The Language of Innovation...

Forth Programmer's Handbook

Edward K. Conklin
Elizabeth D. Rather
and the technical staff of FORTH, Inc.

FORTH, Inc.
111 N. Sepulveda Boulevard, Suite 300
Manhattan Beach, California USA 90266–6847
310.372.8493 800.55.FORTH FAX: 310.318.7130
forthinc@forth.com www.forth.com

First edition, September 1997
Second printing, November 1997
Third printing (2nd ed.), August 1998
Fourth printing, September 1999
Fifth printing, May 2000
Sixth printing, August 2000

ISBN 0-9662156-0-5

FORTH, Inc.
111 N. Sepulveda Boulevard, Suite 300
Manhattan Beach, California USA 90266–6847
310.372.8493 800.55.FORTH FAX: 310.318.7130
sales@forth.com www.forth.com

CONTENTS

List of Figures

List of Tables

Welcome!

About the Forth Programming Language

The Forth programming language was originally developed in the early 1970s by Charles H. Moore, at the National Radio Astronomy Observatory. Forth was used at several NRAO installations for controlling radio telescopes and associated scientific instruments, as well as for high-speed data acquisition and graphical analysis. Today Forth is used worldwide by people seeking maximum flexibility and efficiency in a wide variety of application areas.

About This Book

The *Forth Programmer's Handbook* book provides a detailed technical reference for programmers and engineers who are developing software using versions of Standard Forth *(ANSI X3.215:1994*, the standard adopted in 1994*;* equivalent to *ISO/IEC 15145:1997*) provided by FORTH, Inc. or other vendors. It features Standard Forth and many extensions commonly in use; some information in this book is taken directly from the official standard's documentation.

This book assumes the reader has a general knowledge of programming principles and practices, and general familiarity with computer hardware and software systems.

How to Use This Book

Each section of this book documents a single subject, and many are followed by a glossary containing pertinent Forth words and their descriptions. Each

Forth word is shown with its stack effects and with the Standard Forth word list in which it appears, if any. Some words are included which are not part of Standard Forth; these are indicated by the phrase "common usage." Sections in this book often conclude with references to related topics or other resources.

Appendix C provides an index of each Forth word that appears in these glossaries, including its stack effect, the page on which its description may be found, and the Standard Forth word list, if any, in which it appears.

Reference Materials

The following reference materials may be of use to the reader of this manual.

- *Starting Forth* (introductory tutorial).
- *American National Standard for Information Systems Programming Languages — Forth (ANSI X3.215:1994)*
- *ISO/IEC 15145:1997 Information technology — Programming languages — Forth* (the content of this standard is identical to ANSI X3.215:1994)

Additional recommended publications are listed in Appendix A, "Bibliography" on page 201, along with other sources of information about Forth.

How to Proceed

If you are not already familiar with Forth, we encourage you to begin by reading the "Introduction" and "Forth Fundamentals" chapters carefully, writing simple programs using an ANS Forth system of your choice. Use this book for technical details about your standard-compliant system and to assist you as you move on to more ambitious programming challenges.

Good luck!

1. INTRODUCTION

This *Forth Programmer's Handbook* provides a reference source for the most common features of the integrated software development systems based on the Forth programming language. We assume at least an elementary knowledge of programming, including any high-level language or assembler. If you are new to Forth, we encourage you to begin by reading this chapter and the next carefully, writing simple programs using an ANS Forth system of your choice.

This book is primarily intended to describe how a programmer can use Forth to solve problems. This is a rather different goal from explaining how Forth works, but it is a practical necessity for the new user of a Forth system. This manual is also organized to serve experienced programmers who need to check some point quickly.

We highly recommend that you spend time examining the Forth source code supplied with your system, along with its documentation. Forth was designed to be highly readable, and the source code offers many examples of good usage and programming practice.

This manual does not attempt to cover all Forth commands. Indeed, no book can do that—Forth is an extensible system, and no two implementations need or use identical components. What we can do is provide a detailed exposition of the most valuable and most commonly used features and facilities of the fundamental system from which your application begins.

FORTH, Inc. provides development environments for a growing number of computer systems and embedded microprocessors. Since hardware is unique for each computer, it is not feasible for this document to cover every feature of every system supported. The *Forth Programmer's Handbook* presents features common to Standard Forth and to the most common extensions found in all FORTH, Inc. systems. When discussing hardware-specific features, particu-

larly dictionary structure, high-level object format, database management, and device drivers, an idealized model of a Forth system is used. Separate product documentation provides implementation details and descriptions of features specific to that system.

In this manual, typefaces are used as follows:

- This typeface is used for text, with *italic* used for symbolic notation and for the first appearance of *new terms*;
- Executable Forth commands and source code are shown in distinctive bold type, e.g., **`60 LIST`**.
- Parameters that are described indirectly instead of explicitly are shown in distinctive plain type and inside brackets, e.g., `<block number>` **`LIST`**. When these parameters are discussed in text, they usually are shown in *italic*.
- Non-executable text strings such as error messages are shown in plain type without brackets, e.g., `Page Fault`.

1.1 Forth Language Features

This section highlights special considerations arising from the actual implementation of a system. More detailed technical discussions of subjects covered here will be found in later sections of this book, especially Section 2. Appendix B, "Glossary & Notation" provides supplementary definitions of many of the terms used in this manual, as well as a detailed description of the notation conventions.

1.1.1 Definitions of Terms

Forth allows any kind of ASCII string (except one containing spaces) to be a valid name, and this introduces some ambiguities in references. For instance, Forth calls subroutines *words*, but *word* could also mean an addressable unit of memory. To resolve this, we use the following conventions:

- A Forth execution procedure is called a *definition*. A *word* is the name of such a definition.

- The word length of the processor is always referred to as a *cell*. This is also the size of an address and the size of a single item on Forth's stacks.
- Eight bits is called a *byte*. On a 32-bit or larger processor, a 16-bit item may be called a *16-bit cell* or *half-cell*.

1.1.2 Dictionary

The dictionary contains all the executable routines (or *words*) that make up a Forth system. *System routines* are entries predefined in the dictionary that become available when the system is booted. *Electives* are optionally compiled after booting. *User-defined words* are entries the user adds. In a multi-user configuration, system and elective definitions are available to all users, whereas user-defined words are available only to the user who defines them. Otherwise, there are no differences in size, speed, or structure. You may make user words available to other users simply by loading them with the other electives.

The basic form of the most common type of word definition is:

```
: <name>  <words to be executed> ;
```

where **:** constructs a new definition called *name*, which is terminated by **;**. When *name* is referenced, the words in the body of the definition name will be executed. There are other kinds of words in Forth: words defined in assembler code, words that function as data objects, etc. All have dictionary entries with a similar structure, and are managed by the same internal rules. The various kinds of definitions are discussed in Section 4.2.

The dictionary is the fundamental mechanism by which Forth allocates memory and performs *symbol table* operations. Because the dictionary serves so many purposes, it's important that you understand how to use it.

The dictionary is a linked list of variable-length entries, each of which is a Forth word and its definition. In most implementations, the dictionary grows toward high memory; the discussion in this section will assume it does. Each dictionary entry points to the entry that logically precedes it (see Figure 1). The address of the next available cell at the end of the dictionary is put on the stack by the word **HERE**.

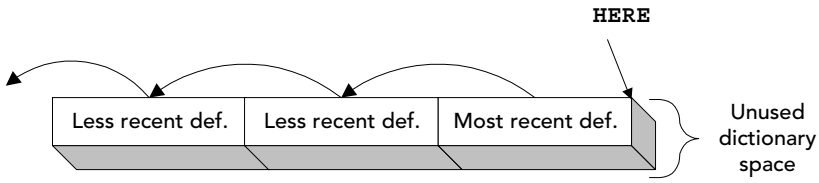

Figure 1. The "top" of a dictionary. `HERE` returns the address of the next available location.

Dictionary entries are not necessarily contiguous. For example, in cross-compilers used to construct programs for embedded systems, the searchable portion of the dictionary (name, link, and a pointer to the content—see Figure 2) may reside in a host computer, and the actual content may reside in a target image being constructed in the host computer's memory for later downloading or for burning into PROM.

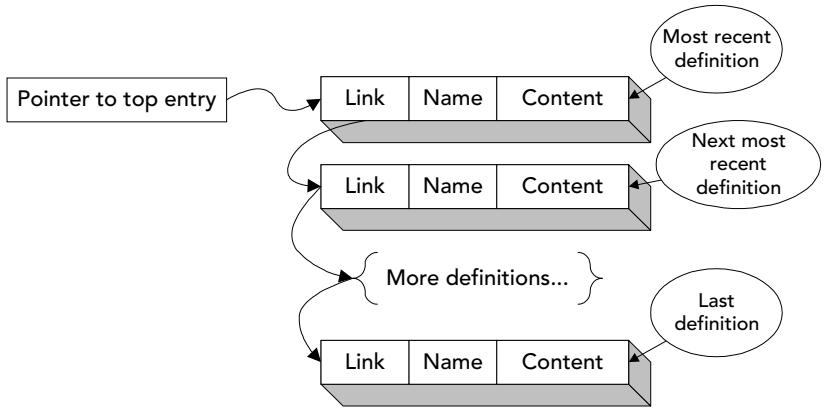

Figure 2. Logical structure of the Forth dictionary

The dictionary is searched by sequentially matching names in source text against names compiled in the dictionary. On some systems, the search is speeded by providing more than one chain of definitions, with the entries linked in logical sequences that do not necessarily reflect their physical location. The Forth *text interpreter* selects one of these chains to search; the selection mechanism is implementation dependent, and may include two or more chains in a programmer-controlled order (see Section 4.6). The search follows

the selected chain until a match is found or the end of the chain is reached. Because the latest definition will be found first, this organization permits words to be redefined, a technique that is frequently useful.

The Standard Forth term for one of these chains is *word list*. A word list is a subset of the dictionary containing words for some special purpose. There usually are several word lists present in a system and these are normally available to all users on a re-entrant basis.

The essential structure of dictionary entries is the same for all words, and is diagrammed in Figure 2. The *link cell* contains the location of the preceding entry. This speeds up searches, which start at the recent end of the dictionary and work backwards to the older end. By this process, the most recent definition of a word is always found. In a developed application, where the user is dealing with the highest level of the program, this process optimizes search time.

The *name field* in a dictionary entry contains the count of characters in the full name, followed by some number of characters in the name. The count (and, thus, the longest allowable name length) usually is limited to 31 characters. On most systems, any characters other than space, backspace, and carriage return can be used as part of a name field. However, Standard Forth advises that you can only depend on being able to use graphic characters.

Some systems are case sensitive and others are not; see your product documentation for details. To avoid problems and to maximize the transportability of code, the names of the words provided in a standard system are defined in all upper-case letters and should always be referred to in all upper-case letters when using them in subsequent definitions. When defining and using new names, it is important to be consistent; always refer to a name using exactly the same case(s) in which it was defined. Also, in systems that are case sensitive, avoid creating names that differ *only* in their use of case; such code will not be transportable to a case-insensitive system.

Although the order of the fields in a dictionary entry is arranged in each implementation to optimize each machine's dictionary search, Figure 3 shows a general model. There will always be a *link field* and a *name field*, and usually a *code field*. The code field directs the system to the run-time code to be executed when this definition is invoked. There is often a *parameter field* of variable length, containing references to data needed when this definition executes. There may also be a *locate field*, containing information about where

this word is defined in source code. When developing programs for embedded systems, this structure may exist only on the host, with a parameter field containing a pointer to the actual executable portion being constructed in the target image.

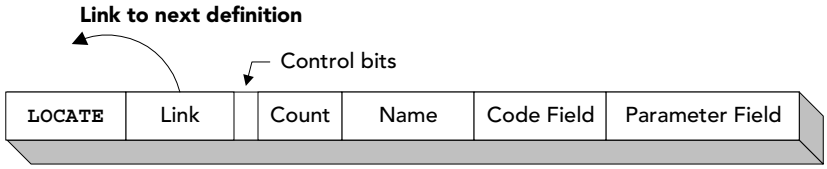

Figure 3. Structural details of a typical dictionary entry

In addition, usually there are several *control bits* to control the type and use of the definition. Since the longest name field in most systems has 31 characters, requiring only five bits to express a count, the control bits are often found in the byte containing the count. The most important control bit is called the *precedence bit*. A word whose precedence bit is set executes at compile time. The precedence bit is set by the word **IMMEDIATE**. The precedence bit is used for a few special words, such as compiler directives, but it is zero for most words.

Another common control bit is the *smudge bit*. A word whose smudge bit is set is invisible to a dictionary search. This bit is set by the compiler when starting to compile a high-level **:** (colon) definition, to prevent unintentional recursive references. It is reset by the word **;** (semicolon) that ends the definition.

The code field, pointing to the run-time code for a definition, causes different behaviors depending on the type of word being defined. In some implementation strategies, the code field is not required, or contains the code itself.

The cells (if any) after the code field address are called the parameter field, which is of variable length. **CONSTANT**s and **VARIABLE**s keep their data in the first cell of the parameter field. Other definitions may keep several values.

References
CODE definitions, Section 5.1
Code field addresses, Section 4.2.4
Creating dictionary entries, Section 4.2.1
Word lists, Section 4.6

1.1.3 Data Stack

Every Forth system contains at least one *data stack*. In a multitasked system, each task may have its own data stack. The stack is a cell-wide, push-down LIFO *(last-in, first-out)* list; its purpose is to contain numeric operands for Forth commands. Commands commonly expect their input parameters on this stack and leave their output results there. The stack's size is indefinite. Usually it is located at a relatively high memory address and grows downward towards areas allocated for other purposes; see your product documentation for your system's particular layout. The data stack rarely grows beyond 10–20 entries in a well-written application.

When numbers are pushed onto or popped off the stack, the remaining numbers are not moved. Instead, a pointer is adjusted to indicate the last used cell in a static memory array. On most implementations, the top-of-stack pointer is kept in a register.

Stacks typically extend toward low memory for reasons of implementation efficiency, but this is by no means required or universally true. On implementations on which the stack grows toward low memory, a *push* operation involves decrementing the stack pointer, while a *pop* involves incrementing it.

A number encountered by the text interpreter will be converted to binary and pushed onto the stack. Forth *data objects* such as **VARIABLE**s and **CONSTANT**s are defined to push their addresses or values onto the stack. Thus, the stack provides a medium of communication not only between routines but between a person and the computer. You may, for example, place numbers or addresses on the stack and then type words which act on them to produce a desired result. For example, typing:

```
12 2400 * 45 / .
```

(a) pushes the number 12 on the stack; (b) pushes 2400 over it (see Figure 4); (c) executes the multiply routine ***** which replaces both numbers by their product; (d) pushes 45 on the stack; (e) executes the divide routine **/** which divides the product by 45; and (f) executes the output routine **.** ("dot"), which removes and displays the top stack item (the quotient). All numbers put on the stack are removed, leaving the stack as it was before typing 12.

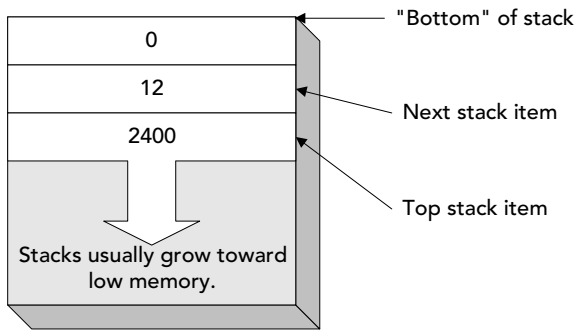

Figure 4. Items on the data stack

The standard Forth dictionary provides words for simple manipulation of single- and double-length operands on the stack: **SWAP**, **DUP**, **DROP**, **2SWAP**, etc. (covered in detail in Section 2.1).

The push-down stack simplifies the internal structure of Forth and produces naturally re-entrant routines. Passing parameters via the stack means fewer variables must be named, reducing the amount of memory required for named variables (as well as reducing the programmer's associated housekeeping).

A pointer to the top (i.e., the latest entry) of the user's stack is maintained by the system. There is also a pointer to the "bottom" of the stack, so that stack-empty or underflow conditions can be detected, and to aid in clearing the stack if an abort condition is detected.

Most Forth systems check for stack underflow only *after* executing (or attempting to execute) a word from the input stream. Underflows that occur in the process of execution will not be detected at that time (see Figure 5).

The usual result of a detected stack underflow is the message:

```
Stack empty
```

followed by a system abort.

References Forth re-entrancy, Section 2.5
Stack manipulation, Section 2.1
System abort routines, Section 2.6
Data types in stack notation, Section B.3
Stack operations, Section 2.1

1.1.4 Return Stack

Every Forth system also has a *return stack*. In a multitasked system, each task has its own return stack, usually located above its data stack in memory. Like the data stack, the return stack is a cell-wide LIFO list. It is used for system functions, but may also be accessed directly by an application program. It serves the following purposes:

- It holds return addresses for nested definitions.
- It holds loop parameters.
- It saves temporary data, such as file and record pointers for database support.
- It saves interpreter pointers when loading source text blocks.

Because the return stack has multiple uses, care must be exercised to avoid conflicts when accessing it directly.

There are no commands for directly manipulating the return stack, except those for moving one or two parameters between the data stack and the return stack.

The maximum size of the return stack for each task is specified at the time the task is defined, and remains fixed during operation; a typical size is 128 cells.

References Loading, Sections 3.4.3, 3.5.1
Loop parameters, Section 2.5.2
Data stack, Section 1.1.3
Transfers between stacks, Section 2.1.4

1.1.5 Text Interpreter

The text interpreter serves these critical functions:

- It executes the commands users type.
- It executes commands in source code stored on disk.
- It executes commands in a string whose address and length are supplied to the word **`EVALUATE`**.

The operator's terminal is the default text source. The keyboard input interrupt handler will accept characters into a text buffer called the *terminal input buffer* until a user event occurs, such as a Return or Enter keypress, function keypress, mouse click, etc. When such an event is detected, the text interpreter will process the text in the buffer. If interpretation is from source code on disk, it is buffered separately in an implementation-dependent fashion. In general, the place where the text resides that the text interpreter is parsing is called the *parse area*.

Text interpretation repeats the following steps until the parse area is exhausted or an error has occurred:

1. Starting at the beginning of the parse area, skip leading spaces and extract a word from the input string using the space character (ASCII 32) as a delimiter. Set the interpreter pointer to point to the first character beyond the delimiter. If there was no delimiter (end of input buffer was reached), set the interpreter pointer to the end of the parse area, to complete the operation. If the text is coming from a text file, the interpreter will treat any non-graphic characters as "whitespace" (equivalent to a space character).
2. Search the dictionary for a definition's name matching the input word (including case sensitivity, if applicable). If a matching definition is found, perform the interpretation behavior of the definition (if currently in interpretation mode), or perform the compilation behavior of the definition (if currently in compiling mode). Then check for stack underflow and, if no error, return to step (1). If there was a stack underflow, abort.
3. If a definition name matching the input word is not found, attempt to convert the word to a binary number (see next section). If this is successful, place the number on the data stack (if currently in interpretation mode); or, if in compiling mode, compile code that, when executed, will place this number on the

data stack (see the definition of **LITERAL**). Then return to step 1.

4. If neither the dictionary search nor the number conversion is successful, abort.

For example, on a block-based system (see Section 3.4), typing:

```
100 LOAD
```

causes these two words to be interpreted from the text input buffer. The string **100** is converted as a number and **LOAD** is found in the dictionary. This particular phrase re-directs the interpreter to the 1024-byte string stored in Block 100. This string is brought into memory from mass storage.

To let one block load another, **LOAD** saves and restores the interpreter pointers. So, in the middle of a **LOAD** of one block, that block may **LOAD** a different block and return to complete the first block. **INCLUDE** and **INCLUDE-FILE** work similarly on source code in text files (Section 3.5).

When the text interpreter executes a defining word (e.g., **CONSTANT**, **VARIABLE**, or **:**), a definition is compiled into the dictionary.

A flow diagram for the interpreting process is shown in Figure 5.

The commands **SAVE-INPUT** and **RESTORE-INPUT** are available if you wish to manually direct the text interpreter to a different area. These are not required around standard words that control the interpreter, such as **LOAD**, **INCLUDE**, and **EVALUATE**, because they handle this internally.

References

Disk blocks, Section 3.4
Text files for program source, Section 3.5
System abort routines, Section 2.6
Text interpreter words, Section 4.1
The **LOAD**ing process, Sections 3.4.3, 3.5.3
Using **EXIT** to leave a load block, Section 3.4.3.1

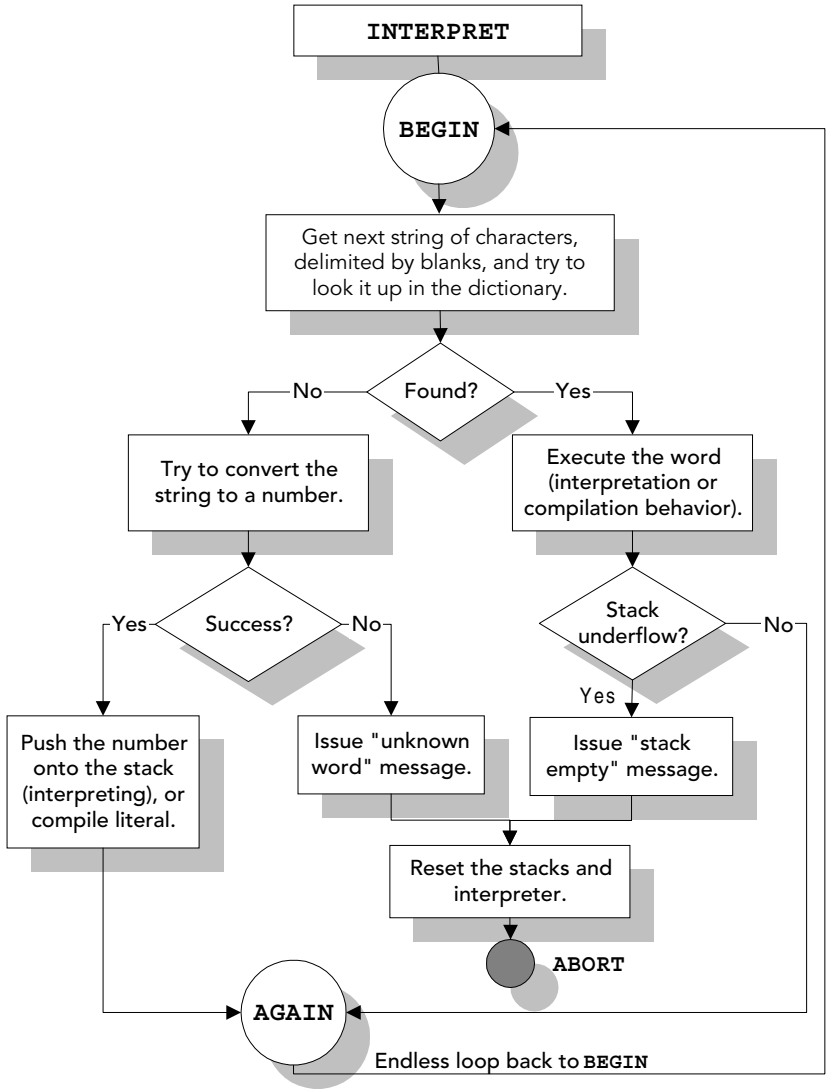

Figure 5. Flow diagram of the text interpreter

1.1.6 Numeric Input

The word **>NUMBER** is used by the text interpreter to convert strings of ASCII numerals and punctuation into binary integers that are pushed onto the stack. If there is no punctuation (except for an optional leading minus sign), a string of valid numerals is converted as a single-cell number, regardless of length. If a string of valid numerals is terminated by a decimal point, the text interpreter will convert it to a double-cell (double-precision) number regardless of length, occupying two data stack locations (high order part on top).

