The **ELSE** part is optional. If flag is false, words from **IF** through **ELSE**, or from **IF** through **ENDIF** (when no **ELSE** is used), are skipped. See **ELSE ENDIF THEN**.

IMMEDIATE —
"immediate"

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

INIT-MULTI —
"init-multi"

Initialises the multi-tasker, task 0 which is the Forth itself, and starts the multi-tasker. Just include this word in **COLD** to kick the multi-tasker into action.

INIT-TCBS —
"init-t-c-bees"

The main part of the multi-tasker reset process.

INTEGERS —
"integers"

Switches the action of **NUMBER?** to be **INTEGER?**. This action reverses that of **FLOATS**. Both **FLOATS** and **INTEGERS** are in the **FORTH** vocabulary.

INTERPRET —
"interpret"

The outer text interpreter which interprets or compiles each word from the input stream according to the state of the variable **STATE**. If the word is not in the dictionary, a number conversion is attempted. If this fails, an error is reported. Text input is performed by **WORD** and numeric conversion is performed by **NUMBER?**.

Words beginning with 'J'

J — w
"j"

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

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

Words beginning with 'K'

KEY — char
"key"

Receives a character from the console/terminal or input stream. All valid characters can be received. According to the Forth 83 standard, control characters should not be processed by **KEY** or the host system for any editing purpose. Characters received by **KEY** will not be displayed.

KEY? — flag
"key-query"

Returns a true flag if a character is available for input by **KEY**.

Words beginning with 'L'

LATEST — addr
"latest"

Returns the address of the most recently defined word in the **CURRENT** vocabulary (the one words are being compiled into).

LEAVE — ; I
"leave"

When **LEAVE** is encountered the loop terminates immediately, and execution resumes after **LOOP** or **+LOOP**. When the loop terminates the loop control parameters are discarded. May only be used in the forms:

```
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** structure.

LINK>N lfa — cfa
"link-to"

Converts a word's link field address to its compilation address (cfa).

LIT — n
"lit"

The primitive compiled by **LITERAL** to return an in-line value. When a number such as 33 is encountered while compiling, it is compiled as **LIT** 33 into the dictionary, and is returned by **LIT** when the word executes.

LITERAL 16b — (compiling)I
"literal"

Typically used in the form:

```
[ 16b ] LITERAL
```

Compiles a system dependent operation so that when later executed, 16b will be left on the stack. Unlike its fig-Forth counterpart, this word is not state-smart. See **LIT**

LOOP sys — (compiling)I
"loop" — (executing)

Increments the DO-LOOP index by one. If the new index crosses the boundary between limit-1 and limit, the loop is terminated and loop control parameters are discarded. When the loop is not terminated, execution continues to just after the corresponding **DO**.

Words beginning with 'M'

M* n1 n2 — d
"m-star"

Two sixteen bit signed numbers are multiplied together to produce a 32 bit signed number.

M/MOD d1 n2 — n3 n4
"m-slash-mod"

A signed mixed magnitude operator. NOT the same as the fig-Forth word of the same name, which is replaced by **MU/MOD**. The 32 bit d1 is divided by the 16 bit n2 to produce a 16 bit remainder n3 and a 16 bit quotient n4. Note the use of floored division.

MAX n1 n2 — n3
"max"

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

MIN n1 n2 — n3
"min"

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

MOD n1 n2 — n3
"mod"

n3 is the remainder after dividing n1 by the divisor n2. The sign of n3 is determined by the rules of floored division.

MOVE addr1 addr2 count —
"move"

An intelligent version of **CMOVE** that copies count bytes starting at addr1 to addr2, such that the destination block is always an image of the source block. Useful when the ranges may overlap.

MSG? task# — t/f
"message-query"

Returns true if the task is holding a message, and is therefore not free to receive another one.

MU/MOD ud1 u2 — u3 ud4
"m-u-slash-mod"

An unsigned mixed magnitude operator. Double number ud1 is divided by u2 to give a remainder u3 and a double quotient ud4. This word is only necessary as the Forth-83 **UM/MOD** does not return a double quotient as did its fig-Forth forbear. Be careful not to confuse **UM/MOD** (part of the Forth-83 standard) with **MU/MOD** (not part of the standard, introduced by F83).

MULTI —
"multi"

Turns the multi-tasker on, by clearing the bit in the TASK# byte in RAM that inhibits the scheduler.

Words beginning with 'N'

N>LINK cfa — lfa
"to-link"

Converts the compilation address of a word (in this case the cfa) to the address of its name field.

NAME> nfa — cfa
"name-to"

Converts a word's name field address to its compilation address (cfa).

NEGATE n1 — n2
"minus"

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

NEXT sys — (compiling)
"next" — (executing)

Marks the end of the **FOR ... NEXT** control structure peculiar to the RTX family of processors. For details see **FOR**.

NIP n1 n2 — n2
"nip"

Removes the second item on the stack. Used for cleaning up.

NOOP —
"no-op"

A dummy word that does nothing.

NOT 16b1 — 16b2
"not"

16b2 is the one's complement of 16b1.

NUMBER? addr — n1..nn n
"number-query"

Performs the function of converting text to binary numbers. The counted string at addr is converted to a number. If conversion is possible the number of words generated is left on the top of the stack as well as a number of that size.

No conversion	— 0
Single number	— n 1
Double number	— d 2
Soft floating point	— f 3

If a comma is encountered, the variable **DPL** will contain the number of digits after the comma, otherwise **DPL** contains -1. See **#LITERAL**

Words beginning with 'O'

OF sys — (compiling)I
"of" n1 n2 — (executing & n1=n2)
n1 n2 — n2 (executing & n1n2)

Used to mark the start of a section of code conditionally executed in a **CASE** ... **OF** ... **ENDOF** ... **ENDCASE** control structure. If n1 is equal to n2 the code between **OF** and **ENDOF** is executed, and control then passes to immediately after **ENDCASE**. Otherwise control passes to immediately after the next **ENDOF**, n1 being kept so that another test can be made in front of another **OF** ... **ENDOF** clause.

OFF addr —
"off"

Clears (zeros) the word at the given address. Used for resetting flags, and clearing counters. See **ON**

ON addr —
"on"

Sets the word at addr to -1. Used for setting flags.

ONLY —
"only"

Reduces the search order to be just the **ROOT** vocabulary, which is a short vocabulary from which all others can be reached.

OR 16b1 16b2 — 16b3
"or"

16b3 is the bit-by-bit inclusive-or of 16b1 with 16b2.

ORDER —
"order"

Displays the order in which vocabularies are searched, starting with the first one searched (the **CONTEXT** vocabulary). The vocabulary into which definitions are built (the **CURRENT** vocabulary) is also displayed.

OVER 16b1 16b2 — 16b1 16b2 16b1
"over"

Copies the second item on the stack to the top of the stack. Like **DUP** this word is useful for getting a copy of a stack item for passing as a parameter to another word. See **DUP**

Words beginning with 'P'

PAD — addr
"pad"

The base address of a scratch area used to hold text and string 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.

PAUSE
"pause"

Waits for one iteration of the scheduler. Equivalent to:

```
1 WAIT
```

PICK +n — 16b
"pick"

16b is a copy of the +nth stack value, not counting +n itself, where 0 refers to the top of the stack.

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

PLACE addr1 len addr2 —
"place"

Copies an uncounted string addr1/len to a counted string at addr2

PREVIOUS —
"previous"

Reduces the vocabulary search order by deleting the first entry in the list. Used with **ALSO** to temporarily add a vocabulary to the search list:

```
ALSO TOOLS ..... PREVIOUS
```

Words beginning with ‘Q’

QUERY —
“query”

Input 80 characters of text (or until a carriage-return) from the user’s terminal. The text is placed at the address contained in **TIB** and the variable **>IN** (position in input line) is set to zero. See **EXPECT**

QUIT —
“quit”

Clears the return stack, sets interpret state, accepts new input from the current input data device, and begins text interpretation. No message is displayed.

Words beginning with ‘R’

R@ — 16b
“r-fetch”

16b is a copy of the top of the return stack. The return stack is unaffected.

R> — 16b
“r-from” or “from-r”

16b is removed from the return stack and transferred to the data stack.

RAD>DEG f1 — f2
“rad-to-deg”

Convert f1 radians to degrees, and put result on the stack.

RECURSE — ; I
“recurse”

Compiles the compilation address (cfa) of a word inside the definition of a word. Normally a word name is not available until its definition is complete, so that a word can be redefined in terms of its previous definition. If recursion is required, this mechanism must be overcome, and that function is performed by **RECURSE**.

REPEAT sys — (compiling)
"repeat" — (executing)

Used in the form:

BEGIN ... flag WHILE ... REPEAT

At execution time, **REPEAT** continues execution to just after the corresponding **BEGIN**. See: **BEGIN WHILE**

RESET-BIT mask addr —
"reset-bit"

A bit masking operation performed on the byte at addr. All the '1' bits in the mask are reset in the byte. Logically, the equivalent of **NOT AND**.

RESTART task# —
"restart"

Restarts a task that was halted by **HALT** or **WAIT**. Unlike **ACTIVATE**, the task resumes where it left off.

RESTORE-INT sr md cr —
"restore-int"

Restore the interrupt enable state previously saved by **SAVE-INT**.

ROLL +n —
"roll"

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. If n is negative no action is taken. n=0 refers to the top of the stack. Note also that this is a slow operation as data is actually copied.
Frequent use of **ROLL** is often a sign of bad factorisation of the problem into separate words.

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"

The return stack pointer is set to the given value. The stack can be reset to its original value by the phrase:

```
R0 @ RP!
```

See also **RP@ R0 SP@ SP! S0**

RP@ — addr
"r-p-fetch"

Returns the current value of the return stack pointer. See also **R0 RP! S0 SP! SP@**

Words beginning with 'S'

S0 — addr
"s-nought"

A user variable holding the address which should be used to reset the data stack.

```
S0 @ SP!        \ reset data stack
```

The initialisation of each task that may reset the data stack should include code to initialise this variable, e.g.