On eight-bit and 16-bit systems, a single-precision integer is 16 bits wide, and a double-precision integer is 32 bits wide. On 32-bit systems, these widths are 32 and 64 bits, respectively. On systems with optional floating-point routines, valid numeric strings containing an **E** or **e** (for *exponent*) will be converted as a floating-point number occupying one floating-point stack location (see Section 3.8 in this book and your product documentation for details).

Table 1: Integer precision and CPU data width

CPU Data Width	Forth Single-Precision Integer	Forth Double-Precision Integer
8 bits	16 bits	32 bits
16 bits	16 bits	32 bits
32 bits	32 bits	64 bits

Some Forth systems will interpret any number containing embedded punctuation (see below) as a double-precision integer. Single-precision numbers are recognized by their lack of special punctuation. Conversions operate on character strings of the following format:

```
[ - ] dddd [ punctuation ] dddd … delimiter
```

where `dddd` is one or more valid digits according to the current base or radix in effect for the user. The user variable **BASE** is always used as the radix. All numeric strings must be ended by a blank or a carriage return. If another character is encountered—i.e., a character which is neither a valid digit in the current base, nor punctuation, nor whitespace characters (see glossary)—an abort will occur. There must be no spaces within the number, since a space is a delimiter.

On systems allowing embedded punctuation, the characters shown in Table 2 may appear in a number. A leading minus sign, if present, must immediately precede the first digit or punctuation character.

Table 2: Valid numeric punctuation characters

Character	Description
,	comma
.	period
+	plus
-	hyphen, may appear anywhere except to the immediate left of the most-significant digit
/	slash
:	colon

All punctuation characters are functionally equivalent, including the period *(decimal point)*. The punctuation performs no other function than to set a flag that indicates its presence. On some systems, a punctuation character also causes the digits that follow it to be counted, with the count available to subsequent number-conversion words. Multiple punctuation characters may be contained in a single number; the following character strings would both convert to the same double-precision integer 123456:

```
1234.56
12,345.6
```

Glossary

BASE *(— a-addr)* Core

Return *a-addr*, the address of a cell containing the current number conversion radix. The radix is a value between 2 and 36, inclusively. It is used for both input and output conversion.

DECIMAL *(—)* Core

Sets **BASE** such that numbers will be converted using a radix of 10.

HEX *(—)* Core Ext

Sets **BASE** such that numbers will be converted using a radix of 16.

References Use of the text interpreter for number input, Section 4.1.4
Floating point input, Section 3.8.2

1.1.7 Two-stack Virtual Machine

A running Forth system presents to the programmer a *virtual machine* (VM), like a processor. It has two push-down stacks, code and data space, an "ALU" that executes instructions, and several registers. Previous sections briefly discuss the stacks and some aspects of memory use in Forth; this section will describe some features of the virtual machine as a processor.

A number of approaches to implementing the Forth VM have been developed over the years. Each has features that optimize the VM for the physical CPU on which it runs, for its intended use, or for some combination of these. We will discuss the most common implementation strategies.

The function of the Forth VM, like that of most processors, is to execute instructions. Two of the VM's registers are used to manage the stacks. Others control execution in various ways. Various implementations name and use these registers differently; for purposes of discussion in this book, we will use the names in Table 3.

Table 3: Registers in the Forth virtual machine

Name	Mnemonic	Description
`S`	data Stack pointer	Pointer to the current top of the data stack.
`R`	Return stack pointer	Pointer to the current top of the return stack.
`I`	Instruction pointer	Pointer to the next instruction (definition) to be executed; controls execution flow.
`W`	Word pointer	Pointer to the current definition being executed; used to get access to the parameter field of the definition.
`U`	User pointer	In multitasked implementations, a pointer to the currently executing task.

A standard Forth high-level, or *colon*, definition consists fundamentally of a

name followed by a number of references to previously defined words. When such a definition is invoked by a call to its name, the run-time code needs to manage the execution, in sequence, of the words making up the body of the definition. Exactly how this is done depends on the particular system and the method it uses to implement the Forth virtual machine. The implementation strategy used affects how definitions are structured and how they are executed. See the relevant product documentation for the method used in your system. There are several possibilities:

- **Indirect-threaded code.** This was the original design, and remains the most common method. Pointers to previously defined words are compiled into the executing word's parameter field. The code field of the executing word contains a pointer to machine code for an *address interpreter*, which sequentially executes those definitions by performing indirect jumps through register `I`, which is used to keep its place. When a definition calls another high-level definition, the current `I` is pushed onto the return stack; when the called definition finishes, the saved `I` is popped off of the return stack. This process is analogous to subroutine calls, and `I` in this model is analogous to a physical processor's instruction pointer.
- **Direct-threaded code.** In this model, the code field contains the actual machine code for the address interpreter, instead of a pointer to it. This is somewhat faster, but takes more memory for some classes of words. For this reason, it is most prevalent on 32-bit systems.
- **Subroutine-threaded code.** In this model, for each referenced definition in the executing word, the compiler places an in-line, jump-to-subroutine instruction with the destination address. On a 16-bit system, this technique costs extra bytes for each compiled reference. This approach is an enabling technique to allow a progression to native code generation. In this model, the underlying processor's instruction pointer is used as Forth's `I` (which usually is not a named register in such implementations).
- **Native code generation.** Going one step beyond subroutine-threaded code, this technique generates in-line machine instructions for simple primitives such as `+`, and uses jumps to other high-level routines. The result can run much faster, at the cost of size and compiler complexity. Native code can also be more difficult to debug than threaded code. This technique is characteristic of optimized systems for native Forth CPUs such as the RTX, and for 32-bit systems, where code compactness is often less critical than speed.
- **Token threading.** This technique compiles references to other words by using

a token, such as an index into a table, which is more compact than an absolute address. Token threading is a key element in MacForth. In other respects, such an implementation resembles an indirect-threaded model.

- **An additional implementation strategy** that may be combined with any of the above VM implementations supports segmented architectures. For example, the 80x86 family supports segmented address spaces. Some Forth systems take advantage of this to enable a 16-bit system to support programs larger than 64K, by using different segments for dictionary, data space, stacks, etc.

References **:**, Section 4.2.4
Compiling words and literals, Section 4.3

1.2 FORTH OPERATING SYSTEM FEATURES

Many Forth products are based on a multitasking executive; some are multi-user as well. Some implementations run in a fully *standalone mode*, in which Forth provides all drivers for the hardware attached to the system. Other versions of Forth run in a *co-resident mode,* with a host operating system such as Windows.

In the latter case, the drivers that supply I/O services for peripherals such as disk and terminals do so by issuing calls to the host system. Although co-resident systems may be somewhat slower than the standalone versions, they offer full file compatibility with the host OS, and usually are more flexible in hardware configuration. Co-resident versions of Forth usually offer all the system-level features of the native systems (including, in some cases, multi-user support on otherwise single-user systems such as MS-DOS), plus added commands for interacting with the host OS; the latter are documented in the system's product documentation.

Disk I/O is handled by Forth systems in different ways, depending on the system environment. Many systems use standard *blocks* of 1024 bytes. This fixed block size applies both to Forth source program text and to data used by Forth programs. This standard format allows I/O on different media with different physical sector or record sizes, or even on different operating systems, to be handled by one standard block handler. Forth systems in a Windows or Macintosh environment access disk using a file-based system. Blocks and files are discussed further in Section 3.4 and Section 3.5, respectively. Also see your

product documentation for details.

Forth makes it easy to control multiple tasks, either asynchronous background tasks or independent terminal service tasks. A small set of commands controls the multitasking facility. The limit on the number of tasks in the system is usually set by memory size. Since Forth definitions are naturally re-entrant, tasks rarely require much memory.

A *terminal task* has associated hardware that allows it to perform text input and output. Each terminal task has a partition that contains its stacks, private (or *user*) variable area, a scratch **PAD** (for text strings), and dictionary. A selected word list may be compiled into this partition to do a particular kind of processing which is a subset of the application but which is not available to other users.

A *background task* has a much smaller area, with only enough space for its stacks; there is no terminal associated with it, and it cannot perform text I/O. The routines the background task executes are located in a shared area or in the dictionary of one of the terminal tasks.

Forth normally runs with interrupts enabled. Interrupt vectors branch directly to the code which services the interrupting device, without any system intervention or overhead. The interrupt code is responsible for saving and restoring any registers it needs.

Interrupt code (actual assembler code) is responsible for performing any time-critical actions needed, such as reading a value from an analog device and storing it in a temporary location. The interrupt routine must also notify the task responsible for the device. Notification may take many forms, ranging from incrementing a counter, to "awakening" the task by storing in the task's status area a pointer to code that will cause the task to become active the next time the task is available. Many interrupt handlers do nothing else.

Any processing which is not time-critical can be done by a task running a routine written in high-level Forth. In effect, the time-critical aspect of servicing an interrupt is *decoupled* from the more logically complex aspects of dealing with the consequences of the event signalled by the interrupt. Thus, it is *guaranteed* that interrupts will be serviced promptly, without having to wait for task scheduling, and yet as a programmer you have the convenience of using high-level Forth executed by the responding task for the main logic of the application.

References **PAD**, Section 2.3.1
Terminal I/O, Section 3.3
Interrupts, Section 5.11

1.3 THE FORTH ASSEMBLER

Most Forth systems contain an assembler for the CPU on which the system runs. Although it offers most of the same capabilities of other assemblers, its integration into the Forth environment means it may not be fully compatible with assemblers supplied by the computer's manufacturer.

A Forth assembler produces *exactly the same code* as a conventional assembler (which means it runs at full machine speed), but does it somewhat differently. The differences are in *notation* and *procedure*, and are described in the following sections.

1.3.1 Notational Differences

Notational differences occur for two reasons:

- To improve transportability of Forth applications between processors by making assembler notation as similar as possible, without impairing the programmer's ability to access and control the processor fully; and
- To yield a compact assembler which can be resident at all times, to facilitate interactive programming and debugging.

This section describes some typical practices.

1.3.1.1 Instruction Mnemonics

Most Forth assembler mnemonics that specify assembler instructions are the same as the manufacturer's mnemonics. Occasionally, there are differences if the manufacturer uses a prefix or suffix on the mnemonic to describe something Forth specifies as a parameter or to differentiate instructions which are really different. For example, Motorola's 68xxx **ADD** instruction may be one of four variants; one popular Forth 68xxx cross-assembler uses one **ADD** instruc-

tion modified by its parameters. Intel uses **MOV** for both memory/register operations and segment register operations in the 80x86 family, whereas some Forth assemblers have different instruction names when segment registers are involved, because the internal instruction format is quite different. The net effect of these differences is usually to facilitate keeping the assembler resident at all times, without adverse impact even on relatively limited platforms, leading in turn to significantly simplified development procedures.

1.3.1.2 Addressing Modes

In all computing, there are only a few specific addressing modes (register direct, register relative, memory indirect, etc.). Notation specifying these has been standardized across all Forths, to make it easier for programmers working with several different CPUs. Naturally, this means the notation differs from the manufacturer's notation; however, all modes supported by the processor are implemented in the Forth assembler.

1.3.1.3 Instruction Format

Most assemblers encourage a four-column format, with one instruction per line, allowing space for labels, opcodes, addressing operands, and remarks. In Forth, the opcode itself is a Forth command which assembles the instruction according to addressing operands passed on the stack. This leads to a format in which the addressing mode specifiers precede the opcode.

1.3.1.4 Labels, Branches, and Structures

Forth assemblers support structured programming in the same way high-level Forth does. Arbitrary branching to labelled locations is discouraged; on the other hand, structures such as **BEGIN** … **UNTIL** and **IF** … **ELSE** … **THEN** are available in the assembler, implemented as macros that assemble appropriate conditional and unconditional branches.

1.3.2 Procedural Differences

1.3.2.1 Resident Assembler

The Forth assembler is normally resident at all times. This means a programmer can assemble code at any time, either from source on disk or by typing it directly from the terminal. Regardless of where the code comes from, the assembled version will be the same.

1.3.2.2 Immediately Executable Code

In conventional programming, assemblers leave their object code in a file, which must be integrated with code in files from high-level language compilers (if any) by a linker before the resultant program can be loaded into memory for testing. The resident Forth assembler assembles the code directly into memory in executable form, thus avoiding this cumbersome procedure.

1.3.2.3 Relationship to Other Routines

The Forth assembler is used to write short, named routines that function just like routines written in high-level Forth; that is, when the name of a routine is invoked, it will be executed. Like other Forth routines, code routines normally expect their arguments on the stack and leave their results there. Within a code definition, one may refer to defined constants (to get a value), variables (to get an address), or other defined data types. Code routines may be called from high-level definitions just as other Forth words are, but cannot themselves call high-level definitions.

1.3.2.4 Register Usage

A Forth system runs on a *virtual machine*. For optimum performance, some of its virtual registers are kept in actual hardware registers, which are permanently assigned. The product documentation for each Forth system documents the register assignments for that CPU. Some registers may be designated as *scratch*, meaning they can be used within a code routine without saving or restoring; those containing Forth pointers must be saved and restored, if needed. Forth system registers are given names which make refer-

ences to them in code easy and readable. Because most Forth code routines can do what they need by using the designated scratch registers, there is less need to save and restore registers than in conventional programming.

References Principles of Forth assemblers, Section 5
The product documentation for a specific Forth system

1.4 Documentation and Programmer Aids

In Forth, as in all other languages, the primary responsibility for producing readable code lies with the programmer. Forth does, however, support the programmer's efforts to produce easily managed code by providing aids to internal documentation. In addition to these, we recommend that each Forth programming group adopt uniform editorial and naming standards and conventions. Sample standards adopted by some groups are offered in Section 6. Although readability is an aesthetic and rather personal value, a set of standards that all members of a group adhere to will significantly improve the ability of members of the group to share code and to support one another.

1.4.1 Comments

Comments embedded in Forth source are enclosed in parentheses. For example:

```
( This is a comment)
```

The word **(** must have a space after it, so that it can be recognized and executed as a command (to begin the comment). A space is not needed before the closing right parenthesis delimiter. On most systems, the \ (backslash) character is also defined, indicating that the entire remainder of the current line of source code is a comment.

The word **.(** (note the preceding dot) is like **(**, but begins a comment that will be displayed when it is encountered. If it occurs inside a definition, the text will be displayed when the definition is compiled, not when it is executed. It is commonly used in source code to indicate progress in compilation, e.g.:

```
.( Begin application compilation)
```

Forth comments are most often used to give a picture of a word's stack arguments and results; for example, a high-level definition of the Forth word = is:

```
: = ( n n -- t )   - NOT ;
```

The dashes in the comment separate a word's arguments (on the left) from its results. By convention, certain letters have specific, common meanings:

Table 4: Common stack notation

Word	Description
n	A single-cell signed integer.
u	A single-cell unsigned integer.
t	A single-cell Boolean value (zero is *false*, non-zero is *true*).
addr	An arbitrary single-cell byte address.
d	A double-cell signed integer.

Thus, in the example above, the word = expects two single-cell integers and returns a truth flag.

Words that have separate interpretive and run-time behaviors should have comments for both sections:

```
: CONSTANT ( n -- )   CREATE ,  DOES>
   ( addr -- n )   @ ;
```

References

Stack notation, Section 2.1.1
Data types in stack notation, Section B.3

Glossary

(*(—)* Core, File

Begin a comment. Stop compilation or interpretation and parse the characters that follow, looking for a right parenthesis) which closes the comment.

.(*(—)* Core Ext

Like (, but begin a comment that will be sent to the display device when it is encountered. Terminated by a right parenthesis).

\ *(—)* Block Ext, Core Ext

Begin a comment which includes the entire remainder of the current line of source code. No closing delimiter is needed.

1.4.2 Locating Command Source

After code has been compiled from source files, the **LOCATE** command can call up the source code for a command, given the command name. For example, the command:

```
LOCATE /STRING
```

starts the editor, opens the correct source block or file, and positions the cursor at the start of the definition of **/STRING**:

```
: /STRING ( c-addr1 len1 u -- c-addr2 len2)
   OVER MIN >R  SWAP R@ +  SWAP R> - ;
```

Similarly, if the compiler encounters an error and aborts, you may go directly to the block (or file) and line at which the error occurred by typing **L**. This is particularly convenient if you have a linked editor, as you can immediately repair the error and recompile.

Glossary

LOCATE <name> *(—)* common usage

If *name* is the name of a definition that has been compiled from source code, display the source code for *name*. On some systems, the phrase **VIEW** *name* performs a similar function.

L *(—)* common usage

Show the current source code file or block and the current cursor position in it. If used after a compiling error, point to the source code that caused the error.

1.4.3 Cross-references

This tool finds all the places a word is used. The syntax is:

```
WHERE <name>
```

It gives the first line of the definition of the word *name*, followed by each line of source code in the currently compiled program that contains *name*.

If the same name has been redefined, **WHERE** gives the references for each definition separately. The shortcut:

```
WH <name>
```

does the same thing.

This command is not the same as a source search, since it is based on the code you have currently compiled and are debugging. This means you will be spared instances of *name* in files you aren't using.

Glossary

WH <name> *(—)* common usage
Short synonym for **WHERE**, defined for typing convenience.

WHERE <name> *(—)* common usage
Display all the places in the currently compiled program where *name* has been used, showing any re-definitions separately.

1.4.4 Decompiler and Disassembler

The disassembler/decompiler is used to reconstruct readable source code from **CODE** and **:** (colon) definitions. This is useful as a cross-check, whenever a new definition fails to work as expected.

The command **SEE** *name* disassembles both **CODE** commands and colon definitions. For example, the source definition for **/STRING** is:

```
: /STRING ( c-addr1 len1 u -- c-addr2 len2)
   OVER MIN >R  SWAP R@ +  SWAP R> - ;
```

but if you decompile it (on a FORTH, Inc. 68000 cross-compiler, for example), you get:

```
SEE /STRING
9B6   4 A6) A6 -) MOV
9BA   ' MIN BSR
9BE   A6 )+ A7 -) MOV
9C0   ' SWAP BSR
9C4   A7 ) A6 -) MOV
9C6   A6 )+ D0 MOV
9C8   D0 A6 ) ADD
9CA   ' SWAP BSR
9CE   A7 )+ A6 -) MOV
9D0   A6 )+ D0 MOV
9D2   D0 A6 ) SUB
9D4   RTS  ok(T)
```

This example clearly shows the combination of in-line code and subroutine calls in this subroutine-threaded implementation.

An alternative approach is to start disassembly or decompilation at some address. This is useful for decompiling headless code, such as code preceded only by a **LABEL**. The command to disassemble a **CODE** definition, given an address *addr*, is:

```
<addr> DASM
```

The word **.'** is becoming increasingly popular in this debugging context, though it is not in Standard Forth nor in all systems. It attempts to identify the definition in which an address occurs. For example, given **/STRING** above, you could type:

```
HEX 9BE .'
```

and get:

```
/STRING +08  ok(T)
```

Glossary

SEE <name> *(—)* Tools

Reconstruct the source code for *name*, using as necessary a decompiler for high-level definitions or a disassembler for code definitions.

DASM *(addr —)* common usage

Begin disassembly at the address *addr* on top of the stack. The disassembler stops when it encounters an unconditional transfer of control outside the range of the definition, such as returns from interrupt or from subroutines, branches, and jumps. Subroutine calls are excluded, as control is assumed to return to the location following the call.

.' *(addr —)* common usage

Display the name of the nearest definition before *addr*, and the offset of *addr* from the beginning of that definition. "dot-tick"

1.5 Interactive Programming—An Example

The Forth language was designed from first principles to support an interactive development style. By developing a very simple application in this section, we will show how this style translates into practice.

The general process of developing a program in Forth is consistent with the recommended development practices of *top-down design* and *bottom-up coding and testing*. However, Forth adds another element: extreme modularity. You don't write page after page of code and then try to figure out why it doesn't work; instead, you write a few very brief definitions and then exercise them, one by one.

Suppose we are designing a washing machine. The overall, highest-level definition might be:

```
: WASHER   WASH SPIN RINSE SPIN ;
```

The colon indicates that a new word is being defined; following it is the name of that new word, **WASHER**. The remainder are the previously defined words that comprise this definition. Finally, the definition is terminated by a semi-colon.

Typically, we design the highest-level routines first. This approach leads to conceptually correct solutions with a minimum of effort. In Forth, words must be compiled before they can be referenced. Thus, a listing begins with the most primitive definitions and ends with the highest-level words. If the higher-level words are entered first, lower-level routines are added above them in the listing.

Figure 6 shows a complete listing of the washing machine example. This is a typical Forth *block* of source code. Comments are in parentheses. In this example, lines 1–3 define named constants, with hex values representing hardware port addresses. Lines 5–15 define, in sequence, the application words that perform the work.

```
 0 ( Washing Machine Application )      HEX
 1 7000 CONSTANT MOTOR   7006 CONSTANT DETERGENT   700A CONSTANT CLUTCH
 2 7002 CONSTANT VALVE   7008 CONSTANT TIMER       7010 CONSTANT LEVEL
 3 7004 CONSTANT FAUCETS      DECIMAL
 4
 5 : ON ( port)    -1 SWAP  OUTPUT ;      : OFF ( port)   0 SWAP OUTPUT ;
 6 : SECONDS ( n)   1000 * MS ;           : MINUTES ( n)   60 * SECONDS ;
 7 : ADD ( port)   DUP ON  10 SECONDS  OFF ;
 8 : TILL-FULL   BEGIN  LEVEL INPUT  UNTIL ;
 9 : DRAIN   VALVE ON  3 MINUTES  VALVE OFF ;
10 : AGITATE   MOTOR ON  10 MINUTES  MOTOR OFF ;
11 : SPIN   CLUTCH ON  MOTOR ON  5 MINUTES  MOTOR OFF  CLUTCH OFF ;
12 : FILL   FAUCETS ON  TILL-FULL  FAUCETS OFF ;
13 : WASH   FILL  DETERGENT ADD  AGITATE  DRAIN ;
14 : RINSE   FILL  AGITATE  DRAIN ;
15 : WASHER   WASH  SPIN  RINSE  SPIN ;
```

Figure 6. Example of a control program that runs a washing machine

The code in this example is nearly self-documenting; the few comments show the parameters being passed to certain words. Forth allows as many comments as desired, with no penalty in object code size or performance.

When reading,

```
: WASHER   WASH  SPIN  RINSE  SPIN ;
```

it is obvious what **RINSE** does. To determine *how* it does it, you read:

```
: RINSE   FILL  AGITATE  DRAIN ;
```

When you wonder how **FILL** works, you find:

```
: FILL   FAUCETS ON  TILL-FULL    FAUCETS OFF ;
```

Reading further, one finds that **FAUCETS** is simply a constant which returns

the address of the port that controls the faucet, while **ON** is a simple word that turns on the bits at that address.

Even from this simple example, it may be clear that Forth is not so much a language, as a tool for building application-oriented command sets. The definition of **WASHER** is based not on low-level Forth words, but on washing-machine words like **SPIN** and **RINSE**.

Because Forth is extensible, Forth programmers write collections of words that apply to the problem at hand. The power of Forth, which is simple and universal to begin with, grows rapidly as words are defined in terms of previously defined words. Each successive, newer word becomes more powerful and more specific. The final program becomes as readable as you wish to make it.

When developing this program, you would follow your top-down logic, as described above. But when the time comes to test it, you see the real convenience of Forth's interactivity.

If your hardware is available, your first step would be to see if it works. Even without the code in Figure 6, you could read and write the hardware registers by typing phrases such as:

```
HEX 7010 INPUT .
```

This would read the water-level register at 7010_H and display its value. And you could type:

```
-1 7002 OUTPUT   0 7002 OUTPUT
```

to see if the valve opens and closes.

If the hardware is unavailable, you might temporarily re-define the words **MOTOR**, etc., as variables you can read and write, and so test the rest of the logic.

You can load your block of source (as described in Section 3.4.3), whereupon all its definitions are available for testing. You can further exercise your I/O by typing phrases such as:

MOTOR ON or **MOTOR OFF**

to see what happens. Then you can exercise your low-level words, such as:

```
DETERGENT ADD
```

and so on, until your highest-level words are tested.

As you work, you can use any of the additional programmer aids described in Section 2.1.5. You can also easily change your code and re-load it. But your main ally is the intrinsically interactive nature of Forth itself.

References

Disk and block layout and design, Sections 3.4, 6.1
Stack notation conventions, Section 2.1, Table 11, and Section B.3
Number base, Sections 1.1.6, 2.4
Numeric output (the word `.`), Section 2.4.1
Programmer conveniences, Section 2.1.5

2. FORTH FUNDAMENTALS

This section defines the major elements of the Forth language. These words are grouped into categories. Except where noted as deriving from "common usage," all words are found in, and comply with, the American National Standard for the Forth language (*ANSI X3.215:1994*, equivalent to *ISO/IEC 15145:1997)*, commonly referred to here as *Standard Forth*. Appendix A,"Glossary & Notation" on page 203 provides definitions of many of the terms used in this section, as well as a detailed description of the notation conventions.

2.1 STACK OPERATIONS

Forth is based on an architecture incorporating push-down stacks (last-in, first-out lists). The *data stack* is used primarily for passing parameters between procedures. The *return stack* is used primarily for system functions, such as procedure return addresses, loop parameters, etc.

Stack operators work on data that are on one or more of the stacks. The words defined in this section use the stack as the major source and destination for their operands. Many other Forth words also result in modification of the stack, and are described in the sections of this manual that deal with their primary functions. Besides the stack operators discussed in this manual, stack manipulation words that relate to assembly language are covered in Section 5 and in your Forth system's documentation.

2.1.1 Stack Notation

Stack parameters used as input to and output from a procedure are described using the notation:

(stack-id before — after)

Operations that use the stack usually require that a certain number of items be present on the stack, and then leave another number of items on the stack as results. Most operations remove their operands, leaving only the results. To help see an operation's effect on the number and type of items on the stack, each word has a *stack notation*.

Individual stack items are depicted using the notation in Table 11, Section B.3. Any other, special notation will be explained when used. Where several arguments are of the same type, and clarity demands that they be distinguished, numeric subscripts are used.

If you type several numbers on a line, the rightmost will end up on top of the stack. As a result, we show multiple stack arguments with the top element to the right. If alternate conditions may exist, they are separated by a vertical bar (|), meaning "or." For example, the notation *(— n_1 | n_2 n_3)* indicates a word that may leave either one or two stack items; and *(— addr | 0)* indicates that the procedure takes no input and returns either a valid address or zero.

Please remember that the items shown in a word's stack notation are relative to the top of the stack and do not affect any stack items that may be below the lowest stack item referenced by the operation. For example, *(x_1 x_2 — x_3)* describes an operation that uses the top two stack items and leaves a different, one-item result. Therefore, if the stack initially contained three items, execution would result in a stack of two items, with the bottom item unchanged and the top item derived as a result of the operation.

Some procedures have stack effects both when they are compiled and when they are executed. The stack effects shown in this manual refer to the execution-time behavior unless specifically noted, because this is usually the behavior of most interest to a programmer.

Where an operation is described that uses more than one stack, the data stack *stack-id* is S: and the return stack *stack-id* is R:. When no confusion is possible, the data stack *stack-id* may be omitted.

With the addition of the floating-point stack (see Section 3.8), it becomes necessary to document its contents, as well. Floating-point stack comments follow the data stack comments, and are indicated by F:. If a command does not affect

the floating-point stack, only the data stack comments are shown, and vice versa. If neither stack is affected, a null data stack comment is shown.

For example:

```
: SF@ ( a-addr  —  ) ( F:  — r )
```

indicates that an address is removed from the data stack, and a floating-point number is pushed on the floating-point stack by the execution of **SF@**.

```
: F. ( F: r  — )
```

indicates that there are no data stack arguments, and that a floating-point number is removed from the floating-point stack by the execution of **F.**.

References Data stack, Section 1.1.3
Data types in stack notation, Section B.3

2.1.2 Data Stack Manipulation Operations

This category of stack operations contains words which manipulate the contents of the data stack without performing arithmetic, logical, or memory reference operations.

Glossary

2DROP (x_1 x_2 —) Core
Remove the top pair of cells from the stack. The cell values may or may not be related. "two-drop"

2DUP (x_1 x_2 — x_1 x_2 x_1 x_2) Core
Duplicate the top cell pair x_1 x_2. "two-dup"

2OVER (x_1 x_2 x_3 x_4 — x_1 x_2 x_3 x_4 x_1 x_2) Core
Copy cell pair x_1 x_2 to the top of the stack. "two-over"

2ROT (x_1 x_2 x_3 x_4 x_5 x_6 — x_3 x_4 x_5 x_6 x_1 x_2) Double Ext
Rotate the top three cell pairs on the stack, bringing cell pair x_1 x_2 to the top of

the stack. "two-rote"

2SWAP ($x_1\ x_2\ x_3\ x_4$ — $x_3\ x_4\ x_1\ x_2$) Core

Exchange the top two cell pairs. "two-swap"

?DUP (x — 0 | $x\ x$) Core

Conditionally duplicate the top item on the stack, if its value is non-zero. "question-dup"

Logically equivalent to: **DUP IF DUP THEN**

DEPTH (— $+n$) Core

Return the number of single-cell values that were on the stack before this word executed. **DEPTH** will return 2 for each double-precision integer on the stack.

DROP (x —) Core

Remove the top entry from the stack.

DUP (x — $x\ x$) Core

Duplicate the top entry on the stack.

NIP ($x_1\ x_2$ — x_2) Core Ext

Drop the second item on the stack, leaving the top unchanged.

OVER ($x_1\ x_2$ — $x_1\ x_2\ x_1$) Core

Place a copy of x_1 on top of the stack.

PICK ($+n$ — x) Core Ext

Place a copy of the nth stack entry on top of the stack. The zeroth item is the top of the stack; i.e., **0 PICK** is equivalent to **DUP** and **1 PICK** is equivalent to **OVER**.

ROLL ($+n$) Core Ext

Move the nth stack entry to the top of the stack, moving down all the stack entries in between. The zeroth item is the top of the stack; i.e., **0 ROLL** does nothing, **1 ROLL** is equivalent to **SWAP**, **2 ROLL** is equivalent to **ROT**.

ROT ($x_1\ x_2\ x_3$ — $x_2\ x_3\ x_1$) Core

Rotate the top three items on the stack.

SWAP ($x_1\ x_2$ — $x_2\ x_1$) Core

Exchange the top two items on the stack.

TUCK (x_1 x_2 — x_2 x_1 x_2) Core Ext

Place a copy of the top stack item below the second stack item.

2.1.3 Memory Stack Operations

This category of operations allows you to reference memory by using addresses that are on the stack.

Glossary

! (*x a-addr* —) Core

Store *x* at the cell at *a-addr*, removing both from the stack. "store"

+! (*n a-addr* —) Core

Add *n* to the contents of the cell at *a-addr* and store the result in the cell at *a-addr*, removing both from the stack. "plus-store"

2! (x_1 x_2 *a-addr* —) Core

Store the cell pair x_1 x_2 into the two cells beginning at *a-addr*, removing three cells from the stack. "two-store"

2@ (*a-addr* — x_1 x_2) Core

Push the cell pair x_1 x_2 at *a-addr* onto the top of the stack. "two-fetch"

@ (*a-addr* — *x*) Core

Replace *a-addr* with the contents of the cell at *a-addr*. "fetch"

C! (*b c-addr* —) Core

Store the low-order byte of the second stack item at *c-addr*, removing both from the stack. "C-store"

C+! (*b c-addr* —) common usage

Add the low-order byte of the second stack item to the byte at *c-addr*, removing both from the stack. "C-plus store"

C@ (*c-addr* — *b*) Core

Replace *c-addr* with the contents of the byte at *c-addr*. The byte fetched is stored in the low-order byte of the top stack item, with the remaining bits cleared to zero. "C-fetch"

2.1.4 Return Stack Manipulation Operations

The *return stack* is so named because it is used by the Forth virtual machine (VM) to keep track of where Forth words will return when they have finished executing. When a high-level Forth word invokes a previously defined Forth word, the address of the next word to be executed is pushed onto the return stack; it will be popped off the return stack when the called word is finished, so execution can resume where it left off.

The return stack is a convenient place to keep frequently used values (by using the words **>R**, **R@**, and **R>**), but it must be cleared before an executing word reaches the end of the current definition, or the virtual machine will return to the "address" on the return stack. This behavior can be useful; for example, on many systems, the definition:

```
: VECTOR ( xt -- )  >R ;
```

will act like the word **EXECUTE**, but will only execute **:** definitions. **VECTOR** works by pushing a word's execution token onto the return stack. Therefore, when the end of the definition pops the return stack into the VM's register **I** (or its implementation-dependent equivalent), the VM will begin to execute the word whose address was on the stack for **VECTOR**. This works on all systems in which *xt*s are actual return addresses.

If you use the return stack for temporary storage, you must be aware that this is also a system resource, and obey the following restrictions:

- Your program must not access values on the return stack (using **R@**, **R>**, **2R@**, or **2R>**) that it did not place there using **>R** or **2>R**.
- When inside a **DO** loop, your program must not access values that were placed on the return stack *before* the loop was entered.
- All values placed on the return stack *within* a **DO** loop must be removed before **I**, **J**, **LOOP**, **+LOOP**, **UNLOOP**, or **LEAVE** is executed.
- All values placed on the return stack within a definition must be removed before the end of the definition or before **EXIT** is executed.

The glossary below documents operations that involve both the return stack and the data stack.

Glossary

2>R (x_1 x_2 —) (R: — x_1 x_2) Core Ext

Pop the top two cells from the data stack and push them onto the return stack. "two-to-R"

2R> (— x_1 x_2) (R: x_1 x_2 —) Core Ext

Pop the top two cells from the return stack and push them onto the data stack. **2R>** is the inverse of **2>R**. "two-R-from"

2R@ (— x_1 x_2) (R: x_1 x_2 — x_1 x_2) Core Ext

Push a copy of the top two return stack cells onto the data stack. "two-R-fetch"

>R (x —) (R: — x) Core

Remove the item on top of the data stack and put it onto the return stack. "to-R"

R> (— x) (R: x —) Core

Remove the item on the top of the return stack and put it onto the data stack. "R-from"

R@ (— x) (R: x — x) Core

Place a copy of the item on top of the return stack onto the data stack. "R-fetch"

References Counting **LOOP**s (**DO**), Section 2.5.2
EXECUTE, Section 2.5.6

2.1.5 Programmer Conveniences

The words in this section are intended as programming aids. They may be used interpretively at the keyboard, or inside definitions—except for **'** (tick); the equivalent of **'** inside a definition is the word **[']**. Because compiling new definitions in Forth is so quick, you are encouraged to create, test, and debug definitions to aid in developing an application.

Glossary

' <name> *(— xt)* Core

Search dictionary for *name* and leave its execution token on the stack. Abort if *name* cannot be found. "tick"

.S *(—)* Tools

Display the contents of the data stack using the current base. Stack contents remain unchanged. "dot-S"

? *(a-addr —)* Tools

Fetch the contents of the given address and display the result according to the current conversion radix. "question"

Equivalent to the phrase: **@ .**

DUMP *(addr +n —)* Tools

Display the contents of a memory region of length *+n* starting at *addr*:

```
<addr> <+n> DUMP
```

Output is formatted with the address on the left and up to eight values on a line. The output conversion radix is the current value of **BASE** (on some systems this word always outputs in hex). Two cells are removed from the stack.

ENVIRONMENT? *(c-addr u — false | i*x true)* Core

This word is used to inquire about the values of system parameters and the existence of options. See Section 3.2 for a full description. "environment-query"

WORDS *(—)* Tools

List all the definition names in the first word list of the search order.

References **[']**, Section 4.1.3
Environmental interrogation, Section 3.2
Search orders, Section 4.6.1

2.2 Arithmetic and Logical Operations

Forth offers a comprehensive set of commands for performing arithmetic and logical functions. The functions in a standard system are optimized for integer arithmetic, because not all processors have hardware floating-point capability and software floating point is too slow for most real-time applications. All Forth systems provide words to perform fast, precise, scaled-integer computations; many provide fixed-point fraction computations, as well. On systems with hardware floating-point capability, many implementations include an optional, complete set of floating-point operations, including an assembler. See Section 3.8 in this manual and the product documentation for these systems for details.

2.2.1 Arithmetic and Shift Operators

In order to achieve maximum performance, each version of Forth implements most arithmetic primitives to use the internal behavior of that particular processor's hardware multiply and divide instructions. Therefore, to find out at the bit level what these primitives do, you should consult either the manufacturer's hardware description or the implementation's detailed description of these functions.

In particular, signed integer division where only one operand (either dividend or divisor) is negative and there is a remainder may produce different, but equally valid, results on different implementations. The two possibilities are *floored* and *symmetric* division. In floored division, the remainder carries the sign of the divisor and the quotient is rounded to its arithmetic floor (towards negative infinity). In symmetric division, the remainder carries the sign of the dividend and the quotient is rounded towards zero, or truncated. For example, dividing -10 by 7 can give a quotient of -2 and remainder of 4 (floored), or a quotient of -1 and remainder of -3 (symmetric).

Most hardware multiply and divide instructions are symmetric, so floored division operations are likely to be slower. However, some applications (such as graphics) require floored division in order to get a continuous function through zero. Consult your system's documentation to learn its behavior.

The following general guidelines may help you use these arithmetic operators:

- The order of arguments to order-dependent operators (e.g., **-** and **/**) is such that, if the operator were moved to an infix position, it would algebraically describe the result. Some examples:

Forth	Algebraic
`a b -`	`a - b`
`a b /`	`a / b`
`a b c */`	`a * b / c`

- All arithmetic words starting with the letter **U** are unsigned; others are normally signed. The exception to this rule is that, on most systems, **M*/** requires a positive divisor.
- When executing operations involving address calculations, use the words **CELL+**, **CELLS**, **CHAR+**, and **CHARS** as appropriate to convert logical values to bytes, rather than to absolute numbers. For example, to increment an address by three cells on a 32-bit system, use **3 CELLS +**, not **12 +**; this makes the code portable to systems that may have different cell widths.

These operators perform arithmetic and logical functions on numbers that are on the stack. In general, the operands are removed (popped) from the stack and the results are left on the stack.

Glossary **Single-Precision Operations**

***** (n_1 n_2 — n_3) Core

Multiply n_1 by n_2 leaving the product n_3. "star"

***/** (n_1 n_2 n_3 — n_4) Core

Multiply n_1 by n_2, producing an intermediate double-cell result d. Divide d by n_3, giving the single-cell quotient n_4. "star-slash"

***/MOD** (n_1 n_2 n_3 — n_4 n_5) Core

Multiply n_1 by n_2, producing intermediate double-cell result d. Divide d by n_3, giving single-cell remainder n_4 and single-cell quotient n_5. "star-slash-mod"

+ (n_1 n_2 — n_3) Core

Add n_1 to n_2, leaving the sum n_3. "plus"

- (n_1 n_2 — n_3) Core

Subtract n_2 from n_1, leaving the difference n_3. "minus"

/ (n_1 n_2 — n_3) Core

Divide n_1 by n_2, leaving the quotient n_3. See the discussion at the beginning of this section about floored and symmetric division. "slash"

/MOD (n_1 n_2 — n_3 n_4) Core

Divide n_1 by n_2, leaving the remainder n_3 and the quotient n_4. "slash-mod"

1+ (n_1 — n_2) Core

Add one to n_1, leaving n_2. "one-plus"

1- (n_1 — n_2) Core

Subtract one from n_1, leaving n_2. "one-minus"

2+ (n_1 — n_2) common usage

Add two to n_1, leaving n_2. "two-plus"

2- (n_1 — n_2) common usage

Subtract two from n_1, leaving n_2. "two-minus"

2* (x_1 — x_2) Core

Return x_2, the result of shifting x_1 one bit toward the most-significant bit, filling the least-significant bit with zero (same as **1 LSHIFT**). "two-star"

2/ (x_1 — x_2) Core

Return x_2, the result of shifting x_1 one bit towards the least-significant bit, leaving the most-significant bit unchanged. "two-slash"

CELL+ (*a-addr*$_1$ — *a-addr*$_2$) Core

Add the size in bytes of a cell to *a-addr*$_1$, giving *a-addr*$_2$. Equivalent to **2+** on a 16-bit system and to **4 +** on a 32-bit system. "cell-plus"

CELLS (n_1 — n_2) Core

Return n_2, the size in bytes of n_1 cells.

CHAR+ (*c-addr*$_1$ — *c-addr*$_2$) Core

Add the size in bytes of a character to *c-addr*$_1$, giving *c-addr*$_2$. "care-plus"

CHARS (n_1 — n_2) Core

Return n_2, the size in bytes of n_1 characters. On many systems, this word is a no-op. "cares"

LSHIFT (x_1 u — x_2) Core

Perform a logical left shift of u places on x_1, giving x_2. Fill the vacated least-significant bits with zeroes. "L-shift"

MOD (n_1 n_2 — n_3) Core

Divide n_1 by n_2, giving the remainder n_3.

RSHIFT (x_1 u — x_2) Core

Perform a logical right shift of u places on x_1, giving x_2. Fill the vacated most-significant bits with zeroes. "R-shift"

Double-precision Operations

D+ (d_1 d_2 — d_3) Double

Add d_1 to d_2, leaving the sum d_3. "D-plus"

D- (d_1 d_2 — d_3) Double

Subtract d_2 from d_1, leaving the difference d_3. "D-minus"

D2* (xd_1 — xd_2) Double

Return xd_2, the result of shifting xd_1 one bit toward the most-significant bit and filling the least-significant bit with zero. "D-two-star"

D2/ (xd_1 — xd_2) Double

Return xd_2, the result of shifting xd_1 one bit towards the least-significant bit and leaving the most-significant bit unchanged. "D-two-slash"

Mixed-precision Operations

D>S (d — n) Double

Convert double-precision number d to its single-precision equivalent n. Results are undefined if d is outside the range of a signed single-cell number. "D-to-S"

FM/MOD (d n_1 — n_2 n_3) Core

Divide d by n_1, using floored division, giving quotient n_3 and remainder n_2. All arguments are signed. This word and **SM/REM** will produce different results on the same data when exactly one argument is negative and there is a remainder. "F-M-slash-mod"

M* (n_1 n_2 — d) Core

Multiply n_1 by n_2, leaving the double-precision result d. "M-star"

M*/ (d_1 n_1 $+n_2$ — d_2) Double

Multiply d_1 by n_1, producing a triple-cell intermediate result t. Divide t by the positive number n_2 giving the double-cell quotient d_2. If double-precision multiplication or division only is needed, this word may be used with either n_1 or n_2 set equal to 1. "M-star-slash"

M+ (d_1 n — d_2) Double

Add n to d_1, leaving the sum d_2. "M-plus"

M- (d_1 n — d_2) common usage

Subtract n from d_1, leaving the difference d_2. "M-minus"

M/ (d n_1 — n_2) common usage

Divide d by n_1, leaving the single-precision quotient n_2. This word does not perform an overflow check. "M-slash"

S>D (n — d) Core

Convert a single-precision number n to its double-precision equivalent d with the same numerical value. "S-to-D"

SM/REM (d n_1 — n_2 n_3) Core

Divide d by n_1, using symmetric division, giving quotient n_3 and remainder n_2. All arguments are signed. This word and **FM/MOD** will produce different results on the same data when exactly one argument is negative and there is a remainder. "S-M-slash-rem"

T* (d n — t) common usage

Multiply d by n, yielding a triple-precision result t. Used in **M*/**. "T-star"

T/ (t $+n$ — d) common usage

Divide a triple-precision number t by the positive number $+n$, leaving a double-precision result d. Used in **M*/**. "T-slash"

UM/MOD (ud u_1 — u_2 u_3) Core

Divide ud by u_1, leaving remainder u_2 and quotient u_3. This operation is called **UM/MOD** because it assumes the arguments are unsigned, and it produces unsigned results. Compare with **SM/REM** and **FM/MOD**. "U-M-slash-mod"

UM* (u_1 u_2 — ud) Core

Multiply u_1 by u_2, leaving the double-precision result ud. All values and arithmetic are unsigned. "U-M-star"

2.2.2 Logical and Relational Operations

As in the case of arithmetic operations, Forth's implementation of logical and relational operations optimizes speed and simplicity. The words described in this section provide a rich, flexible set of logical operations.

Glossary **Single-Precision Logical Operations**

ABS (n — $+n$) Core

Replace the top stack item with its absolute value.

AND (x_1 x_2 — x_3) Core

Return x_3, the bit-by-bit logical *and* of x_1 with x_2.

INVERT (x_1 — x_2) Core

Invert all bits of x_1, giving its logical inverse x_2.

MAX (n_1 n_2 — n_3) Core

Return n_3, the greater of n_1 and n_2.

MIN (n_1 n_2 — n_3) Core

Return n_3, the lesser of n_1 and n_2.

NEGATE (n — $-n$) Core

Change the sign of the top stack value; if the value was negative, it becomes positive. The phrase **NEGATE 1-** is equivalent to **INVERT** (one's complement of the input value).

OR (x_1 x_2 — x_3) Core

Return x_3, the bit-by-bit inclusive *or* of x_1 with x_2.

WITHIN (x_1 x_2 x_3 — *flag*) Core

Return *true* if x_1 is greater than or equal to x_2 and less than x_3. The values may all be either unsigned integers or signed integers, but must all be the same type.

XOR (x_1 x_2 — x_3) Core

Return x_3, the bit-by-bit exclusive *or* of x_1 with x_2. The phrase **-1 XOR** is equivalent to **INVERT** (one's complement of the input value).

Double-Precision Logical Operations

DABS (d — $+d$) Double

Return the absolute value of a double-precision stack value.

DMAX (d_1 d_2 — d_3) Double

Return d_3, the larger of d_1 and d_2.

DMIN (d_1 d_2 — d_3) Double

Return d_3, the lesser of d_1 and d_2.

DNEGATE (d — $-d$) Double

Change the sign of a double-precision stack value. Analogous to **NEGATE**.

2.2.3 Comparison and Testing Operations

These operations leave on the stack a number that is based upon a test of the contents of one or more items on top of the stack. In general, the test is destructive, in that it replaces the item(s) tested with the numerical results of the test. All numbers in Forth may be interpreted as *true* or *false* values; zero equals *false,* and *any* non-zero value equals *true*. The words below, which perform explicit tests, return -1 for *true.* Comparison and testing operations generally precede an **IF**, **WHILE**, or **UNTIL** construct.

You may also use **-** (minus) or **D-** as a *not-equal* test, because they return a non-zero difference if the two single- or double-precision numbers are unequal.

Glossary

0< (n — *flag*) Core

Return *flag*, which is *true* if and only if n is less than zero. "zero-less-than"

0<> (*n* — *flag*) Core Ext
Return *flag*, which is *true* if and only if *n* is not equal to zero. "zero-not-equal"

0= (*n* — *flag*) Core
Return *flag*, which is *true* if and only if *n* is equal to zero. "zero-equal"

0> (*n* — *flag*) Core Ext
Return *flag*, which is *true* if and only if *n* is greater than zero. "zero-greater-than"

< (n_1 n_2 — *flag*) Core
Return *flag*, which is *true* if and only if n_1 is less than n_2. "less-than"

<> (n_1 n_2 — *flag*) Core Ext
Return *flag*, which is *true* if and only if n_1 is not equal to n_2. "not-equal"

= (n_1 n_2 — *flag*) Core
Return *flag*, which is *true* if and only if n_1 is equal to n_2. "equal"

> (n_1 n_2 — *flag*) Core
Return *flag*, which is *true* if and only if n_1 is greater than n_2. "greater-than"

D0< (*d* — *flag*) Double
Return *flag*, which is *true* if and only if the double-precision value *d* is less than zero. "D-zero-less"

D0= (*d* — *flag*) Double
Return *flag*, which is *true* if and only if the double-precision value *d* is equal to zero. "D-zero-equal"

D< (d_1 d_2 — *flag*) Double
Return *flag*, which is *true* if and only if d_1 is less than d_2. "D-less-than"

D= (d_1 d_2 — *flag*) Double
Return *flag*, which is *true* if and only if d_1 is equal to d_2. "D-equals"

DU< (ud_1 ud_2 — *flag*) Double Ext
Return *flag*, which is *true* if and only if ud_1 is less than ud_2. "D-U-less"

FALSE (— *flag*) Core Ext
Return a *flag* that is *false* (binary zero).