```
SP@ S0 !        \ initialise
```

S>D n — d
"s-to-d"

The signed 16 bit number n is converted to a signed 32 bit number d.

S>F n — f
"s-to-f"

Converts a single (16 bit) number to a normalized f.p. number.

S= addr1 addr2 length — flag
"s-equals"

The two strings at addresses addr1 and addr2, length bytes long, are compared, and if they are identical a true flag is returned.

SAVE-BUFFERS —
"save-buffers"

The contents of all block buffers marked as **UPDATE**d are written to their corresponding mass storage blocks. All buffers are marked as no longer being modified, but remain assigned.

SAVE-INT — sr md cr
"save-int"

Saves the current state of the interrupt enable on the stack, and disables interrupts. See **RESTORE-INT**.

SCAN addr len char — addr' len'
"scan"

A text scanning primitive. Given the address of some ASCII text, the number of bytes to go, and the character to look for, the text is scanned for the given character. The address at which the character was found, and the number of bytes remaining is returned. If the number of bytes remaining is 0, the character was not found. See **SKIP**

SELF — task#
"self"

Returns the task number of the current task. Useful with **MSG?** in particular to determine whether or not a message has been received by the task.

SEND-MESSAGE message task# —
"send-message"

Sends a message to the given task. The message can be used on its own, or as a pointer to an extended message.

SET-BIT mask addr —
"set-bit"

A byte-wide bit masking operation. All '1' bits in the mask are set in the byte at addr. Logically equivalent to **OR**. See **RESET-BIT TEST-BIT TOGGLE-BIT**

SIGN n —
"sign"

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

SINGLE —
"single"

Turns off the multi-tasker by setting the scheduler disable bit in the **TASK#** byte in internal RAM.

SINT f — n
"sint"

Takes the single number integer part of f and puts it on the stack.

SKIP addr len char — addr' len'
"skip"

As **SCAN**, but **SKIP** looks for the first character that is NOT the specified character. See **SCAN**

SMUDGE —
"smudge"

Toggles the 'smudge' bit of the most recently defined words name field. If the bit is set the word cannot be found by a normal dictionary search. If an error occurs during compilation the phrase:

SMUDGE FORGET <name>
can be used to remove from the dictionary the word in which the error occurred.

SOURCE — addr len
"source"

Returns the address and length of the current input buffer. This will be the text input buffer or the disc buffer, depending on the value of **BLK**.

SP! addr —
"s-p-store"

Sets the parameter stack pointer to the given value.

SP@ — addr
"s-p-fetch"

Returns the current value of the parameter stack pointer.

SPACE —
"space"

Display an ASCII space.

SPACES n —
“spaces”

Display +n ASCII spaces. Nothing is displayed if n is zero or negative.

STATE — addr
“state”

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. Usually only used by ‘state-smart’ words (e.g. ASCII) in application programs to determine whether Forth is compiling or executing.

STATUS — n
“status”

Returns the task status byte of the current task but with the top bit (bit 7) masked off. If this value is non-zero, the task has been awakened for a reason other than for normal running.

SWAP 16b1 16b2 — 16b2 16b1
“swap”

The top two stack entries are exchanged.

Words beginning with ‘T’

TCBS — addr
“t-c-b-st”

A label, NOT a word, that returns the start address in DATA RAM of the table holding the action words for all the tasks. In some systems this is implemented as a constant for visibility.

TEST-BIT mask addr — b
“test-bit”

A byte-wide bit-masking operation. b is the result of testing all the bits at addr that are ‘1’ bits in the mask. Logically equivalent to **AND**.

THEN sys — (compiling)
"then"

Used in the forms:

```
flag IF ... ELSE ... THEN
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 ENDIF**

THRU n1 n2 —
"thru"

Screens n1 to n2 inclusive are loaded.

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.

TO-DO — (compiling)I
"to-do" cfa — (executing)

Sets the action of the deferred word by writing the cfa into the parameter field (body) of a word created by **DEFER**. Used in the form:

```
DEFER word
ASSIGN action-word TO-DO word
```

TO-EVENT cfa task# —
"to-event"

Sets the CFA of a Forth word as the action to run when the task's event trigger is set.

```
ASSIGN <word> <n> TO-EVENT
```

TO-TASK cfa task# —
"to-task"

Stores the CFA of the word forming the task action in the task table entry for the task.

```
ASSIGN <word> <n> TO-TASK
```

TOGGLE-BIT mask addr —
“toggle-bit”

A byte-wide bit-masking operation. All the ‘1’ bits in the mask are inverted at addr. Logically equivalent to XOR. See **TEST-BIT SET-BIT RESET-BIT TOGGLE**

TUCK n1 n2 — n2 n1 n2
“tuck”

Saves a copy of the top item on the stack under the second item.

TYPE addr n —
“type”

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

Words beginning with ‘U’

U. u —
“u-dot”

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

U< u1 u2 — flag
“u-less-than”

The flag is true if u1 is logically less than u2.

U> u1 u2 — flag
“u-greater-than”

The flag is true if u1 is logically greater than u2.

UM* u1 u2 — ud
“u-star”

Two unsigned 16 bit numbers are multiplied together to produce an unsigned 32 bit number.

UM/MOD ud u1 — u2 u3
"u-m-slash-mod"

The 32 bit unsigned number ud is divided by the unsigned 16 bit number u1 to produce 16 bit unsigned numbers. The remainder is u2 and the quotient is u3.