NOT (*x* — *flag*) common usage

Identical to **0=**, used for program clarity to reverse the results of a previous test. For example, the following code would test for a value greater than or equal to zero:

```
0< NOT
```

TRUE (— *flag*) Core Ext

Return a *flag* that is *true* (single-cell value with all bits set).

U< (u_1 u_2 — *flag*) Core

Return *flag*, which is *true* if and only if u_1 is less than u_2. "U-less-than"

U> (u_1 u_2 — *flag*) Core Ext

Return *flag*, which is *true* if and only if u_1 is greater than u_2. "U-greater-than"

References Conditionals, Section 2.5.3
MAX and **MIN**, Section 2.2.2
Post-testing loops, Section 2.5.2
Pre-testing loops, Section 2.5.1
String comparisons, Section 2.3.4

2.3 Character and String Operations

Forth contains many words used to reference single characters (bytes) or character strings. Characters may be grouped and thought of as a string; this group is then operated on as a single variable. Character strings are supported by the words documented in this section.

A string may or may not include its length *as part of its structure*. If it does not, it is referred to as a *character string* and is specified by a cell pair (*c-addr u*) representing its starting address *c-addr* and length *u* in characters. If it does include the length, it is referred to as a *counted string* and is specified only by its starting address *c-addr*. The byte at that address contains a binary count of the number of data characters in the string, which immediately follow the count byte. The maximum length of a counted string is 255 data characters (256 bytes total).

A standard working area is used to hold most character strings for processing; this area is referred to as **PAD**.

In addition to the words described in this section, other words may be used to reference character data in specific environments, e.g., database support. Such words are described in the appropriate sections of product-specific manuals.

2.3.1 The PAD—Scratch Storage for Strings

PAD is a storage area of indefinite size (84 characters minimum) that is used to hold strings for intermediate processing. Each terminal task contains a **PAD** area. The word **PAD** places the address of the first byte in this area on the top of the stack.

The contents of the region addressed by **PAD** are under the complete control of the user. No words defined in a Standard Forth system or described in this manual place anything in this region, although changing data space allocations (e.g., by adding new words to the Forth dictionary) may change the address returned by **PAD**.

In cases where **PAD** is located relative to the dictionary pointer, the location of **PAD** changes whenever something is added to the dictionary. Common operations that affect the dictionary pointer may include: adding definitions; adding data or data areas by using **,** (comma), **C,** (c-comma), or **ALLOT**; and discarding definitions. Thus, information left in **PAD** before one of these operations may not be addressable after the operation (and may, in fact, be overwritten by a new definition).

Glossary

HERE *(— addr)* Core

Push the address of the next available memory location onto the stack.

PAD *(— addr)* Core Ext

Return the address of a temporary storage area, usually used for processing strings. The area can hold at least 84 characters. It may be located relative to the top of the dictionary, in which case the address of **PAD** will vary as the dictionary is modified.

References **,** and **C,**, Section 4.3.2
ALLOT, Section 4.3.1

2.3.2 Single-Character Reference Words

The words **C@** and **C!** are used to reference single characters in the same way that **@** and **!** are used to reference cells.

C@ expects an address on top of the stack. This address is replaced with the contents of the addressed byte. This byte will be placed in bits 0–7 of the cell on top of the stack, with the higher order bits set to zero. **C@** does not "sign extend," i.e., it does not propagate the sign bit leftward into more-significant bit positions.

C! expects an address on top of the stack and a character in bits 0–7 of the cell underneath the byte pointer. The high-order bits of this lower cell are ignored. The character is stored in the addressed byte; the address and character cells are removed from the stack.

For example, the following phrase would fetch the first character in **PAD** to the top of the stack:

```
PAD C@
```

References **C@** and **C!**, Section 2.1.3

2.3.3 String Management Operations

Forth contains several words used to reference strings, compare and adjust them, and move strings between different locations. Additional words are used to input or output character strings; these are discussed in Section 3.3.

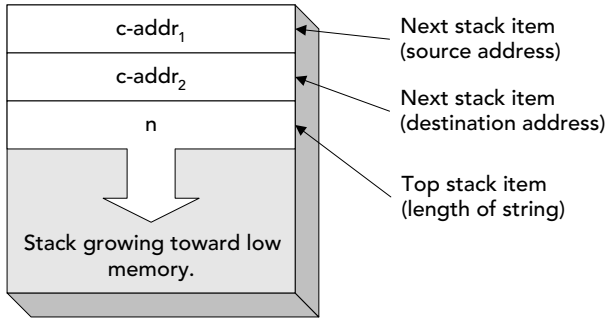

Figure 7. Format of arguments for most two-string operators

Most words that operate on one string expect the length of that string to be on top of the stack, with its address beneath it. Many words that operate on two separate strings expect three items on top of the stack, in the format shown in Figure 7, where one length applies to both strings. The above format is used instead of two separate character counts.

In files, fields containing character strings have names which, when executed, return the address of the field. Thus, such field names may be used to supply arguments for these string operations.

Glossary

-TRAILING *(c-addr u_1 — c-addr u_2)* String

Determine if there are trailing blanks in a string at address *c-addr* whose original length is u_1, and return adjusted string parameters. The same address is returned with an adjusted length u_2, equal to u_1 less the number of spaces at the end of the string. If u_1 is zero, or if the entire string consists of blanks, u_2 is zero. "minus-trailing"

/STRING *(c-addr$_1$ u_1 +n — c-addr$_2$ u_2)* String

Return parameters for a string with the first +*n* characters removed. The original string is at address *c-addr*$_1$ and is of length u_1. The returned string parameters are address *c-addr*$_2$ = *c-addr*$_1$ + n, and length $u_2 = u_1 - n$. "slash-string"

BLANK *(c-addr u —)* Core

Set a region of memory, at address *c-addr* and of length u, to ASCII blanks (hex

20). Two cells are removed from the stack.

ERASE *(c-addr u —)* Core Ext

Erase (set to zero) a region of memory, given its starting address *c-addr* and length *u*:

```
<addr> <count> ERASE
```

Two cells are removed from the stack.

FILL *(c-addr u b —)* Core

Fill a region of memory, at address *c-addr* and of length *u*, with the least-significant byte of the top-of-stack item. Three cells are removed from the stack.

MOVE *($addr_1$ $addr_2$ u —)* Core

Copy *u* bytes from a source starting at $addr_1$ to the destination starting at $addr_2$. After the transfer, the destination area at $addr_2$ contains exactly what the source area $addr_1$ did before the transfer. If the data areas overlap, the original will not be preserved!

CMOVE *($c\text{-}addr_1$ $c\text{-}addr_2$ u —)* String

Copy *u* bytes from a source starting at address $c\text{-}addr_1$ to the destination starting at $c\text{-}addr_2$. The copy proceeds character-by-character from lower to higher addresses. Three cells are removed from the stack. "C-move"

CMOVE> *($c\text{-}addr_1$ $c\text{-}addr_2$ u —)* String

CMOVE> has the same arguments as **CMOVE**, but the copy proceeds character-by-character from *higher* to *lower* addresses. **CMOVE>** is used for transferring from a data field to an overlapping data field in higher memory. Three cells are removed from the stack. "C-move-back"

References Character string I/O, Section 3.3
String processing, Section 4.1.5

2.3.4 Comparing Character Strings

Character-string comparisons operate on two separate character strings; this allows the two to be compared by use of the ASCII collating sequence. The

words in the following glossary are provided.

As an example of their use, you could compare a string whose address is returned by **NAME** with one temporarily stored at **PAD**, testing as follows:

```
PAD <length> DUP NAME SWAP COMPARE
```

SEARCH is generally used to find a short string in a longer string. It is used by the Forth editor.

Glossary

COMPARE (*c-addr*$_1$ u_1 *c-addr*$_2$ u_2 — n) String

Compare the string specified by *c-addr*$_1$ u_1 to the string specified by *c-addr*$_2$ u_2 and return a result code n. The strings are compared character-by-character, beginning at the given addresses, up to the length of the shorter string or until a difference is found. If the two strings are identical and of equal lengths, n is zero. If the two strings are identical up to the length of the shorter string, n is -1 if u_1 is less than u_2, and +1 otherwise. If the two strings are not identical up to the length of the shorter string, n is -1 if the first non-matching character in the string at *c-addr*$_1$ has a lesser numeric value than the corresponding character in the string at *c-addr*$_2$, and +1 otherwise.

SEARCH (*c-addr*$_1$ u_1 *c-addr*$_2$ u_2 — *c-addr*$_3$ u_3 *flag*) String

Search for a match for the string *c-addr*$_2$ u_2 in the string *c-addr*$_1$ u_1 (which is presumed to be longer). If a match is found, return *true* with the address *c-addr*$_3$ of the first matching character and the length u_3 of the remainder of the string. If no match is found, *c-addr*$_3$ = *c-addr*$_1$, $u_3 = u_1$, and *flag* is *false*.

References **PAD**, Section 2.3.1

2.4 NUMERIC OUTPUT WORDS

Numeric output words allow the display of numeric quantities as ASCII characters. This output is generally directed to the terminal.

Numeric output words are divided into two categories: normal output words

and conversion output words. The latter allow the *picturing* of ASCII text, in a manner that resembles COBOL picturing.

All numeric output words produce ASCII text, which is the ASCII number expressed in the current **BASE**. **BASE** is a user variable containing the current conversion radix, and is controlled with the appropriate radix word (e.g., **DECIMAL** or **HEX**) or by setting the value of **BASE** directly. For example, **BASE** may be set to binary by:

```
2 BASE !
```

References Numbers, Section 1.1.6

2.4.1 Standard Numeric Output Words

Several standard words allow displaying single- or double-precision signed numbers in various formats. All of them remove their arguments from the stack. To preserve a number you are about to display, **DUP** it first. Each display word produces an output string that consists of the following characters:

1. If the number is negative, a leading minus sign (hyphen).
2. The absolute value of the number, with leading zeroes suppressed. (The number zero results in a single zero in the output.)
3. In some cases, a trailing blank.

The standard numeric output words are:

Glossary

. (n —) Core

Remove the top of stack item and display it as a signed single-precision integer followed by one space. "dot"

.R (n_1 $+n_2$ —) Core Ext

Display signed single-precision integer n_1 with leading spaces, to fill a field of width $+n_2$, right-justified. This word expects positive integer n_2 on top of the stack to specify the length of the output field. The width of the printed string

that would be output by **.** is used to determine the number of leading blanks. No trailing blanks are printed. If the magnitude of the number to be printed prevents printing within the number of spaces specified, all digits are displayed with no leading spaces in a field as wide as necessary. "dot-R"

? *(a-addr —)* Tools

Display the contents of the address on the stack. "question"

? is equivalent to the phrase: **@ .**

D. *(d —)* Double

Display the top cell pair on the stack as a signed double-precision integer. "D-dot"

D.R *(d +n —)* Double

Display the top cell pair on the stack as a signed double-precision integer in a field of width +*n*, as for **.R**. "D-dot-R"

U. *(u —)* Core

Display the top stack item as an unsigned single-precision integer followed by one space. "U-dot"

U.R *(u +n —)* Core Ext

Similar to **.R**, but unsigned. Display the unsigned single-precision integer *u* with leading spaces to fill a field of width +*n*, right-justified. "U-dot-R"

2.4.2 Pictured Number Conversion

Forth contains words that allow numeric quantities to be displayed through use of a *pictured format* control. These words allow specification of field sizes, embedded punctuation, etc.

In Forth, the description of the desired output format starts with the rightmost character and continues to the left. Although this is the reverse of the method apparently used in other languages, it is the *actual* conversion process in all languages.

These words are used to convert numbers on the stack into ASCII character strings formatted according to the picture specifications. These strings are

built in a temporary area in memory, which is large enough to accommodate at least 66 characters of output (32-bit CPUs) or 34 characters (16-bit CPUs). After the picture conversion, the address of the beginning of the string, and the count of the number of characters in it, are passed to the user. At this point, the converted string can be printed at the terminal with **TYPE** or can be used in some other way.

The standard numeric output words (see previous section) also use this temporary region in the user's partition. As a result, these words may not be executed while a pictured output conversion is in process (e.g., during debugging). Furthermore, the user may not make new definitions during the pictured conversion process, since this may move the area in which the string is being generated.

References Standard numeric output, Section 2.4.1
TYPE, Section 3.3.3

2.4.2.1 Using Pictured Numeric Output Words

These words provide control over the conversion of binary numbers into digits. This section describes only pictured words which result in numeric output (digits); the following sections describe output of non-numeric punctuation, such as periods and commas. Throughout the number-conversion process, the number being operated on remains on the stack, where it is repeatedly divided by **BASE** as digits are converted; it is finally discarded by **#>** at the end of the process.

As an example of the use of these words, consider a definition of the standard Forth word **.** ("dot"):

```
: . ( n -- )   DUP ABS 0  <# #S ROT SIGN #>  TYPE SPACE ;
```

DUP ABS leaves two numbers on the stack: the absolute value of the number is on top of the original number, which is now useful only for its sign. **0** adds a cell on top of the stack, so that the **0** cell and the **ABS** cell form the required double-precision integer to be used by the **<#** … **#>** conversion routines. **<#** initializes the conversion process; then **#S** and **SIGN** assemble the string. **#>** completes the conversion and leaves the address and count of the ASCII string on the stack, suitable as input to **TYPE**.

To print a signed double-precision integer with the low-order three digits always appearing, regardless of the value, you could use the following definition:

```
: NNN ( d -- )   SWAP OVER DABS  <# # # #S
   ROT SIGN  #>  TYPE SPACE ;
```

The **SWAP OVER DABS** phrase establishes the signed value beneath the absolute value of the number to be printed, for the word **SIGN**. The sequence **# #** converts the low-order two digits, regardless of value. The word **#S** converts the remaining digits and always results in at least one character of output, even if the value is zero.

From the time when the initialization word **<#** executes until the terminating word **#>** executes, the number being converted remains on the stack. It is possible to use the stack for intermediate results during pictured processing but any item placed on the stack must be removed before any subsequent picture editing or fill characters may be processed.

Glossary

<# (*ud — ud*) *or* (*n ud — n ud*) Core

Initialize pictured output of an unsigned double-precision integer. If the output is to be signed, a signed value *n* must be preserved somewhere, typically immediately beneath this integer, where it may later be passed to **SIGN** (below). "bracket-number"

(ud_1 — ud_2) Core

Divide ud_1 by **BASE**, giving the quotient ud_2 and the remainder *n*. Convert *n* to an ASCII character and append it to the beginning of the existing output string. Must be used after **<#** and before **#>**. The first digit added is the lowest-order digit (units), the next digit is the **BASE** digit, etc. Each time **#** is used, a character is generated, even if the number to be converted is zero. "number-sign"

#S (ud_1 — ud_2) Core

Convert digits from ud_1 repetitively until all significant digits in the source item have been converted, at which point conversion is completed, leaving ud_2 (which is zero). Must be used after **<#** and before **#>**. **#S** always results in at least one output character, even if the number to be converted is zero. "number-sign-S"

SIGN *(n —)* Core

Insert a minus sign at the current position in the string being converted if the signed value *n* is negative. This signed value *n* is a single-precision number; if the high-order bit is set, a minus sign will be introduced into the output as the leftmost non-blank character. The magnitude of the signed value is irrelevant. In order for the sign to appear at the left of the number (the usual place), **SIGN** must be called after all digits have been converted.

#> *(ud — c-addr u)* Core

Complete the conversion process after all digits have been converted. Discard the (presumably) exhausted double-precision number, and push onto the stack the address of the output string, with the count of bytes in this string above it. "number-bracket"

References **TYPE**, Section 3.3.3

2.4.2.2 Using Pictured Fill Characters

In addition to pictured numeric output, it is possible to introduce arbitrary fill characters (or punctuation) into the output string at any point through the use of **HOLD**. **HOLD** requires as a parameter the numeric value of the ASCII character to be inserted. Thus,

```
2F HOLD
```
(value given in hex) or

```
[CHAR] / HOLD
```
(value computed by **[CHAR]** from the ASCII character following)

inserts the character **/** into the output string at the point where **HOLD** is executed. The phrase `<value>` **HOLD** may be executed as many times as desired in a given output conversion sequence.

If fill characters are likely to be used in several definitions, you may wish to add specific commands for them. The following format may be used for such a definition:

```
: '<name>'   <char-value> HOLD ;
```

where *char-value* is the ASCII value of the character in the current radix and

'*name***'** is the name of the word to be defined. (There are no restrictions on the format of name, **'***name***'** is merely an often-used convention that includes the specified character in the name itself.) **HOLD** is defined in such a way that executing **'***name***'** during pictured editing causes the indicated fill character to be introduced into the output.

In the following example, **'.'** produces a decimal point at the current position in the pictured numeric output. Then the word **.$** is defined to print double-precision integers as signed amounts with two decimal places:

```
: '.'   [CHAR] . HOLD ;
: .$ ( d)   SWAP OVER DABS  <# # # '.'
   #S ROT SIGN #> TYPE  SPACE ;
```

The word **[CHAR]** is used in definitions. At run time, it places on the stack the ASCII value of the first character in the word following it. **CHAR** is similar, but is used interpretively (i.e., not in definitions).

Glossary

CHAR *(— char)* Core

Parse the word following **CHAR** in the input stream. Put the ASCII value of the first character of this word on the stack. **CHAR** is normally used interpretively; see **[CHAR]** for the equivalent function inside a definition. "care"

HOLD *(char —)* Core

Append *char* to the current beginning of the pictured numeric output string. **HOLD** must occur only inside a **<#** … **#>** number conversion sequence.

[CHAR] *(— char)* Core

At compile time, parse the word following **[CHAR]** in the input stream. At run time, put the ASCII value of the first character of this word on the stack. "bracket-care"

2.4.2.3 Processing Special Characters

The normal pictured output capabilities described in the preceding two sections can handle most output requirements. But special cases, such as introducing commas in a number or floating of a character (e.g., **$**), require special processing.

In order to perform certain of these operations, it is necessary to refer to the unconverted portion of a number being printed. This unconverted portion is equivalent to the original number divided by the current radix, for each numeric digit already generated. For example, if the initial number is 123 and the radix is 10, the intermediate number is 12 (following the conversion of the first digit) and 1 (following conversion of the second digit).

The value of this number may be tested, and logical decisions may be made based upon its value. To illustrate, consider the following block of source code. The word **D.ENG** prints a double-precision integer in U.S. engineering format (i.e., a comma after every three decimal places):

```
 0 ( Number Formats)   DECIMAL   VARIABLE #PLACES
 1 : ','   44 HOLD ;
 2 : (D.ENG) ( d)   SWAP OVER DABS  <#  BEGIN
 3    #  1 #PLACES +!  2DUP D0= NOT WHILE
 4    #PLACES 3 MOD NOT IF ','  THEN   REPEAT
 5   ROT SIGN #> ;
 6 : D.ENG ( d)   (D.ENG)  TYPE  SPACE ;
 7
 8
 9
10
11
12
13
14
15
```

Using techniques similar to those above, you can do any kind of numeric output editing in Forth.

References Block-based disk access, Section 3.4

2.5 PROGRAM STRUCTURES

This section describes a set of Forth words used to establish program loops and to alter the normal, sequential execution of words. Similar words for use in **CODE** definitions are defined in the **ASSEMBLER** word list.

Logic control words must be used within a definition. They will not operate properly when typed from a keyboard, because the text interpreter (which is sequentially processing the input stream) has no way to tell where a forward branch is to terminate. Loops must be opened and closed within the same definition. Loops may be nested to any depth.

Some words in this section are called *compiler directives*. When the compiler sees most words, it compiles references to the words' run-time behavior. But when the compiler sees a compiler directive, it executes it immediately, rather than compiling it. Forth is extensible, so you may define your own compiler directives. Specific techniques appear in the section referenced below.

References Compiler directives, Section 4.4

2.5.1 Indefinite Loops

The simplest looping method available in Forth is the **BEGIN** … **AGAIN** loop. This loop endlessly repeats the code that is between the **BEGIN** and **AGAIN**. **BEGIN** … **AGAIN** loops are used for control activities which are not expected to stop. These commonly are used to define the power-up behavior of an embedded system, or a loop that will only terminate if an error condition causes a **THROW**. Examples of such applications include process-control loops and computer-sequenced machinery. **BEGIN** … **AGAIN** is also used in **QUIT**, the highest-level word of an interactive Forth system. Loops with no exit can only be used at the highest level in a program.

An example of the outermost loop in a program to control an industrial process might be:

```
: REACTION   CONTROLS CLEAR
   BEGIN  DATA  ERROR  CORRECT  AGAIN ;
```

This process-control loop clears the controls, then enters an infinite loop which continuously collects data, calculates an error quantity, and applies a correction function. Usually, such a program is run asynchronously by a background task, and the operator stops it with a word built from task-control words.

BEGIN does not actually compile anything, it simply pushes the address of the

next available dictionary location on the stack at compile time. Thus, it marks a location for use by a subsequent compiler directive's operation.

BEGIN and **UNTIL** allow the user to set up a loop which may be executed repetitively, in a manner similar to **BEGIN** … **AGAIN** loops except that a test is performed before the loop repeats.

The form of a **BEGIN** … **UNTIL** construct is:

```
BEGIN  <words to execute repeatedly>  <test value>  UNTIL
```

When execution reaches the word **UNTIL**, the test value on top of the stack is examined and removed from the stack. If this value is *false* (zero), execution returns to the word that follows **BEGIN**; if the value is *true* (non-zero), execution continues with the word that follows **UNTIL**.

BEGIN loops can be nested. However, a loop of any type must be nested entirely within any outer loop. There is no way to branch directly out of a control structure. For example,

```
BEGIN <words>
   BEGIN <words>
   UNTIL <words>
UNTIL <words>
```

BEGIN … **UNTIL** may only be used within a definition; it may not be executed interpretively from a terminal.

The **ASSEMBLER** word list also contains words named **BEGIN** and **UNTIL**; these words function similarly to their equivalents, but with differences related to the context in which they occur.

Pre-testing indefinite loops are similar to **BEGIN** … **UNTIL** loops, except the test to leave the loop is performed *before* the end of the loop code. The syntax of the Forth pre-testing loop is:

```
BEGIN  <executed every iteration>  <test> WHILE
   <not executed on the last iteration> REPEAT
```

There may be no code before **WHILE** except the test, in which case the loop code may not execute at all. **WHILE** removes the top number from the stack and tests it, then leaves the loop if the value is *false* (zero), skipping the words

between **WHILE** and **REPEAT**. If the value on the stack is *true* (non-zero), **WHILE** continues to the next word in the loop. When the program execution reaches **REPEAT**, it branches unconditionally back to the words immediately after **BEGIN** and repeats the loop.

For an example, consider a word that counts fruit in a mechanical sorter:

```
: GOOD ( -- n )   0  BEGIN
   FETCH FRUIT ?GOOD
   WHILE  1+  REPEAT ;
```

As long as the machine sees good fruit in the test cell, the loop continues and the machine considers the next fruit. When the test fails, the fruit remains in the test cell, to be evaluated by some process other than the word **?GOOD**.

In situations when both are equally convenient, the **BEGIN** … **UNTIL** loop is faster and requires fewer bytes, and thus is preferable to the **BEGIN** … **WHILE** … **REPEAT** loop.

Glossary

AGAIN *(—)* Core Ext

At compile time, compile an unconditional backward branch to the location on the control-flow stack (usually left there by **BEGIN**; see Section 4.4.2). At run time, execute the branch.

BEGIN *(—)* Core

At compile time, save the next available dictionary location on the control-flow stack. This action marks the destination of a backward branch for use by other compiler directives. At run time, simply continue execution.

REPEAT *(—)* Core

At compile time, resolve two branches, usually set up by **BEGIN** and **WHILE**. In the most common usage, **BEGIN** leaves a destination on the control-flow stack and **WHILE** places an origin under **BEGIN**'s destination. Then **REPEAT** compiles an unconditional backward branch to the destination location following **BEGIN** and provides the location following **REPEAT** as the destination address for the forward conditional branch originated by **WHILE**.

At run time, execute the unconditional backward branch to the location fol-

lowing **BEGIN**.

UNTIL (*x* —) Core

At compile time, compile a conditional backward branch to the location on the control-flow stack (usually left there by **BEGIN**). At run time, if *x* is zero, take the backwards branch; otherwise, continue execution beyond the **UNTIL**.

WHILE (*x* —) Core

At compile time, place a new unresolved forward reference origin on the control stack, under the topmost item (which is usually a destination left by **BEGIN**). At run time, if *x* is zero, take the forward branch to the destination that will have been supplied (e.g., by **REPEAT**) to resolve **WHILE**'s origin; otherwise, continue execution beyond the **WHILE**.

References **BEGIN** … **UNTIL** for the Assembler word list, Section 5.8
Logic operations, Section 2.2.2
Control-flow stack, Section 4.4.2

2.5.2 Counting (Finite) Loops

Forth provides words to allow looping in a manner similar to most other high-level languages. The words associated with counting loops are given in the glossary at the end of this section.

The possible forms of a counting loop in Forth are as follows:

```
     <limit> <initial> DO <words to repeat> LOOP
or   <limit> <initial> DO <words to repeat> <value> +LOOP
```

A **DO** … **LOOP** increments by one and always runs in the positive direction. A **DO** … **+LOOP** increments by the given integer *value*, which may be positive or negative.

To illustrate the use of loops, the word **SUM** is defined to sum the values of the integers 1 to 100 and to leave the result on the stack:

```
: SUM   0  101 1 DO  I +  LOOP ;
```

The limit value is specified as 101, not 100, because the loop index is incre-

mented before the termination test, and the loop will terminate when the index is equal to the limit. The word **I** returns the current loop index on the stack.

Loops may be nested to any depth, limited only by the capacity of the return stack. At each point in a nested loop, the word **I** returns the index of the innermost active loop, and the word **J** returns the index of the next outer loop.

+LOOP allows descending index values to be used. When an index value is descending, however, the loop is terminated when the limit is passed (not merely reached). When the index value is ascending (i.e., the increment value specified for **+LOOP** is positive), the loop terminates when the index value is reached, as for **LOOP**.

To illustrate the use of **+LOOP** with descending index values, the following definition is equivalent to the first definition of **SUM**:

```
: SUM   0  1 100 DO  I +  -1 +LOOP ;
```

Here the initial value of the index is 100 and the final value is 1.

Loop parameters usually are kept on the return stack (see glossary entry for **DO**, below), and are not affected by structures other than **DO** … **LOOP**.

Because loop parameters are checked at the end of the loop, any loop will always be executed at least once, regardless of the initial values of the parameters. Because a **DO** loop with equal input parameters will execute not once but a very large number of times—equal to the largest possible single-cell unsigned number—the word **?DO** should be used in preference to **DO** if the loop parameters are being calculated and might be equal (e.g., both zero).

Glossary

DO (n_1 n_2 —) Core

Establish the loop parameters. This word expects the initial loop index n_2 on top of the stack, with the limit value n_1 beneath it. These values are removed from the stack and stored elsewhere, usually on the return stack, when **DO** is executed.

?DO (n_1 n_2 —) Core Ext

Like **DO**, but check whether the limit value and initial loop index are equal. If they are, continue execution after the **LOOP**; otherwise, set up the loop values

and continue execution immediately following **?DO**. This word should be used in preference to **DO** whenever the parameters may be equal. "question-do"

LOOP *(—)* Core

Increment the index value by one and compare it with the limit value. If the index value is equal to the limit value, the loop is terminated, the parameters are discarded, and execution resumes with the next word. Otherwise, control returns to the word that follows the **DO** or **?DO** that opened the loop.

+LOOP *(n —)* Core

Like **LOOP**, but increment the index by the specified signed value *n*. After incrementing, if the index crossed the boundary between the loop limit minus one and the loop limit, the loop is terminated as with **LOOP**. "plus-loop"

I *(— n)* Core

Push a copy of the current value of the index onto the data stack. This word may only be used for this purpose within the definition that opened the loop, not in definitions the loop invokes, because nested colon definitions may cause a return address to be put on the stack on top of the loop index. If the code in the body of the loop places any values explicitly on the return stack, they must be removed before **I** is executed; otherwise, an erroneous index value may result. On many systems, **I** is identical to **R@**, but this may not be relied on because some systems calculate **I** from other values kept on the return stack.

J *(— n)* Core

Push a copy of the next-outer loop index onto the data stack. When two **DO** … **LOOP**s are nested, this obtains the value of the outer index from inside the inner loop. On many systems, **J** is kept directly on the return stack; but in others, **J** is a calculated value, so you should not attempt to obtain the outer loop index except by using **J**.

LEAVE *(—)* Core

Discard loop parameters and continue execution immediately following the innermost **LOOP** or **+LOOP** containing this **LEAVE**.

UNLOOP *(—)* Core

Discard the loop parameters for the current nesting level. This word is not needed when a **DO** … **LOOP** completes normally, but it is required before leaving a definition by calling **EXIT**. One **UNLOOP** call for each level of loop nesting is required before leaving a definition.

References **EXIT** and un-nesting definitions, Section 2.5.5
Control-flow stack, Section 4.4.2

2.5.3 Conditionals

These words allow conditional execution of words within a single definition. They may only appear within a definition and may not be used in interpretive text or in text executed by direct entry from a terminal. There are similar conditional words that can be used interpretively (see Section 4.1.6).

The general usage of these words is:

```
    <test value> IF <true clause> ELSE <false clause> THEN
or  <test value> IF <true clause> THEN
```

When **IF** is executed, the item on top of the stack is removed and examined. If *test value* is *true* (non-zero), execution continues with the words after **IF** (the *true clause*). If *test value* is *false* (zero), execution resumes with the words after **ELSE** (the *false clause*) or, if **ELSE** is not present, with the words after **THEN**.

Execution of the *true clause* terminates with the word **ELSE**, if present, and resumes with the word after **THEN**.

Both the *true clause* and the *false clause* may be any group of previously defined Forth words. Either clause may contain **DO** … **LOOP**s, **BEGIN** … **UNTIL** loops, and/or other **IF** … **ELSE** … **THEN** structures, so long as the entire structure is contained within the clause. Similarly, one **IF** structure may be nested inside another structure of any kind, so long as the **THEN** that terminates the structure appears within the same clause.

Glossary

ELSE *(—)* Core

At compile time, originate the true clause branch and resolve the false clause branch. It is assumed that there is a branch origin on the control stack, usually left there by **IF**. Provide the location following **ELSE** as the destination address for the forward conditional branch originated by **IF**. Place a new forward reference origin on the control stack, marking the beginning of an unconditional

branch at the end of the true clause (this will later be resolved by **THEN**). At run time, execute the unconditional branch to skip over the false clause.

IF (*x* —) Core

At compile time, place a forward reference origin on the control stack, marking the beginning of a conditional branch. At run time, if *x* is zero take the forward branch to the destination that will have been supplied (e.g., by **ELSE** or **THEN**); otherwise, continue execution beyond the **IF**.

THEN (—) Core

At compile time, provide the location beyond **THEN** as the destination address for the forward branch origin found on the control stack. This origin will normally have been placed there either by **ELSE** (if there was a *false* clause) or by **IF** (if there was no *false* clause). At run time, simply continue execution.

References Logic operations, Section 2.2.2
Control-flow stack, Section 4.4.2
Text interpreter directives, Section 4.1.6

2.5.4 CASE Statement

A high-level **CASE** statement structure is available for situations in which an input condition needs to be checked against more than one or two possible values. The usual syntax is:

```
CASE
   <x1> OF < x1 action> ENDOF
   <x2> OF < x2 action> ENDOF
   …
   <default action> ENDCASE
```

The structure begins with the word **CASE**. When it executes, a case selector *x* must be on the stack. A series of **OF** … **ENDOF** clauses follow, each **OF** being preceded by a test value on the stack (x_1, x_2, etc.). The case selector is compared against the test values in order. If it matches one, the corresponding code between that **OF** and **ENDOF** is executed, and execution branches beyond the **ENDCASE**. If the case selector does not match any of the test values, it remains on the stack after the last **ENDOF**, and some default action may be

taken. Any action should preserve the stack depth (use **DUP** if necessary), because **ENDCASE** performs a **DROP** (presumably on the case selector) and then continues execution beyond **ENDCASE**.

This structure is flexible, and is more readable than nested **IF** statements if there are more than two or so comparisons. **CASE** statements may be nested; there may be any number of **OF** … **ENDOF** pairs; and there may be any amount of logic inside an **OF** … **ENDOF** clause, including computation of the next test value.

Implementation details of a **CASE** statement are system dependent—especially with regard to the method of handling the multiple branches—but in all cases, the logic conforms to the above description.

Glossary

CASE (—) Core Ext

At compile time, mark the start of a **CASE** … **OF** … **ENDOF** … **ENDCASE** structure. At run time, continue execution.

ENDCASE (x —) Core Ext

At compile time, resolve all branches in the case structure, such that the location beyond **ENDCASE** becomes the destination for all branches originated by occurrences of **ENDOF**. At run time, discard the top stack value x (presumably the case selector) and continue execution.

ENDOF (—) Core Ext

At compile time, provide the location following **ENDOF** as the destination address for the forward conditional branch originated by the corresponding **OF**. Place a new forward reference origin on the control stack, marking the beginning of an unconditional branch (this will later be resolved either by the next **ENDOF** or by **ENDCASE**). Perform other system-dependent functions which may involve resolution of other clause origins. At run time, execute the unconditional branch.

OF (x_1 x_2 — \| x_1) Core Ext

At compile time, place a forward reference origin on the control stack, marking the beginning of a conditional branch. At run time, if test value x_2 is not equal to case selector x_1, discard x_2 and take the forward conditional branch to the destination that will have been supplied by **ENDOF**; otherwise, discard both values and continue execution beyond the **OF**.

References Logic operations, Section 2.2.2
Control-flow stack, Section 4.4.2

2.5.5 Un-nesting Definitions

When a high-level definition calls another, it is said to *nest* the calls, because the return will normally be to the next location in the calling definition. The called definition *un-nests* when it is finished executing, to effect this return.

EXIT is a function that causes un-nesting to occur. **EXIT** may be used to leave a definition at any point. In indirect-threaded implementations, **EXIT** is compiled by **;** at the end of every **:** definition, but may also be called directly. **EXIT** leaves the current definition, and resumes execution of the next word in the definition which called the word containing **EXIT**. Because many implementations use the return stack to control nesting, it must be clear of any temporarily stored data before an **EXIT** can be performed; for example, loop parameters must be discarded by **UNLOOP** if **EXIT** is executed within a **DO** … **LOOP**.

A trivial example of **EXIT** is:

```
: TEST ( n)   1 .  IF EXIT THEN  2 . ;

0 TEST 1 2
1 TEST 1
```

Frequently, words containing **EXIT** will have different stack results, depending on whether the word **EXIT**s or not. The standard stack notation for such a situation is:

```
( input-arguments -- EXIT-case | normal-case)
```

EXIT is the only Forth word which permits unstructured programs (modules with multiple exit points). Because unstructured techniques tend to impair code's readability and maintainability, they should be used sparingly—only when the overall effect is to simplify the code. It is considered bad form to use **EXIT** more than once in a word; if you believe you need to do so, try factoring that word into several words.

Glossary

EXIT *(—); (R: nest-sys —)* Core

Return control immediately to the calling definition specified by *nest-sys*. Before executing **EXIT**, a program must remove any items explicitly stored on the return stack. If **EXIT** is called within a **DO** … **LOOP**, **UNLOOP** must be executed first to discard the loop-control parameters.

References Interpreter pointer, Section 1.1.7
LOAD, Section 3.4.3.1
Text interpreter, Section 1.1.5
UNLOOP, Section 2.5.2

2.5.6 Vectored Execution

Although normal Forth usage (as well as good programming practice) emphasizes the *structured programming* modes of sequential, iterative, and conditional execution, it is sometimes desirable to direct Forth to execute a specific function in response to some external stimulus. This technique may be used, for example, by a report that searches a database, selecting records according to a criterion which may need to vary; by a bank of push-buttons, each of which is attached to a particular Forth word; or by a routine that computes the address of a function to be executed.

The word **EXECUTE** expects an *execution token* on the stack—a value, usually an address, that points to the execution behavior of a definition. **EXECUTE** removes the token from the stack and uses it to cause the given definition to execute.

For example:

```
VARIABLE NUMERAL
: T1   1 . ;
: T2   2 . ;
: ONE    ['] T1  NUMERAL ! ;
: TWO    ['] T2  NUMERAL ! ;
: N   NUMERAL @ EXECUTE ;
```

If the user types:

```
ONE N
```

the system will display `1`. Typing:

```
TWO N
```

will produce `2`.

The stack effect of all members of a set of words to be **EXECUTE**d in a particular context must be the same. That is, they must all require and leave the same number of items on the stack.

The word **DEFER** provides a convenient means of managing a single execution vector. The syntax is:

```
DEFER <name>
```

This creates a dictionary entry for *name* and makes it an *execution variable*. *name* is similar to a variable, but specifically contains the execution token of another word; the other word is executed when *name* is executed. The execution token of the other word to be executed is stored into the data area of *name* by the word **IS**. If *name* is executed before it has been initialized by **IS**, an error will occur.

DEFER lets you change the execution of previously defined commands by creating a slot which can be loaded with different behaviors at different times. The preceding example would be defined this way using **DEFER**:

```
DEFER NUMERAL
: T1   1 . ;
: T2   2 . ;
: ONE   ['] T1  IS NUMERAL ;
: TWO   ['] T2  IS NUMERAL ;
```

Then, typing:

```
ONE NUMERAL
```

displays `1`, and

```
TWO NUMERAL
```

displays 2.

Most uses of **EXECUTE** are for implementing a variable function, as described in the previous sections. The ability to generate and manage a table of execution addresses is also extremely useful for such purposes as managing a function-button pad, a function menu on a graphics tablet, etc. The following example will outline a simple button-response application which may serve as a model for similar situations.

Let us assume that the word **BUTTON** has been defined to wait until a button is pressed and then to return the button number (0–15) of the button (the actual definition of **BUTTON** would depend on the computer and interface). Now consider the following:

```
VARIABLE BUTTONS   15 CELLS ALLOT
: IGNORE ;
' IGNORE BUTTONS !  BUTTONS DUP CELL+
   14 CELLS CMOVE
```

The above lines create a table with one cell for each button, and initialize all positions to contain the address of an empty definition (which effectively "ignores" an undefined button). The move and replication of the **IGNORE** address must be done with a **CMOVE** instead of a **MOVE**, because only **CMOVE** is guaranteed to move one cell or less at a time, achieving the replication of the address.

Now we will define special versions of **:** (colon) and **;** (semicolon) that will not only create an ordinary definition but also store its execution token into a specified cell of **BUTTONS**. To do this, we use the word **:NONAME**, which returns the execution token of the current definition on the stack. Because **:NONAME** does not create a dictionary entry, we need to make an entry explicitly with **CREATE**, and we need to store the execution token into this word's parameter field (done by the second **!** in the definition of **;B** below).

```
: :B ( n)   CREATE  HERE  0 ,  :NONAME  DOES> @ EXECUTE ;
: ;B   POSTPONE ;  ROT OVER SWAP CELLS BUTTONS + !
    SWAP ! ;  IMMEDIATE
```

Now we can create definitions which are attached to certain buttons by using

:B with the button number as a parameter, and concluding the definition with **;B**. Each such definition will have a name, to allow it to be tested independently of the button pad. For example,

```
0 :B ESCAPE   1 ABORT" ?" ;B
```

defines Button 0 to be an "escape" button. All that remains is to define a routine to monitor the button pad and to handle responses:

```
: MONITOR   BEGIN BUTTON BUTTONS
   + @ EXECUTE  AGAIN ;
```

Typing **MONITOR** will place the terminal task in an infinite loop that responds to buttons. Button 0 will cause an abort and return control to the terminal.

In practice, **MONITOR** may very likely be executed by a background task (on systems that support multitasking). In this case, you may need techniques other than **ABORT"** (which requires a terminal) for halting.

Glossary

DEFER <name> *(—)* common usage

Define *name* to be an execution variable. When *name* is executed, the execution token stored in *name*'s data area will be retrieved and the behavior associated with that token will be performed. An error will occur if *name* is executed before it has been initialized by **IS**.

EXECUTE *(i*x xt — j*x)* Core

Remove execution token *xt* from the stack and perform the execution behavior it identifies. Other stack effects are due to the word that is **EXECUTE**d.

IS <name> *(xt —)* common usage

Store *xt* in the data area of the execution variable *name*.

References

['], Section 4.3.6
:NONAME definitions, Section 4.2.4
ABORT", Section 2.6
Custom defining words and **DOES>**, Section 4.2.6.2
IMMEDIATE and **POSTPONE**, Section 4.4.1
TO used with **VALUE**s, Section 4.2.3

2.6 EXCEPTION HANDLING

Forth provides several methods for error handling. **ABORT** and **ABORT"** may be used to detect errors. However, they are relatively inflexible, in that they unconditionally terminate program execution and return to the idle state.

The words **CATCH** and **THROW**, discussed in this section, provide a method for propagating error handling to any desired level in an application program. **THROW** may be thought of as a multi-level **EXIT** from a definition, with **CATCH** marking the location to which the **THROW** returns.

Suppose that, at some point, word A calls word B, whose execution may cause an error to occur. Instead of just executing word B's name, word A calls word B using the word **CATCH**. Someplace in word B's definition (or in words that B's definition may call) there is at least one instance of the word **THROW**, which is executed if an error occurs, leaving a numerical *throw code* identifier on the stack. After word B has executed and program execution returns to word A just beyond the **CATCH**, the throw code is available on the stack to assist word A in resolving the error. If the **THROW** was not executed, the top stack item after the **CATCH** is zero.

When **CATCH** executes, it requires the execution token of the lower-level routine it is to call to be on top of the stack:

```
… ['] <routine name> CATCH …
```

is the typical syntax. At the time **CATCH** executes, there may be other items on the data stack.

After the lower-level routine has executed and control has returned to the **CATCH**ing routine, the data stack will have one of two behaviors. If the lower-level routine (and any words it called) did not cause a **THROW** to execute, the top stack item after the **CATCH** will be zero and the remainder of the data stack may be different than it was before, changed by the behavior of the lower-level routine. If a **THROW** did occur, the top stack item after the **CATCH** will contain the throw code, and the remainder of the data stack will be restored to the same *depth* (although *not necessarily to the same data*) it had just before the **CATCH**. The return stack will also be restored to the depth it had before the **CATCH**.

When **THROW** executes, it requires a throw code on top of the stack. If this code

is zero, **THROW** does nothing except to remove the zero from the stack; the remainder of the stack is unchanged. If the throw code is non-zero, **THROW** returns the code on top of the stack, restores the data stack *depth* (but not necessarily the data) to its value when **CATCH** was executed, restores the return stack depth, and passes control to the **CATCH**ing routine. If a non-zero **THROW** occurs without a corresponding application program **CATCH** to return to, it is treated as an **ABORT**.

The set of information (e.g., stack depths) that may be needed for restoration is called an *exception frame*. Exception frames are placed on an *exception stack* in order to allow nesting of **CATCH**es and **THROW**s. Each use of **CATCH** pushes an exception frame onto the exception stack. If execution proceeds normally, **CATCH** pops the frame; if an error occurs, **THROW** pops the frame and uses its information for restoration.

An example of **CATCH** and **THROW** taken from Standard Forth is:

```
: COULD-FAIL ( -- c)   KEY DUP  [CHAR] Q = IF
   1 THROW  THEN ;

: DO-IT ( n n -- c)   2DROP  COULD-FAIL ;

: TRY-IT ( -- )   1 2  ['] DO-IT CATCH  IF
   2DROP ." There was an exception"  CR
   ELSE ." The character was " EMIT CR  THEN ;
```

The upper-level word **TRY-IT** calls the high-risk operation **DO-IT** (which in turn calls **COULD-FAIL**) using **CATCH**. Following the **CATCH**, the data stack contains either the character returned by **KEY** and a zero on top, or two otherwise-undefined items (to restore it to the depth before the **CATCH**) and a one on top. Since any non-zero value is interpreted as *true*, the returned throw code is suitable for direct input to the **IF** clause in **TRY-IT**.

Standard Forth reserves negative throw codes for system implementors, and positive throw codes for applications. Throw codes -1 through -255 are reserved for assignment by the Standard itself, and are used to specify common types of errors (such as divide by zero, invalid address, stack underflow/overflow, etc.) in order that different Forth implementations will have compatible behaviors in these common cases.

Frequently, when a terminal task aborts, it is desirable to display a message, clear stacks, and re-enter a default state awaiting user commands. This is the primary use of **ABORT"**.

Glossary

ABORT *(i*x —); (R: j*x —)* Core, Exception Ext

Unconditionally terminate execution, empty both stacks, and return to the task's idle behavior (usually **QUIT**—see Section 4.1.2). No message is issued. May be executed by any task in a multitasked implementation.

ABORT" `<text>"` *(i*x flag —); (R: j*x —)* Core, Exception Ext

If *flag* is *true* (non-zero), type the specified *text* at the user's terminal, clear both stacks, and return to the task's idle behavior. Must be used inside a definition. "abort-quote"

The definition of **ABORT"** concludes with the word **ABORT** (or otherwise includes its functionality).

CATCH *(i*x xt — j*x 0 | i*x n)* Exception

Save information about the depth of the data and return stacks in an exception frame and push the frame on the exception stack. Execute the execution token *xt* (as with **EXECUTE**). If the execution of *xt* completes normally (i.e., a non-zero **THROW** is not executed), pop the exception frame and return zero on top of the data stack, above whatever stack items were returned by *xt* **EXECUTE**, and delete the stack-depth information. Otherwise, see the definition of **THROW** for completion of the exception-processing behavior.

THROW *(k*x n — k*x | i*x n)* Exception

If *n* is zero, simply remove *n* from the data stack. If *n* is non-zero, pop the top-most frame from the exception stack, restore the input source specification that was in use before the corresponding **CATCH**, and adjust the depths of all stacks so they are the same as the depths saved in the exception frame (the value of *i* in **THROW**'s stack comments is the same as the value of *i* in **CATCH**'s comments). Place *n* on top of the data stack and transfer control to a point just beyond the corresponding **CATCH** that pushed the exception frame.

References **EXECUTE**, Section 2.5.6

3. SYSTEM FUNCTIONS

Forth is more than a programming language. The earliest versions of Forth ran standalone on primitive minicomputers and microprocessors in the 1970s, providing an integrated system, language, and application functions in a single package. This heritage persists in the Forth *virtual machine*, even though today it is frequently implemented on top of a conventional operating system.

This section describes words used to load, organize, and manage Forth applications, as well as standard system devices (disk, terminal, and clock). But before considering details of various Forth system functions, the next section will present a fundamental concept commonly used to implement system functions.

3.1 VECTORED ROUTINES

It is often desirable to modify or re-direct system functions, because of changing hardware or application requirements, without recompiling the system kernel. Forth facilitates this by providing *execution vectors* containing the addresses of the current versions of these functions. There are two groups of vectored routines: system-wide functions and terminal-dependent functions (i.e., those whose behavior differs between different kinds of CRT or between keyboard/display and printer). For each vectored function, there are at least three Forth words: the function itself (which performs a **`@ EXECUTE`** on the vector), the vector itself, and at least one routine to be executed.