UNTIL sys — (compiling) I
"until" flag — (executing)

Used in the form:

BEGIN ... flag UNTIL

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

UPC char1 — char2
"u-p-c"

If char1 is a lower case letter, it is converted to upper case. See **UPPER**

UPDATE —
"update"

The currently valid block buffer is marked as modified. Blocks marked as modified will subsequently be automatically transferred to mass storage should its memory buffer be needed for storage of a different block or upon execution of **FLUSH** or **SAVE-BUFFERS**.

UPPER addr len —
"upper"

The string of len bytes starting at addr is converted to upper case using **UPC**. See **UPC**.

USER n —
"user"

A defining word used in the form:

n USER cccc

which defines a user variable whose address is n bytes from the start of the user area. When the user variable cccc is executed the address of its data area is returned. User variables usually contain system information which will be affected by a multi-user or

multi-tasking environment. The base address of the user area is held in the UP register.

Words beginning with 'V'

V-FIND addr1 addr2 — cfa +/-1
"paren-find" addr1 addr2 — addr1 0

A vocabulary defined by addr2 is searched. Each word name is tested against the string at addr1. If a match is found, the cfa, and a flag are returned. The flag is 1 for an immediate word, and -1 for a normal word. If no match is found, the string address and 0 are returned. The vocabulary address addr2 is the same address as is set into **CONTEXT** by executing it.

VARIABLE —
"variable" — addr [child]

A defining word executed in the form:

VARIABLE <name>

A dictionary entry for <name> is created and two bytes are **ALLOT**ted in its parameter field. This parameter field is to be used for the contents of the variable. The contents are initialised to zero. When <name> is later executed, the address of its parameter field is placed on the stack.

VOCABULARY —
"vocabulary" — [child]

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 **CONTEXT** vocabulary with <name>. When <name> becomes the compilation vocabulary new definitions will be appended to <name>'s list. See: **DEFINITIONS CONTEXT CURRENT V-FIND**

VOCS —
"vocs"

Displays a list of all the vocabularies in the dictionary.

Words beginning with 'W'

WAIT n —
"wait"

Suspends the current task for n iterations of the scheduler. If n is 0, the task is suspended until a message or event is received.

WAIT-EVENT/MSG —
"wait-event-or-message"

The current task is suspended until it receives a message or an event trigger. The words **MSG?** and **EVENT?** can be used to determine whether a message or an event trigger terminated the wait. Note that if an event trigger is received, the event handler will have been called, and the event run flag (bit 4 in the status byte) will be set.

WHILE sys — sys (compiling)
"while" flag — (executing)

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. See: **BEGIN REPEAT**

WITHIN? n1 n2 n3 — flag
"within"

The flag is returned true if n1 is in the range n2..n3 inclusive.

WORD char — addr
"word"

Generates a counted string by non-destructively accepting characters from the input stream until the delimiting character char

is encountered or the input stream exhausted. Leading delimiters are ignored.
The entire character string is stored in memory beginning at **'WORD** 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. The address returned is the address at which the string was placed.

WORDS —
"words"

Lists the names of the words in the **CONTEXT** vocabulary. Pressing the space bar will halt the listing, which can be restarted by pressing the space bar again. Any other key will cause the listing to abort.

Words beginning with 'X'

XOR 16b1 16b2 — 16b3
"x-or"

16b3 is the bit-by-bit exclusive-or of 16b1 with 16b2.

Words beginning with 'Y'

None

Words beginning with 'Z'

None

Words after ‘Z’

[— ; I
“left-bracket” — (compiling)

Switches to the interpretation state. The text from the input stream is subsequently interpreted. For typical use see **LITERAL**. The use of **[** and **]** must be balanced.

[COMPILE] — (compiling) ; I
“bracket-compile”

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.

\ — I
“back-slash”

Defines a comment to the end of the input line. This word can be used with any input source.

] —
“right-bracket”

Sets compilation state. The text from the input stream is subsequently compiled. For typical usage see **LITERAL**. The use of **[** and **]** must be balanced. See: **[**

Blank Page

Word list

A

B

C

D

E

F

G

H

P

Q

R

S

T

U

V

W

X