Table 5 summarizes the vectored routines controlled on a system-wide basis on some Forth systems.

Table 5: Common system-level vectored routines

Function	Vector	Primitive	Description*
BLOCK	**'BLOCK**	**(BLOCK)**	Returns the address of a specified block.
BUFFER	**'BUFFER**	**(BUFFER)**	Returns the address of an available buffer, identified as containing a specified block.
CREATE	**'CREATE**	**(CREATE)**	Creates a dictionary entry.
NUMBER	**'NUMBER**	**(NUMBER)**	Converts a string at a given address to binary on the stack.

* See references, below, for details.

In addition, some routines in multitasking systems are vectored through user variables for differing task-specific functions. Typically, these control CRTs, printers, and other serial devices, or application functions such as databases. Refer to your product documentation for specific details of vectored system functions on your implementation.

References

BLOCK and **BUFFER**, Section 3.4
CREATE, Section 4.2.1
Support of special terminal functions, Section 3.3.3
TYPE, Section 3.3.2
Vectored execution, Section 2.5

3.2 System Environment

Standard Forth systems provide a mechanism for inquiring about the configuration and parameters of a particular system, either interactively at the keyboard or within program code. The word **ENVIRONMENT?** expects to find on the stack the address and length of a text string referring to an option or parameter, and returns either a single *false* flag (parameter/option is unknown), or a *true* flag (known) on top of the stack, with a second flag or data value beneath. The word **S"** (see Section 4.1.5.2), which returns the address and length of a string, is often used with **ENVIRONMENT?**. For example, the string STACK-CELLS is defined as indicating the maximum number of cells in the data stack. You might type at the keyboard, or include in a definition, the phrase:

```
S" STACK-CELLS" ENVIRONMENT?
```

which might return

```
256 -1
```

where the **-1** (true) indicates that the system recognized the STACK-CELLS string, and the **256** shows that the maximum size of the stack is 256 cells. Table 6 gives the standard strings available for environmental queries and the data values they may return. The data type is the type of the associated data or second flag.

Most word sets contain a basic part and extensions, which may be tested for individually. For example, in this table, BLOCK and BLOCK-EXT separately test for the presence of the basic block word set and the block extensions word set.

Table 6: Environmental query strings and associated data

String	Type	Meaning
/COUNTED-STRING	*n*	Maximum size of a counted string, in characters.
/HOLD	*n*	Maximum size of pictured numeric output string, in characters.
/PAD	*n*	Size of the scratch area **PAD** in characters.
BLOCK	*flag*	*true* if block word set is present.
BLOCK-EXT	*flag*	*true* if block extensions word set is present.
CORE	*flag*	*true* if complete Standard Forth core word set is present.
CORE-EXT	*flag*	*true* if complete Standard Forth core extensions word set is present.
DOUBLE	*flag*	*true* if double number integer word set is present.
DOUBLE-EXT	*flag*	*true* if double-number extensions integer word set is present.
EXCEPTION	*flag*	*true* if exception word set is present.
EXCEPTION-EXT	*flag*	*true* if exception extensions word set is present.
FACILITY	*flag*	*true* if facility word set is present.
FACILITY-EXT	*flag*	*true* if facility extensions word set is present.
FILE	*flag*	*true* if file word set is present.

Table 6: Environmental query strings and associated data *(continued)*

String	Type	Meaning
FILE-EXT	*flag*	*true* if file extensions word set is present.
FLOATING	*flag*	*true* if floating-point word set is present.
FLOATING-EXT	*flag*	*true* if floating-point extensions word set is present.
FLOATING-STACK	*n*	If *n*=0, floating-point numbers are kept on the data stack; otherwise, *n* is the maximum depth of the separate floating-point stack.
FLOORED	*flag*	*true* if floored division is the default, *false* if symmetric division is the default.
MAX-CHAR	*u*	Maximum value of a character in the implementation-defined character set.
MAX-D	*d*	Largest usable signed double number.
MAX-FLOAT	*r*	Largest usable floating-point number.
MAX-N	*n*	Largest usable signed integer.
MAX-U	*u*	Largest usable unsigned integer.
MAX-UD	*ud*	Largest usable unsigned double number.
MEMORY-ALLOC	*flag*	*true* if memory-allocation word set is present.
RETURN-STACK-CELLS	*n*	Maximum size of the return stack, in cells.
STACK-CELLS	*n*	Maximum size of the data stack, in cells.
SEARCH-ORDER	*flag*	*true* if search-order word set is present.
SEARCH-ORDER-EXT	*flag*	*true* if search-order extensions word set is present.
STRING	*flag*	*true* if string word set is present.
TOOLS	*flag*	*true* if programming tools word set is present.
TOOLS-EXT	*flag*	*true* if programming tools extensions word set is present.
WORDLISTS	*n*	Maximum number of word lists usable in the search order.

Because a system may load options in any order, some environmental queries could return either *false* or *true*, depending on when they were executed. The Standard Forth requirements are:

- If a query returns *false* (unknown) in response to a string, subsequent queries with that string *may return true*, because additional capabilities may have been

acquired.

- If a query returns *true* (known) and a numerical value, subsequent queries with the same string *must also return true* and the same numerical value. In other words, added capabilities may not take away or fundamentally alter entitlements already presented to the program.
- Flags indicating presence or absence of optional word sets *may* change; the flag indicating floored or symmetric division *may not* change.

Glossary

ENVIRONMENT? *(c-addr u — false | i*x true)* Core

Return information about the system software configuration. The character string specified by *c-addr u* should contain one of the strings from Table 6. If it does not, return *false*; otherwise, return *true* with data specified in Table 6 for that string. "environment-query"

3.3 SERIAL I/O

Forth supports a variety of means to perform I/O with a terminal, printer, or other serial-type I/O device. In addition, a simplified method is provided to make use of cursor positioning and other hardware-dependent features, without forcing the use of particular terminal models.

References **ACCEPT**, Section 3.3.1
TYPE, Section 3.3.2

3.3.1 Terminal Input

The words described in this section handle character input from devices. The input is received from the *current input device* (e.g., keyboard, serial port). Selection of the current input device is system dependent.

ACCEPT awaits a character string from the terminal or other serial device, given the maximum number of characters and the address where they are to be stored. Input is terminated by a return ($0D_H$). If the terminator is not received before the maximum character count is reached, the excess characters

are discarded. **ACCEPT** returns the length of the character string that was stored at the given address. For example,

```
PAD 10 ACCEPT
```

will await up to ten characters, place them at **PAD**, and return the actual character count on the stack.

On most systems, incoming characters are checked for the return, which terminates input; and for backspace (08) or DEL ($7F_H$), which cause the character pointer to be "backed up" one and a backspace (or equivalent) to be sent to the terminal. All other characters are echoed to the terminal.

ACCEPT should not be executed if there is no terminal or serial device capable of providing input for the task.

No indication is provided at the terminal that the system is awaiting input as a result of an **ACCEPT** request. The programmer should indicate this fact through some output message issued prior to the **ACCEPT** request.

The command **KEY** awaits one character and leaves it on the stack. It does not edit or echo.

The conventional place to put incoming strings is the input message buffer. At least 80 bytes are available. The system text interpreter **ACCEPT**s 80 bytes into the input message buffer and performs the necessary housekeeping to process the text. The text interpreter is called by **QUIT**, which performs a terminal's basic idle loop behavior.

Glossary

ACCEPT (*c-addr* $+n_1$ — $+n_2$) Core

Get, at most, $+n_1$ characters from the current input device, echo each, and place them in memory beginning at *c-addr*. The process continues until **ACCEPT** encounters a carriage return (line terminator). If the line terminator is not received before a count of $+n_1$ is reached, any excess characters are discarded. Return $+n_2$, the actual count of characters received. An example of use is:

```
PAD 5 ACCEPT <carriage-return> 12345 ok
. 5  ok
```

ACCEPT is used for most terminal input. On many systems, **ACCEPT** will back up over previously input characters in response to the backspace or rubout key. When the character pointer points to *c-addr*, the original address, **ACCEPT** stops backing up and may thereafter emit a tone for each backspace or rubout it receives.

EKEY *(— u)* Facility Ext

Receive one keyboard event and place the result on the stack. The encoding of keyboard events is system dependent. "E-key"

EKEY>CHAR *(u — u 0 | char -1)* Facility Ext

Attempt to convert a keyboard event into a character. If successful, return the character and *true*, otherwise return the event and *false*. "E-key-to-care"

EKEY? *(— flag)* Facility Ext

Check whether a valid keyboard event has been received on the task's serial device since the last call to **ACCEPT**, **KEY**, or **EKEY**. If so, return *true*, otherwise return *false*. The value of the event may be obtained by the next execution of **EKEY**. After **EKEY?** returns with a value of *true*, subsequent executions of **EKEY?** before executing **KEY**, **KEY?**, or **EKEY** will also return *true*, because they refer to the same event. "E-key-question"

KEY *(— b)* Core

Accept exactly one byte of data from the input device and place its value on the stack. **KEY** does not echo. **KEY** is sometimes used for input prompting and in serial protocols. **KEY** is also often useful to interactively determine the ASCII numeric value of a character. For example, if you type:

```
KEY .
```

…the system will wait for you to press one key and will display its ASCII value.

KEY? *(— flag)* Facility

Check whether a valid character has been received on the current input device since the last call to **ACCEPT**, **KEY**, or **EKEY**. If so, return *true*, otherwise return *false*. Invalid (non-character) keyboard events occurring before a valid character are discarded and made unavailable. The value of the character received may be obtained by the next execution of **KEY**. After **KEY?** returns with a value of *true*, subsequent executions of **KEY?** before executing **KEY** or **EKEY** will also return *true*, without discarding keyboard events. "key-question"

References String operations, Section 2.3
Input number conversion, Section 4.1.4
QUIT, Section 4.1.2

3.3.2 Terminal Output

Forth provides words to output character strings, as well as single characters. The output is sent to the current output device (e.g., display, printer). Selection of the current output device is system dependent.

TYPE outputs a character string to the terminal or other serial device (such as a printer). The character string is emitted exactly as it appears in storage, with parity bits added by the hardware, if required by the terminal in use.

The length of the string, in bytes, must be on top of the stack, with the address of the first byte of the string beneath it.

For example, you could use the following phrase to display thirty-two characters from **PAD** on the terminal:

```
PAD 32 TYPE
```

The command **EMIT** will transmit a single ASCII character, given its value on the stack. Thus,

```
65 EMIT
```

will output an "A".

Glossary

EMIT *(b —)* Core
Output one character from the least-significant byte of the top item on the stack, then pop the stack. **EMIT** is often useful for initial "cut-and-try" definitions.

EMIT? *(— flag)* Facility Ext
Check that it is okay to output a character (e.g., the device is ready, etc.). Return *flag*, which is *false* if it is known that the execution of **EMIT** instead of

EMIT? would suffer an indefinite delay; otherwise, return *true*, including the case where the device status is indeterminate. Used, for example, in modem protocols with the RTS line. "emit-question"

TYPE (*c-addr u* —) Core

Output the character string at *c-addr*, length *u*.

References **PAD**, Section 2.3.1
Scanning strings, Section 4.1.5
Vectored execution, Section 2.5.6

3.3.3 Support of Special Terminal Features

Each terminal task in a Forth system has unique user variables, including a port address, or other device- and system-specific interrupt vectoring. Each task may require different control character sequences for functions such as **CR** (go to beginning of next line) and **PAGE** (go to top of next page).

The standard Forth words that perform terminal functions are listed in this section. The method of vectoring these functions to the particular output sequences required for given devices is system dependent.

Glossary

AT-XY (u_1 u_2 —) Facility

Configure the current output device so the next character displayed will appear in column u_1, row u_2 of the device's output area. The upper-left corner of this area is at $u_1 = 0$, $u_2 = 0$. "at-X-Y"

CR (—) Core

Cause subsequent output to appear at the beginning of the next line on the current output device. "C-R"

PAGE (—) Facility

Move to another page for output on the current device. On a CRT, clear the screen and reset the cursor position to the upper-left corner. On a printer, perform a form feed.

SPACE (—) Core

Display one space on the current output device.

SPACES (*u* —) Core

Display *u* spaces on the current output device.

3.4 Block-Based Disk Access

Forth systems provide access to mass storage using either a block-based or a file-based method (and occasionally both). In a block-based system, mass storage is partitioned into some number of *blocks*, each 1024 bytes long. These blocks may be in files, depending on the underlying operating system (if any). In a file-based system, a host operating system is required. It provides and manages files of variable length, which Forth uses directly for mass storage.

This section discusses the words used to access and manage disk blocks and block buffers in Forth. Section 3.5 discusses the words used to access mass storage using files. One of these sections (and sometimes both) will be relevant to a particular Forth system.

3.4.1 Overview

The block-based disk access method is intended to be simple and to require a minimum of effort to use. The disk driver makes data on disk directly accessible to other Forth words by copying disk data into a buffer in memory, and placing on the stack the buffer's address. Thus, Forth routines access disk data using the same techniques it uses for other memory accesses. Because disk data always appears to be in memory, this scheme is a form of *virtual memory* for program source and data storage.

Another consideration in the design of the disk driver is to make disk access as fast as possible. Because disk operations are very slow, compared to memory operations, data is read from disk or written to disk only when necessary.

The disk is partitioned into 1024-byte data areas called *blocks*. This standard unit has proven to be a useful increment of mass storage. As a unit of source text, for example, it contains an amount of source which can be comfortably

displayed on a CRT screen; as the basis for a database system, it is a convenient, common multiple of typical record sizes.

Each block is addressed by a block number. On native Forth systems, the block number is a fixed function of the block's physical position on the disk. Absolute addressing of the disk both speeds the driver's execution and eliminates most of the need for disk directories and indexes. On OS-hosted Forth systems, the blocks may be located in one or more files, each an integral multiple of the block size, and an internal table maps OS files to block space.

3.4.2 Block-Management Fundamentals

A program ensures that a block is in memory in a *block-buffer* by executing the word **BLOCK**. **BLOCK** uses a block number from the stack and returns the address of the first byte of that block in memory. For example:

```
9 BLOCK U.
```

will return an address such as:

```
46844 ok
```

where 46844 is the address of the first byte of the buffer containing block 9. If a block is already in memory, **BLOCK** will not re-read it from disk.

Although **BLOCK** uses a disk read to get data if it is not already in memory, **BLOCK** is not merely a read command. If **BLOCK** must read a requested block from disk, it uses **BUFFER** to select a buffer to put it in. **BUFFER** frees a block buffer, writing the buffer's previous contents to disk if it is marked (by **UPDATE**, see below) as having been changed since the block was read into memory.

BUFFER expects a block number on the stack, and returns the address of the first byte of the available block buffer it assigns to this block. For example:

```
127 BUFFER U.
```

will get a block buffer, assign block number **127** to the buffer, and then type the address of the buffer's first byte:

```
36084 ok
```

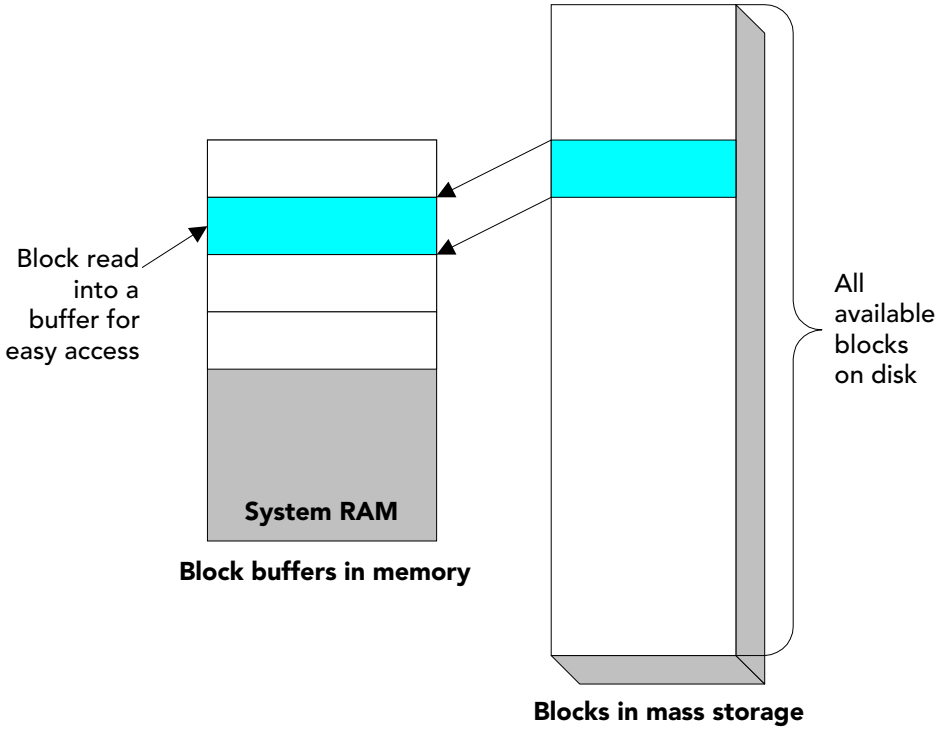

Figure 8. Block handling in a file-based Forth system

Although **BUFFER** may write a block, if necessary, it will *not* read data from disk. When **BUFFER** is called by **BLOCK** to assign a buffer, **BLOCK** will follow the selection of a buffer by actually reading the requested block from disk into the buffer.

The following example displays an array of the first 100 cells in block 1000, shown with five numbers per line:

```
: SHOW ( -- )                    \ Display array contents
   100 0 DO
      I 5 MOD 0= IF CR THEN      \ Allow 5 per line
      1000 BLOCK I CELLS + ?     \ Show Ith value in block
   LOOP ;
```

The phrase **I CELLS +** converts the loop counter from cells to bytes (because internal addresses are always byte addresses), and adds the resulting byte offset to the address of the block buffer returned by **BLOCK**. The word **?** fetches and types the cell at that address.

BUFFER may be used directly (i.e., without being called by **BLOCK**) in situations where no data needs to be read from the disk. Examples include initializing a region of disk to a default value such as zero, or a high-speed data acquisition routine writing incoming values directly to disk from a memory array 1024 bytes at a time.

Forth systems will have at least one, but usually many, block buffers. The number of buffers may be changed easily. Applications with several users using disk heavily may run slightly faster with more buffers. Your product documentation will give details on changing the size of the buffer pool.

The command **UPDATE** marks the data in a buffer as having been changed, so that it will be written to disk when the buffer must be used for another block. **UPDATE** works on the most recently referenced buffer, so it must be used immediately after any operation that modifies the data.

The following example uses **BUFFER** to clear a range of blocks to zero:

```
: ZEROS ( first last -- )
   1+ SWAP DO
      I BUFFER 1024 ERASE  UPDATE
   LOOP ;
```

As another example, assume that an application has defined **A/D** to read a value from an A/D converter. To record up to 512 samples in block 700, use:

```
: SAMPLES ( n -- )           \ Record n samples
   512 MIN 0 DO              \ Clip n at 512
      A/D                    \ Read one sample
      700 BLOCK I CELLS + ! UPDATE  \ Record it
   LOOP ;
```

In this example, the phrase **512 MIN** "clips" the specified number of samples at 512. As in the example of **SHOW** above, the phrase **I CELLS** converts the loop counter (in samples) into a byte offset to be added to the address of the start of the block, returned by **BLOCK**. **BUFFER** cannot be used in this case,

because we are adding samples one at a time and must preserve previous samples written in the block.

Because **BLOCK** maps disk contents into memory, *virtual memory* applications are simple. The first step is to write a word to transform an application address into a physical address, consisting of a block number and an offset within that block. For a virtual byte array, such a definition is:

```
: VIRTUAL ( i -- a ) \ Return the addr of the ith byte
   1024 /MOD         \ Q=blk offset, R=byte in the block
     250 +           \ Add starting blk#=250
   BLOCK + ;         \ Fetch block, add byte offset
```

Here, 1024 is the number of bytes per disk block and 250 is the block number where the virtual array starts. The array may occupy any number of blocks, limited only by physical mass storage constraints.

Fetch and store operations for this virtual memory scheme are defined as:

```
: V@ ( i -- n )        \ Return ith byte in the array
    VIRTUAL C@ ;

: V! ( b i -- )        \ Store b in ith byte
    VIRTUAL C!  UPDATE ;
```

BLOCK does not normally perform any error checking or retries at the primitive level, because an appropriate error response is fundamentally application-dependent. Some applications processing critical data in non-real-time (e.g., accounting applications) should attempt retries* and, if these fail, stop with an error message identifying bad data. Other applications running continuously at a constant sampling rate (e.g., data loggers) cannot afford to wait, and should simply log errors.

* Most disk controllers and all OSs perform retries automatically. On these, there is nothing to be gained by attempting retries from within a Forth application.

Glossary

BLOCK *(u — addr)* Block

Return the address of a buffer containing a copy of the contents of block *u*, having read it from disk, if necessary. If a read occurred, the previous contents of the buffer is first written to disk, if it has been marked as updated.

BUFFER *(u — addr)* Block

Return the address of a buffer marked to contain block *u*, having written previous contents to disk, if necessary (does not perform any read operation).

UPDATE *(—)* Block

Mark the most recently referenced buffer as having been updated. The contents of a buffer that has been marked in this way will be written to disk when its buffer is needed for a different block.

FLUSH *(—)* Block

Ensure that all updated buffers are written to disk, and free all the buffers.

SAVE-BUFFERS *(—)* Block

Write all updated buffers to disk, leaving them in the buffers but with their **UPDATE** flags cleared.

EMPTY-BUFFERS *(—)* Block Ext

Erase all block buffers without saving them. **EMPTY-BUFFERS** works by clearing the update bits in all buffers and performing a **FLUSH** to free the buffers.

3.4.3 Loading Forth Source Blocks

Most compiled languages require a three-step process to construct executable programs:

1. Compile the program to an object file on disk.
2. Link this program to other previously compiled and/or assembled routines.
3. Load the result into memory.

This often-lengthy procedure has a negative effect on a programmer's effectiveness. Forth supports fully interactive programming by shortening this

cycle to a single, fast operation: compiling from source to executable form in memory. This process is accomplished by the word **LOAD**.

3.4.3.1 The LOAD Operation

LOAD specifies the interpretation of source text from a disk block. **LOAD** expects on top of the stack the block number of the Forth block to be **LOAD**ed:

```
<number> LOAD
```

This block number is also stored in the variable **BLK**, used by Forth's text interpreter. If **BLK** contains zero, the source is not a block, and usually is the terminal. When **BLK** is zero, the word **SOURCE-ID** returns a value indicating the input source (zero if it is the user input device or terminal, -1 if it is a character string passed by evaluate, and optionally a file-identifier if the input is a text file—see Section 3.5).

When **LOAD** is encountered, interpretation of text from the current input source is suspended and input is taken from the specified disk block. The text interpreter starts at the beginning and processes each word until it reaches the end of the block after 1024 characters. On some systems, if the word **EXIT** is encountered interpretively in the block, it will cause processing to terminate at once.

When all processing specified by the disk block is complete (assuming no errors were encountered while processing the block), execution resumes with input from the source that was in control when the **LOAD** was encountered.

If a block contains definitions, the result of a **LOAD** operation will be to process them via the text interpreter and compile them into the dictionary. The process of **LOAD**ing disk blocks is identical to processing the same information entered from the terminal or text file, but all information in a single disk block is processed as a single string (i.e., there will be no embedded carriage returns).

The block to be **LOAD**ed may itself contain a **LOAD** command, at which point the **LOAD**ing of the first block is suspended. When this occurs, the block number of the current block, the current text interpreter pointers are saved on the return stack, pending loading of the requested block. This nested **LOAD**ing process may continue indefinitely, subject to return stack size.

A group of blocks to be **LOAD**ed should be specified by **LOAD** commands con-

tained in a single block, called a *load block*, as opposed to serial nesting (i.e., having each block load the next block in sequence). From a management viewpoint, loading groups of related blocks from a single load block aids readability and maintainability.

The command **THRU** can load a group of sequential blocks. For example, if blocks 260 through 270 need to be loaded, **THRU** could be used:

```
260 270 THRU
```

A **LOAD** operation may also be compiled in a definition, in which case the requested **LOAD** is done when the definition is executed. Following the **LOAD**, execution will resume at the word immediately after **LOAD**.

If an error is detected during the **LOAD**ing process, an error message is produced and all **LOAD**ing ceases. Both the return stack and the data stack are cleared, and Forth reverts to terminal input.

During loading, all text interpreter input is taken from the specified disk block. All output, however, proceeds to its normal destination. Thus, **.** ("dot") or other output commands will send output to the terminal of the task executing the **LOAD**.

Glossary

BLK *(— a-addr)* Block

Return *a-addr*, the address of a cell containing the number of the mass-storage block being interpreted, or zero if the current input source is not a block. "B-L-K"

LOAD *(i*x u — j*x)* Block

Save the current input source specification in a system-specific manner. Store *u* in the variable **BLK**, thus making block *u* the input source. Set the input buffer to contain the contents of block *u*. Set the buffer pointer **>IN** to zero and interpret the buffer contents. When the parse area is exhausted, restore the prior input specification. Any other stack effects are due to the words executed as a result of the **LOAD**.

THRU *(i*x u_1 u_2 — j*x)* Block Ext

Execute **LOAD** in sequence for each of the blocks numbered u_1 through u_2. Any other stack effects are due to the words executed as a result of the **LOAD**s.

References **EXIT**, Section 2.5.5
Input source identification, Section 4.1.1
Text file identifiers, Section 3.5.1

3.4.3.2 Named Program Blocks

The defining word **CONSTANT** may be used to give names to important blocks, such as load blocks, which load other blocks to form a utility or application. For example, define:

```
120 CONSTANT OBSERVING
```

which will be used as:

```
OBSERVING LOAD
```

The above has the effect of loading block 120 and executing any other **LOAD** instructions specified in that block.

CONSTANT is particularly appropriate when you want to use the name in other ways, such as:

```
OBSERVING LIST
```

We recommend the use of a *key block* for each major section of an application. The key block should primarily load other associated blocks, specified numerically or through **CONSTANTS**; it may also contain other brief, application-wide definitions. Then you can see at a glance which of your application blocks are loaded, and in what order.

This technique is much safer than *chaining* blocks (i.e., serial nesting), which can cause a return-stack overflow. Generally, a single block names all the key blocks in that system, and is **LOAD**ed immediately after booting.

A convenient side effect of named blocks is that they can be successfully **LOAD**ed when in any number conversion base. But, for this reason, named key blocks should have a **DECIMAL** command in the first line to guard against incorrect loading of subsidiary blocks due to an unexpected current base.

References **CONSTANT**, Section 4.2.3
LOAD and the return stack, Section 3.4.3.1
LIST, Section 3.4.3.3

3.4.3.3 Block-based Programmer Aids and Utilities

As a consequence of its standalone heritage, Forth has traditionally accompanied its block-based systems with a rich portfolio of programmer aids and utilities. These will vary depending upon the implementation, but a fully supported block system will normally include:

- **An editor.** Traditional Forth block editors format a block in 16 lines of 64 characters each, as this is a convenient size on most displays. By convention, the first line of each block includes a comment summarizing the contents of that block. The balance of the block should contain a few simple definitions related to its stated objective. Most block editors provide a command line and are string oriented. Some are quite powerful. All will include the basic command **LIST** to display a block, and the variable **SCR** which contains the number of the block most recently **LIST**ed.
- **Shadow-block, on-line documentation.** Space within a block is limited, so comments are conventionally kept in a separate block, and the system pairs each source block with its shadow. From a keyboard, you should be able to toggle between a source block and its shadow counterpart. Shadow blocks are not compiled or executed.
- **Program listing utilities.** Typical systems include a utility to print indexes (the first, or comment, line from each of a range of blocks) and lists of blocks. Depending on the printer, it is normally possible to print source and shadow blocks side-by-side, with three such pairs on a page.
- **Disk-management utilities.** These include simple functions for moving groups of blocks and their associated shadows, initializing regions of disk, browsing disk (displaying the first-line comments), etc.
- **Source-block comparison utilities.** Comparison utilities that highlight any differences between ranges of similar blocks are extremely helpful on multi-programmer projects when work has to be merged from several sources.
- **Programmer aids.** The programmer aids described in Section 1.4 (page 1–22) are normally available on block-based systems. **LOCATE**, for example, will show the block from which a word was compiled; with a single keystroke, you

can display its associated shadow block.

Consult your product documentation for further details regarding your system's features.

Glossary

LIST *(u —)* Block Ext

Display block *u* in a system-dependent format (usually 16 lines of 64 characters each). Store *u* in the variable **SCR**.

SCR *(— a-addr)* Block Ext

Return *a-addr*, the address of a cell containing the block number of the most recently **LIST**ed block. "S-C-R"

3.5 File-Based Disk Access

Forth systems provide access to mass storage using a block-based or file-based method. This section discusses words that access mass storage using files. Section 3.4 discusses words used to access and manage disk blocks and block buffers in Forth. One or the other of these sections (occasionally both) will be relevant to a particular Forth system.

In a block-based system, mass storage is partitioned into *blocks* of 1024 bytes. In a file-based system, a host operating system is required; it provides and manages files of variable length, which Forth uses directly for mass storage.

Many items discussed in this section, such as the specific value and meaning of non-zero I/O result codes, allowable forms of filenames, values of line terminators, etc. are system dependent. Consult your product documentation for details.

3.5.1 Overview

The Forth words described in this section provide access to mass storage in the form of *files*, under the following conditions and assumptions:

- Files are provided by a host operating system.
- File state information (e.g., current position in the file, size, etc.) is managed by the host operating system. File sizes are dynamically variable, so write operations will increase the size of a file as necessary.
- Filenames are represented as character strings. The format of the names is determined by the host operating system. Filenames may include system-specific pathnames.
- A *file identifier* (*fileid*) is a single-cell value that is passed to file operators to refer to specific files. Opening a file assigns it a file identifier, which remains valid until the file is closed. When the text interpreter is using a file as the input, its *fileid* will be returned by **SOURCE-ID**. The other possible values that **SOURCE-ID** can return are zero (if the user input device is the source), and -1 (if the source is a character string passed by **EVALUATE**).
- File contents are accessed as a sequence of characters. The *file position* is the character offset from the start of the file. The file position is updated by all read, write, and re-position commands.
- File read operations return an *actual* transfer count, which can differ from the *requested* transfer count.
- A *file access method* (*fam*) is a single-cell value indicating a permissible means of accessing a specific file, such as read/write or read-only.
- An *I/O result* (*ior*) is a single-cell value indicating the result of an I/O operation. A value of zero always indicates success; non-zero values are definition- and system-specific. An operation reaching the end of a file shall not consider it an error and shall return a zero *ior*.

3.5.2 Global File Operations

The words in this section are used to manipulate files as entire entities.

Glossary

CLOSE-FILE *(fileid — ior)* File

Close the file identified by the *fileid*. Return an I/O result code.

CREATE-FILE *(c-addr u fam — fileid ior)* Core

Create a file whose name is given by the character string at *c-addr* and whose length is *u*, and open it with file access method *fam*. If the file already exists, re-create it as an empty file, replacing the pre-existing file of that name. If creation and opening are successful, return an *ior* of zero and the *fileid*. Otherwise, return a non-zero *ior* and an undefined value for *fileid*.

DELETE-FILE *(c-addr u — ior)* File

Delete the file whose name is given by the character string at *c-addr* and whose length is *u*. Return an I/O result code.

FLUSH-FILE *(fileid — ior)* File Ext

Force any buffered contents of the file referred to by *fileid* to be written to mass storage, and the size information for the file to be recorded by the system, if changed. Return an *ior* of zero if successful; otherwise, return a system-dependent value.

OPEN-FILE *(c-addr u fam — fileid ior)* File

Open the file whose name is given by the character string at *c-addr* whose length is *u*, with file access method *fam*. If opening is successful, set the file position to zero, and return an *ior* of zero and the *fileid*; otherwise, return a non-zero *ior* and an undefined value for *fileid*.

RENAME-FILE *($c\text{-}addr_1$ u_1 $c\text{-}addr_2$ u_2 — ior)* File Ext

Rename the file whose current name is given by the character string at $c\text{-}addr_1$ and whose length is u_1, to the name given by the character string at $c\text{-}addr_2$ and whose length is u_2. Return an I/O result.

RESIZE-FILE *(ud fileid — ior)* File

Set the size of the file identified by *fileid* to *ud*, and return an I/O result code. If the file size is increased, the contents of the newly allocated space is indeterminate. After this operation (if successful), **FILE-SIZE** will return the same value for *ud*, and **FILE-POSITION** returns an undefined value.

3.5.3 File Reading and Writing

The words in this section are used to read or write to a specific file.

Glossary

INCLUDE-FILE *(fileid —)* File

Read and interpret the given file, performing the following steps: Save the current input source specification. Store the given *fileid* in **SOURCE-ID**, set **BLK** to zero, and make this file the input source. Read a line from the file at the current file position, fill the input buffer with the contents of the line, set **>IN** to zero, and interpret the buffer contents. Continue reading lines until the end of file is reached. When the end of the file is reached, close the file and restore the previous input source specification.

INCLUDED *(c-addr u —)* File

Same as **INCLUDE-FILE**, except the file is specified by its name, which is stored at *c-addr* and is of length *u*. The file is opened and its *fileid* is stored in **SOURCE-ID**.

INCLUDE `<filename>` *(—)* common usage

Same as **INCLUDE-FILE**, except the file is specified by the *filename* which follows in the input stream.

READ-FILE *(c-addr u_1 fileid — u_2 ior)* File

Read and store text from the given file, without interpretation, and update **FILE-POSITION**. From the current position in the file identified by *fileid*, read u_1 consecutive characters, storing them at *c-addr*. Return an *ior* and u_2, the number of characters successfully read. If no exception occurs, return *ior* of zero and $u_2 = u_1$ or the number of characters actually read before encountering the end of the file, whichever is smaller. If **FILE-POSITION** was equal to **FILE-SIZE** before executing **READ-FILE**, u_2 is zero. If a non-zero *ior* is returned, u_2 is the number of characters successfully transferred before the exception occurred.

READ-LINE *(c-addr u_1 fileid — u_2 flag ior)* File

Read and store one line of text from the given file, without interpretation, and update **FILE-POSITION**: From the current position in the file identified by

fileid, read up to u_1 consecutive characters, storing them at *c-addr*. Terminate the read if end-of-line delimiter(s) are encountered. Return an *ior* and u_2, the number of characters successfully read, not including the line delimiter(s) if any. One or two line delimiters may be read into memory at the end of the line in addition to u_2; therefore, the buffer at *c-addr* should be at least u_1+2 characters long. If $u_2 = u_1$, the line delimiter was not reached. If no exception occurs, return *ior* of zero and *flag* is true. If **FILE-POSITION** was equal to **FILE-SIZE** before executing **READ-LINE**, *flag* is false, *ior* is zero, and u_2 is zero. If an non-zero *ior* is returned, other returned parameters are undefined.

REFILL *(— flag)* Block Ext, Core Ext, File Ext

When the input source is a text file, attempt to read the next line from the current file. If successful, make the result the current input buffer, set **>IN** to zero, and return *true*; otherwise, return *false*.

WRITE-FILE *(c-addr u fileid — ior)* File

Write *u* characters from *c-addr* to the file identified by *fileid*, starting at its current file position. Increase **FILE-SIZE** if necessary. Return an I/O result code. After this operation, **FILE-POSITION** returns the next file position after the last character written to the file, and **FILE-SIZE** returns a value equal to or greater than **FILE-POSITION**.

WRITE-LINE *(c-addr u fileid — ior)* File

Same as **WRITE-FILE**, except a line terminator is written to the file after the *u* characters.

3.5.4 File Support Words

The words in this section provide support for other file access functions.

Glossary

BIN *(fam_1 — fam_2)* File

Modify the given file access method fam_1 to additionally select a binary (not line-oriented) file access method, returning the modified access method fam_2.

FILE-POSITION *(fileid — ud ior)* File

Return the double-cell current file position *ud* for the file identified by *fileid*, and an I/O result code. If the *ior* is non-zero, the position *ud* is undefined.

FILE-SIZE *(fileid — ud ior)* File

Return the double-length file size *ud* for the file identified by *fileid*, and an I/O result code. This operation does not affect the value returned by **FILE-POSITION**. If the *ior* is non-zero, the size *ud* is undefined.

FILE-STATUS *(c-addr u — x ior)* File Ext

Return the status of the file whose name is given by the character string at *c-addr* and whose length is *u*. The *ior* is zero if the file exists, otherwise it is a system-dependent value. Cell *x* contains system-dependent information about the file.

R/O *(— fam)* File

Return the read-only file access method. "R-O"

R/W *(— fam)* File

Return the read/write file access method. "R-W"

REPOSITION-FILE *(ud fileid — ior)* File

For the file identified by *fileid*, reset the file position to *ud*, and return an I/O result code. After this operation (if successful), **FILE-POSITION** will return this same value for *ud*.

S" `<string>` **"** *(— c-addr u)* Core, File

This word normally compiles a string in a definition, returning its address and count when executed. In a file-based disk access system, this word is extended to operate interpretively, for use with filenames. When interpreting, looks ahead in the input stream and obtains a character string, delimited by **"**. Stores the string in a temporary buffer (which is at least 80 characters long) and returns the address and length of the string. "S-quote"

W/O *(— fam)* File

Return the write-only file access method. "W-O"

3.6 Time and Timing Functions

Many Forth systems support an asynchronous, free-running milliseconds timer, and retrieval of date and time from a host operating system (if the host provides this function). The precision of the milliseconds timer depends on the resolution of the system clock and on relevant hardware characteristics. A task executing **MS** is suspended until its time-out period has elapsed.

Glossary

MS (*u* —) Facility Ext

Wait for at least *u* milliseconds, but not more than *u* plus twice the resolution of the system clock. "M-S"

TIME&DATE (— u_1 u_2 u_3 u_4 u_5 u_6) Facility Ext

Return the current time and date: u_1=seconds (0–59), u_2=minutes (0–59), u_3=hours (0–23), u_4=days (0–31), u_5=months (1–12), and u_6=years (0–9999).

3.7 Dynamic Memory Management

In some applications, the need arises for dynamic data storage. For example, a large number of asynchronous tasks may be taking data intermittently. When one of these tasks receives a burst of data, it needs a temporary buffer to hold and process the data, but can relinquish the buffer when processing is complete.

The words in this section allocate, resize, and free regions of data space. Memory regions allocated in this way are located at arbitrary addresses, and so are useful only for data. They cannot be used, for example, for the Forth dictionary, because there is no way for an application to manage the dictionary pointer. Although a given region will be internally contiguous, it is not guaranteed to be contiguous with any other regions, so no operations should be performed that attempt to cross a region's boundary.

Glossary

ALLOCATE *(u — a-addr ior)* Memory

Attempt to allocate *u* bytes of contiguous data space. The dictionary pointer is unaffected by this operation. The initial content of the allocated space is not defined. If the allocation is successful, the aligned starting address *a-addr* of the allocated space and an *ior* of zero is returned. If the allocation is not successful, *a-addr* is an undefined value and a system-dependent non-zero *ior* is returned.

FREE *(a-addr — ior)* Memory

Release the contiguous data space identified by *a-addr* to the system for later re-allocation. The address *a-addr* is a value previously returned by **ALLOCATE** or **RESIZE**. The dictionary pointer is unaffected by this operation. If the release operation succeeds, *ior* is zero; otherwise, it is a system-dependent non-zero value describing the failure.

RESIZE *($a\text{-}addr_1$ u — $a\text{-}addr_2$ ior)* Memory

Change the size of a contiguous data space previously allocated by **ALLOCATE** or **RESIZE** at $a\text{-}addr_1$ to *u* bytes, where *u* may be either larger or smaller than the current size of the space. The dictionary pointer is unaffected by this operation. If the operation succeeds, $a\text{-}addr_2$ is the aligned starting address of the *u* bytes of allocated memory, and *ior* is zero. $a\text{-}addr_2$ may be, but need not be, the same as $a\text{-}addr_1$. In any case, the contents of the area before and after the **RESIZE** are preserved up to *u* bytes or to the original size, whichever is smaller. If $a\text{-}addr_2$ is not the same as $a\text{-}addr_1$, the region of memory at $a\text{-}addr_1$ is released to the system as by **FREE**. If the resize operation fails, $a\text{-}addr_2$ equals $a\text{-}addr_1$, the contents of the region of memory at $a\text{-}addr_1$ is unaffected, and a system-dependent non-zero *ior* code is returned.

3.8 Floating Point

Many Forth applications, especially embedded applications, do not require floating-point routines. Arithmetic that, at first glance, would seem to need floating-point calculations, can often be done more simply, taking less memory and executing faster, when coded with integer operators and with intelligent use of scaling words such as ***/**. The key issue is dynamic range in the variables of interest; if that is limited to fewer than 15 bits, say (as it usually will be

if driven by I/O devices), integer math is usually the better choice.

For some applications, however, floating-point mathematics is essential. Forth defines a full set of optional floating-point operators to support such applications. This section describes the general operators available on systems that comply with Standard Forth systems and support this option. Their implementation is very system specific, and may use floating-point hardware (such as a numeric processor). Your implementation-specific documentation should also be consulted for additional features that may be present (such as hardware-stack implementation and additional hardware error trapping on the 80387/80486 floating-point processor).

References Multi-stack notation, Section 2.1.1

3.8.1 Floating-Point System Guidelines

Because floating-point packages may exist on systems with widely varying hardware capabilities, and thus may require different implementation strategies, the basic Standard Forth floating-point word set is flexible in many areas. For details of a particular implementation, you will need to consult CPU-specific documentation. The following guidelines apply:

- The internal representation of a floating-point number, including the format and precision of both significand and exponent, is implementation specific, as is the largest usable floating-point number. For portability, supplementary words are defined that fetch and store to standard 32- or 64-bit IEEE floating-point number format (see ANSI/IEEE Standard 754 -1985).
- Since the length in memory of a floating-point number is implementation specific, the question of alignment arises. A *float-aligned address* (stack comment *f-addr*) is an address where a floating-point number can be accessed. Similarly, a *single-float aligned address* (*sf-addr*) or *double-float-aligned address* (*df-addr*) is an address where a single-precision (32-bit) or double-precision (64-bit) IEEE standard floating-point number can be accessed.
- There is a logically separate *floating-point stack*. Both the width and the depth are implementation specific, but the stack must be able to contain at least six items.
- The floating-point stack may be physically separate, or it may be implemented using the data stack. If it uses the data stack, integer data and floating-point

numbers can become mixed on the same stack. An application program intended to be portable across different implementations (with and without separate stacks) must order its operations carefully. For example, it must clear the floating-point stack of all items before trying to access any data stack items that may be underneath (and vice versa). It must also ensure that arguments to operations using both stacks (e.g., **F!**) are produced in the correct order. A program can determine whether floating-point numbers are kept on the data stack by passing the string `FLOATING-STACK` to **ENVIRONMENT?** (see Section 3.2). If the value returned is zero, the data stack is used; otherwise, the non-zero value indicates the maximum depth of the separate floating-point stack.

- For floating-point input and output, the current base must be **DECIMAL**; if the base is other than decimal, number conversion or display will not take place. Floating-point numbers to be interpreted by a system that complies with Standard Forth must contain an exponent indicator **E** or **e**. For example, one legitimate floating-point representation of the number 12300 is 1.23E4, where 1.23 is the *significand* and 4 is the *exponent*.
- Floating-point operators may address memory in data space regions declared with **FVARIABLE**. These regions are *not necessarily contiguous* with subsequent regions allocated with **,** (comma) or **ALLOT**.

3.8.2 Input Number Conversion

A floating-point number in Forth must contain an **E** or an **e** (signifying an exponent), and must begin with a digit (optionally preceded by an algebraic sign). For example, `-0.5e0` is valid, but `.2e0` is not. A number does not need to contain a decimal point or a value for the exponent; if there is no exponent value, it is assumed to be zero (multiplier of one). *Punctuation other than a decimal point is not allowed in a floating-point number.*

During number conversion, **BASE** must be **DECIMAL** so that numbers such as `1E` are not interpreted as hexadecimal digits. If **BASE** is not **DECIMAL**, floating-point number conversion will not take place.

All the following are valid floating-point numbers:

```
3.14159E+00  -3E-07  1e  1.E  0.005e02
```

but the following are double-precision integers (under the enhanced rules

described in Section 1.1.6), not floating-point numbers:

```
3.14159  -1,000,000.12  -0.003
```

Input conversion of floating numbers is accomplished by adding an additional level to Standard Forth number conversion routines. First, an attempt is made to convert an input string to a floating-point number. If this succeeds, the number is returned on the floating-point stack; otherwise, control passes to the integer number conversion routines. Thus, `20.E` would be converted as a floating number and `20.` as a double-precision integer.

References Input number conversion, Sections 1.1.6, 4.1.4

3.8.3 Output Formats

Three standard output formats are provided to display floating-point numbers. All of them remove the top item on the floating-point stack. The number of significant digits to display is set globally for all three formats and will remain in use until changed. There is also low-level support for custom output (and input) formatting; see Section 3.8.11.

Glossary

F. *(F: r —)* Floating Ext

Display the top number on the floating-point stack, followed by a space. Uses fixed-point notation (decimal point only, no exponent). The number of significant digits displayed is set by **SET-PRECISION**. "F-dot"

FE. *(F: r —)* Floating Ext

Display the top number on the floating-point stack, followed by a space. Uses engineering notation (the significand is greater than or equal to 1.0 and less than 1000.0, and the decimal exponent is a multiple of three). The number of significant digits displayed is set by **SET-PRECISION**. "F-E-dot"

FS. *(F: r —)* Floating Ext

Display the top number on the floating-point stack, followed by a space. Uses scientific notation (significand plus exponent), where the significand is greater than or equal to 1.0 and less than 10.0. The number of significant digits dis-

played is set by **SET-PRECISION**. "F-S-dot"

PRECISION (— *u*) Floating Ext
Return the number of significant digits currently displayed by **F.**, **FE.**, or **FS.**.

SET-PRECISION (*u* —) Floating Ext
Set the number of significant digits to be used by **F.**, **FE.**, or **FS.** to *u*.

3.8.4 Floating-Point Constants, Variables, and Literals

There are floating-point counterparts to the integer Forth words **CONSTANT**, **VARIABLE**, and **LITERAL**. The memory storage requirements, maximum value, and precision of the floating-point versions are implementation specific.

Glossary

FCONSTANT <name> (*F: r* —) Floating
Define a floating-point constant with the given *name* whose value is *r*, e.g., **3.14159E FCONSTANT PI**. When *name* is executed, the value *r* is returned on the floating-point stack. "F-constant"

FLITERAL (*F: r* —) Floating
Used only within a definition. When the definition is compiled and the word **FLITERAL** is reached, there must be a value *r* on the floating-point stack, which will be removed and added to the definition. When the definition is executed, **FLITERAL** returns the value *r* on the floating-point stack. "F-literal"

FVARIABLE <name> (—) Floating
Define a floating-point variable with the given *name*. Execution of *name* will return the address of its data space. Floating-point variables may be stored in a separate region of memory, and subsequent allocations of memory with **,** or **ALLOT** may not be contiguous with an **FVARIABLE**. An **FVARIABLE** may be initialized with, e.g., **F!** (see below). "F-variable"

3.8.5 Memory Access

Memory access words similar to those in other parts of a Forth system are pro-

vided for floating-point data types. These words obtain addresses from the data stack, and transfer data to and from the floating-point stack. Named regions for transferring IEEE standard data formats may be defined with phrases such as **CREATE** … **ALLOT** (not **FVARIABLE**, because its length in memory is implementation specific).

Glossary

F! *(f-addr —); (F: r —)* Floating

Store the floating-point value *r* at *f-addr*. In single-stack implementations, *f-addr* must be on top of the stack. "F-store"

F@ *(f-addr —); (F: — r)* Floating

Fetch the value stored at *f-addr* to the floating-point stack. "F-fetch"

DF! *(df-addr —); (F: r —)* Floating Ext

Store the floating-point value *r* as a 64-bit IEEE double-precision number at *df-addr*, rounding if the internal representation has more precision. In single-stack implementations, *df-addr* must be on top of the stack. "D-F-store"

DF@ *(df-addr —); (F: — r)* Floating Ext

Fetch the 64-bit IEEE double-precision number at *df-addr*, convert to internal representation, and place on the floating-point stack, rounding if the internal representation has less than 64-bit precision. "D-F-fetch"

SF! *(sf-addr —); (F: r —)* Floating Ext

Store the floating-point value *r* as a 32-bit IEEE single-precision number at *sf-addr*, rounding if the internal representation has more than 32-bit precision. In single-stack implementations, *sf-addr* must be on top of the stack. "S-F-store"

SF@ *(sf-addr —); (F: — r)* Floating Ext

Fetch the 32-bit IEEE single-precision number at *sf-addr*, convert to internal representation, and place on the floating-point stack, rounding if the internal representation has less than 32-bit precision. "S-F-fetch"

3.8.6 Floating-Point Stack Operators

A set of floating-point stack operators is provided, corresponding generally to

the operators for the integer data stack. Operators are also provided for exchanging values between the data and floating-point stacks. Before coding complicated floating-point stack maneuvers, check your particular system's maximum floating-point stack depth; it may be small. On systems that keep floating-point numbers on the data stack, take care with the order of floating-point and integer operations. These operators function as described, even if the system implements the floating-point stack on the data stack.

Glossary

D>F *(d —); (F: — r)* Floating

Convert a double-precision integer *d* to internal floating-point representation *r* and place on the floating-point stack. "D-to-F"

F>D *(— d); (F: r —)* Floating

Convert a floating-point number *r* to a double-precision integer *d*, discarding the fractional part, and place on the data stack. "F-to-D"

FDEPTH *(— +n)* Floating

Return *+n*, the number of values on the floating-point stack. If floating-point numbers are kept on the data stack, *+n* is the maximum number of possible floating-point values, given the current data stack depth in cells. "F-depth"

FDROP *(F: r —)* Floating

Drop the top item on the floating-point stack. "F-drop"

FDUP *(F: r — r r)* Floating

Duplicate the top item on the floating-point stack. "F-dupe"

FOVER *(F: r_1 r_2 — r_1 r_2 r_1)* Floating

Copy r_1 to the top of the floating-point stack. "F-over"

FROT *(F: r_1 r_2 r_3 — r_2 r_3 r_1)* Floating

Rotate the top three items on the floating-point stack. "F-rote"

FSWAP *(F: r_1 r_2 — r_2 r_1)* Floating

Exchange the top two items on the floating-point stack. "F-swap"

3.8.7 Floating-Point Arithmetic

The words in this section implement arithmetic on the floating-point stack. All operations are carried out to the full precision of the implementation-specific representation of a floating-point number.

Glossary

F* (*F:* r_1 r_2 — r_3) Floating
Multiply r_1 by r_2, giving r_3. "F-star"

F** (*F:* r_1 r_2 — r_3) Floating Ext
Raise r_1 to the power r_2, giving the result r_3. "F-star-star"

F+ (*F:* r_1 r_2 — r_3) Floating
Add r_1 to r_2, giving the sum r_3. "f-plus"

F- (*F:* r_1 r_2 — r_3) Floating
Subtract r_2 from r_1, giving the difference r_3. "F-minus"

F/ (*F:* r_1 r_2 — r_3) Floating
Divide r_1 by r_2, giving the quotient r_3. "F-slash"

FABS (*F:* r_1 — r_2) Floating Ext
Return r_2, the absolute value of r_1. "F-abs"

FLOOR (*F:* r_1 — r_2) Floating Ext
Round r_1 to an integral value, rounding toward negative infinity, giving r_2. "floor"

FMAX (*F:* r_1 r_2 — r_3) Floating
Return r_3, the greater of r_1 and r_2. "F-max"

FMIN (*F:* r_1 r_2 — r_3) Floating
Return r_3, the lesser of r_1 and r_2. "F-min"

FNEGATE (*F:* r_1 — r_2) Floating
Return r_2, the negation of r_1. "F-negate"

FROUND (F: r_1 — r_2) Floating

Round r_1 to the nearest integral value, giving r_2. "F-round"

FSQRT (F: r_1 — r_2) Floating Ext

Return r_2, the square root of r_1. An error may occur if r_1 is less than zero. "F-square-root"

3.8.8 Floating-Point Conditionals

Conditional tests of floating-point numbers consume their argument(s) on the floating-point stack and return a truth flag to the data stack. The word **F~** provides both exact and near-equality testing. **F~** is usually preferable to **F0=**, since a floating-point number may fail to be zero by an infinitesimal amount.

Glossary

F0< (— *flag*); (F: r —) Floating

Return *true* if and only if *r* is less than zero. "F-zero-less-than"

F0= (— *flag*); (F: r —) Floating

Return *true* if and only if *r* is exactly equal to zero. (See the definition of **F~** below.) "F-zero-equals"

F< (— *flag*); (F: r_1 r_2 —) Floating

Return *true* if and only if r_1 is less than r_2. "F-less-than"

F~ (— *flag*); (F: r_1 r_2 r_3 —) Floating Ext

Test for equality or near equality, on an absolute or relative basis. If the increment r_3 is positive, return *true* if and only if the absolute value of [r_1 - r_2] is less than r_3. If the increment r_3 is zero, return *true* if and only if r_1 and r_2 are exactly identical (be aware that some implementations may encode positive zero and negative zero differently). If the increment r_3 is negative, return *true* if and only if the absolute value of [r_1 - r_2] is less than the absolute value of r_3 times the sum of the absolute values of r_1 and r_2. "F-proximate"

3.8.9 Logarithmic and Trigonometric Functions

The words in this section provide a full set of logarithmic, exponential, and trigonometric functions. All angles are in radians. The function **FSINCOS** is a little unusual; it returns the sine *and* the cosine of the given angle (cosine on top). **FSINCOS** and **FATAN2** are complementary operators that convert angles to 2-vectors and vice versa. They correctly handle the conversion even when the tangent of the angle would be infinite. The pair of values returned by **FSINCOS** are a Cartesian unit 2-vector in the direction of the given angle, measured counter-clockwise from the positive X-axis. **FATAN2** takes arguments in the same order, converting a 2-vector back to a scalar angle; for all principal angles (-pi to +pi radians), the phrase **FSINCOS FATAN2** is an identity operation within the accuracy and range of the operators. The phrase **FSINCOS F/** is functionally equivalent to **FTAN**, but is useful only over a limited range of angles, whereas **FSINCOS** and **FATAN2** are useful for all angles.

Glossary

FACOS (F: r_1 — r_2) Floating Ext
Return r_2, the principal radian angle (zero to +pi) whose cosine is r_1. "F-A-cos"

FACOSH (F: r_1 — r_2) Floating Ext
Return r_2, the floating-point value whose hyperbolic cosine is r_1. "F-A-cosh"

FALOG (F: r_1 — r_2) Floating Ext
Raise 10 to the power r_1, giving r_2. "F-A-log"

FASIN (F: r_1 — r_2) Floating Ext
Return r_2, the principal radian angle (-pi/2 to +pi/2) whose sine is r_1. "F-A-sine"

FASINH (F: r_1 — r_2) Floating Ext
Return r_2, the floating-point value whose hyperbolic sine is r_1. "F-A-sine-H"

FATAN (F: r_1 — r_2) Floating Ext
Return r_2, the principal radian angle (-pi/2 to +pi/2) whose tangent is r_1. "F-A-tan"

FATAN2 (F: r_1 r_2 — r_3) Floating Ext
Return r_3, the principal radian angle (-pi to +pi) whose tangent is r_1/r_2. The

values r_1 and r_2 may be, but need not be, components of a unit vector. An error will occur if both r_1 and r_2 are zero (vector of zero magnitude). "F-A-tan-two"

FATANH (F: r_1 — r_2) Floating Ext

Return r_2, the floating-point value whose hyperbolic tangent is r_1. "F-A-tan-H"

FCOS (F: r_1 — r_2) Floating Ext

Return r_2, the cosine of the radian angle r_1. "F-cos"

FCOSH (F: r_1 — r_2) Floating Ext

Return r_2, the hyperbolic cosine of r_1. "F-cosh"

FEXP (F: r_1 — r_2) Floating Ext

Raise e (2.71828…) to the power r_1, giving r_2. "F-E-X-P"

FEXPM1 (F: r_1 — r_2) Floating Ext

Raise e (2.71828…) to the power r_1 and subtract one, giving r_2. This function provides increased accuracy over **FEXP** when the argument r_1 is close to zero. "F-E-X-P-M-one"

FLN (F: r_1 — r_2) Floating Ext

Return r_2, the natural logarithm of r_1. "F-L-N"

FLNP1 (F: r_1 — r_2) Floating Ext

Return r_2, the natural logarithm of $(1 + r_1)$. This function provides increased accuracy over **FLN** when its argument r_1 is close to zero. "F-L-N-P-one"

FLOG (F: r_1 — r_2) Floating Ext

Return r_2, the base-ten logarithm of r_1. "F-log"

FSIN (F: r_1 — r_2) Floating Ext

Return r_2, the sine of the radian angle r_1. "F-sine"

FSINCOS (F: r_1 — r_2 r_3) Floating Ext

Return r_2 (sine) and r_3 (cosine) of the radian angle r_1. "F-sine-cos"

FSINH (F: r_1 — r_2) Floating Ext

Return r_2, the hyperbolic sine of r_1. "F-sine-H"

FTAN (F: r_1 — r_2) Floating Ext

Return r_2, the tangent of the radian angle r_1. "F-tan"

FTANH (F: r_1 — r_2) Floating Ext

Return r_2, the hyperbolic tangent of r_1. "F-tan-H"

3.8.10 Address Management

The floating-point command set introduces three new data types: internal floating point, 32-bit IEEE single-precision floating point, and 64-bit IEEE double-precision floating point. An application creating data structures using any of these types should use the support words described in this section to manage the address space. For example, the length of an internal floating-point number should always be referred to indirectly with words such as **FLOAT+** or **FLOATS**, since the size may vary in different implementations.

When defining custom data structures, be aware that **CREATE** does not necessarily leave the data-space pointer aligned for the various floating-point data types. You can ensure alignment by explicitly specifying it both at compile time and execution time. An example from Standard Forth is:

```
: FCONSTANT ( F: r -- )   CREATE  FALIGN  HERE
  1 FLOATS ALLOT  F!   DOES> ( F: - r)   FALIGNED  F@ ;
```

In this example, the **FALIGN** after **CREATE** ensures that the address returned by **HERE** is float-aligned for the **F!** operation. **FALIGN** may have needed to reserve extra data space to do this, so, when an example of **FCONSTANT** is executed (using the code following **DOES>**) and the example's address is returned, the word **FALIGNED** is needed to skip over that same space (if any) and access the floating-point value properly with **F@**.

In many implementations, alignment of floating-point data types requires nothing more than ordinary cell alignment, in which case, words such as **FALIGN** and **FALIGNED** may simply be aliases for **ALIGN** and **ALIGNED**. An application should not rely on this equivalence, however, and should use the floating-point words in this section.

Glossary

FALIGN *(—)* Floating

If the data-space pointer is not float aligned, reserve enough data space to make it so. "F-align"

FALIGNED *(addr — f-addr)* Floating

Return *f-addr*, the first float-aligned address equal to or greater than *addr*. "F-aligned"

FLOAT+ *($f\text{-}addr_1$ — $f\text{-}addr_2$)* Floating

Add the size in bytes of a floating-point number to $f\text{-}addr_1$, giving $f\text{-}addr_2$. "float-plus"

FLOATS *(n_1 — n_2)* Floating

Return n_2, the size in bytes of n_1 internal floating-point numbers.

DFALIGN *(—)* Floating Ext

If the data-space pointer is not double-float aligned, reserve enough data space to make it so. "D-F-align"

DFALIGNED *(addr — df-addr)* Floating Ext

Return *df-addr*, the first double-float-aligned address equal to or greater than *addr*. "D-F-aligned"

DFLOAT+ *($df\text{-}addr_1$ — $df\text{-}addr_2$)* Floating Ext

Add the size in bytes of a 64-bit IEEE double-precision floating-point number to $df\text{-}addr_1$, giving $df\text{-}addr_2$. "D-float-plus"

DFLOATS *(n_1 — n_2)* Floating Ext

Return n_2, the size in bytes of n_1 64-bit IEEE double-precision floating-point numbers. "D-floats"

SFALIGN *(—)* Floating Ext

If the data-space pointer is not single-float aligned, reserve enough data space to make it so. "S-F-align"

SFALIGNED *(addr — sf-addr)* Floating Ext

Return *sf-addr*, the first single-float-aligned address equal to or greater than *addr*. "S-F-aligned"

SFLOAT+ *(sf-addr$_1$ — sf-addr$_2$)* Floating Ext

Add the size in bytes of a 32-bit IEEE single-precision floating-point number to *sf-addr$_1$*, giving *sf-addr$_2$*. "S-float-plus"

SFLOATS *(n$_1$ — n$_2$)* Floating Ext

Return n_2, the size in bytes of n_1 32-bit IEEE single-precision floating-point numbers. "S-floats"

3.8.11 Custom I/O

The input number conversion routines in the text interpreter and the standard output words **F.**, **FE.**, and **FS.** can be used for most floating-point I/O, but there are cases where more control over the process is desired. The words **>FLOAT** (for input) and **REPRESENT** (for output) can be used as the basis for custom I/O routines. The input word **>FLOAT** is more flexible than the text interpreter routines (for example, an exponent marker **E** or **e** is not required), but it cannot vector between integers and floating-point numbers; it assumes that the input string is to be converted as a floating-point number, if at all possible. **>FLOAT** is defined broadly to permit valid floating-point input from many standard programming environments.

Glossary

>FLOAT *(c-addr u — true | false); (F: — r |)* Floating

Attempt to convert the string specified by starting address *c-addr* and length *u* to internal floating-point representation. If the conversion is successful, its floating-point value *r* and *true* are returned. If it was not successful, only *false* is returned. A string of blanks should be converted as floating-point zero. "to-float"

Nearly any reasonably constructed string will convert. Decimal base is assumed. A valid number has a significand and optional exponent. The significand has an optional sign and at least one digit, with or without a decimal point. The exponent, if present, is signified by **E**, **e**, **D**, or **d** followed by an optional integer (signed or unsigned), or by a plain **+** or **-** followed by an optional integer. **>FLOAT** will convert all the following to valid floating-point numbers:

```
-1.23e-01  -1  .3  5D  -6.12+34  7+3  .456-  .006  9+00
```

REPRESENT *(c-addr u — n $flag_1$ $flag_2$); (F: r —)*[*] Floating

Attempt to convert the significand of the floating-point number *r*. At *c-addr*, place an ASCII representation of the *u* most-significant digits of the significand. The string is to be interpreted as a decimal fraction, with an implied decimal point to the left of the first digit; the first digit is zero only if all digits are zero. Return on the stack the resulting decimal base exponent *n*, the sign of the floating-point number as $flag_1$ (true if *r* was negative) and a valid-result $flag_2$ (*true* if *r* was in the implementation-defined range of valid floating-point numbers). The significand is rounded to *u* digits following the round-to-nearest-integer rule, and *n* is adjusted as necessary after the rounding.

If $flag_2$ is false, *n*, $flag_1$, and the contents of *c-addr* are implementation specific. However, the string at *c-addr* shall consist of displayable characters. For example, a system might return the informative messages `+infinity` or `nan` ("not a number") to *c-addr*.

* For single-stack systems, the order of the input arguments to **REPRESENT** is *r* on the bottom followed by *c-addr* and *u* (top).

4. The Forth Interpreter and Compiler

Forth is primarily a development environment, usually presented as a fairly complete "integrated development environment" including programming tools, libraries, compiler, assembler, and (in many cases) an editor. This section describes features specific to the Forth compiler as well as Forth's uniquely powerful ability to construct data objects of various types. Unlike more conventional compilers, the Forth compiler uses a text interpreter that is also available to the programmer for application use. Moreover, the tools used inside the Forth compiler and interpreter are available so that you can modify and extend both.

4.1 The Text Interpreter

The text interpreter in Forth is used for terminal interaction and for processing text on disk (either in direct execution or in compilation). A brief description of its operation was given in Section 1.1.5. This section covers the text interpreter in more detail and discusses ways the programmer may use the text interpreter in application routines.

References Text interpreter, Section 1.1.5

4.1.1 Input Sources

The text interpreter always interprets from an *input buffer (*also called an *input stream)*, which may or may not be a physically separate location. There are up to four sources for input: the *user input device* (almost always a keyboard), a character string, a text file, and a block file. The default source is the keyboard, and all systems have a keyboard input buffer (typically 80 characters long).

All systems can also treat a character string in memory as an input buffer, if given the string's address and length. When systems with source code in text files interpret from files, the current line in the current file is the input buffer. When systems with source code in blocks interpret from blocks, the current block (1024 bytes) is the input buffer.

The word **SOURCE-ID** returns a value that identifies the input source, unless it is a block. On systems that contain blocks, the variable **BLK** contains the block number that is the current input source, or zero if the input is not a block. On systems with all four inputs, checking **BLK** first and then **SOURCE-ID** will uniquely identify the input. See Table 7 below:

Table 7: Identifying the input source

Input Source	SOURCE-ID returns:	BLK @ returns:
User Input Device (keyboard)	0	0
Character String	-1	0
File	Text file *fileid*	0
Block	(undefined)	block number

Glossary

BLK *(— a-addr)* Block

Return the address of a cell containing zero or the number of the mass-storage block being interpreted. If **BLK** contains zero, the input source can be identified by **SOURCE-ID**. "B-L-K"

SOURCE-ID *(— n)* Core Ext, File

Return a value indicating the current input source. The value is 0 if the source is the user input device, -1 if the source is a character string, a *fileid* if the source is a file, and undefined if the source is a block.

References Block-based disk access, Section 3.4
File-based disk access, Section 3.5

4.1.2 Source Selection and Parsing

The word **QUIT** is the basic idle behavior of the terminal task that controls the user input device. Executing **QUIT** makes the user input device the current input source, and awaits a line of input into the keyboard buffer. When this is received, the character pointer **>IN** is set to zero and interpretation begins. If interpretation completes normally, the system-defined prompt is displayed (typically OK), and **QUIT** awaits the next line of input.

EVALUATE directs interpretation to take place from a specified character string. When **EVALUATE** is executed, the address and count of a character string must be on the stack. **EVALUATE** saves the current input source specification, makes the character string the input buffer, sets **>IN** to zero and begins interpretation. When the parse area is empty (there are no more words to be interpreted in the string), the prior input source is restored.

Interpretation from a file usually is done with **INCLUDE-FILE**, **INCLUDE**, or **INCLUDED**. These and other file-handling words are described in detail in Section 3.5 of this manual.

Interpretation from blocks is done with **LOAD** or **THRU**. See Section 3.4 for details of block reading and writing.

The support words in the following list are connected with text interpretation. In general, they are used at the system level to create custom text interpretation words, and will not be needed by an application; for example, all standard source-selection words—such as **EVALUATE**, **INCLUDE**, and **LOAD**—automatically save and restore the current input source specification. Some lower-level words—such as **READ-FILE** and **READ-LINE**—do not, and might need explicit uses of **SAVE-INPUT** and **RESTORE-INPUT**.

Glossary

>IN *(— a-addr)* Core

Return the address of a cell containing the offset (in characters) from the start of the input buffer to the start of the parse area. "to-in"

EVALUATE *(i*x c-addr u — j*x)* Core, Block

Save the current input source specification. Set **SOURCE-ID** to -1. Make the

string at *c-addr*, length *u*, the input source and input buffer, set **>IN** to zero, and interpret. When the parse area is empty, restore the prior input source specification. Other stack effects are due to the word(s) that were **EVALUATE**d.

PARSE *(char — c-addr u)* Core Ext

Parse the input buffer until the delimiter *char* is encountered. Return the address and length of the parsed string. If the parse area was empty, *u* is zero and *c-addr* is undefined.

QUIT *(i*x —); (R: j*x —)* Core

Terminate execution of the current word (and all words that called it). Clear the return and data stacks. No indication is given to the terminal that a **QUIT** has occurred. Enter interpretation state and begin an infinite loop of awaiting a line of text from the input source and interpreting it. **QUIT** is the default idle behavior for terminals.

REFILL *(— flag)* Block Ext, Core Ext, File Ext

Attempt to fill the input buffer from the input source, returning a *flag* that is *true* if successful. If no input is available from the current source, return *false*.

If the input source is the keyboard, await a line of input. If successful (a line of zero characters—i.e., only CR was pressed—is successful), set **>IN** to zero and return *true*.

If the input source is a string from **EVALUATE**, return *false* and take no other action.

If the input source is a block, make the next block the input source and buffer by adding one to **BLK** and setting **>IN** to zero. Return *true* if the new value of **BLK** is a valid block number, otherwise *false*.

If the input source is a text file, attempt to read the next line from the file. If successful, make the result the current input buffer, set **>IN** to zero, and return *true*; otherwise, return *false*.

RESTORE-INPUT *(x_n … x_1 n — flag)* Core Ext

Attempt to restore the input source specification to the state described by the parameters on the stack. The number and content of the parameters are system dependent. Return *true* if the input source *cannot* be so restored. It is an error if the input source represented by the arguments is not the same as the current input source (i.e., **SAVE-INPUT** and **RESTORE-INPUT** are intended for re-posi-

tioning within a given source, not switching between sources).

SAVE-INPUT *(— x_n … x_1 n)* Core Ext

Save *n* parameters (plus *n* itself), which describe the current state of the input source specification, for later use by **RESTORE-INPUT**. The number and content of the parameters are system dependent. The parameters will include the value of **>IN** and others that are input-source dependent.

SOURCE *(— c-addr u)* Core

Return the address and length of the input buffer.

References Text interpretation and **>IN**, Section 1.1.5

4.1.3 Dictionary Searches

It must be possible to look up words and their definitions in the dictionary. Forth provides several words to do this, each of which performs a search and returns information about a word, typically its *execution token*. These searches are used in the text interpreter and colon compiler.

The word **'** ("tick") performs a dictionary search for the word that immediately follows it in the current input stream.

The phrase:

```
' <name>
```

when typed at a terminal or executed interpretively in source text, pushes onto the stack the execution token of *name* if *name* can be found in the dictionary. If *name* cannot be found, an abort will occur with an error message such as:

```
<name> ?
```

Since the precise definition of the "execution token" returned from dictionary searches varies, depending upon the implementation, the word **>BODY** is provided. Given an execution token, it will always return the parameter field (content) address. On many systems **>BODY** is a no-op.

The most common uses of **'** for dictionary searches are:

- To learn whether a word has been defined.
- To find the location of a word, using **>BODY** (for example, to **DUMP** its contents).
- To obtain the location of a data object (again, using **>BODY**) whose run-time behavior is other than returning its address.

' is not immediate. That is, if you wish to *compile* a reference to an address, as in the third item listed above, you must use **[']**. The word **'** always takes its operands from the current input stream at the time it is executed.

The following are dictionary search words:

Glossary

' <name> *(— xt)* Core

Search the dictionary for *name*. If *name* is found, **'** return the word's execution token; otherwise, abort. "tick"

['] <name> *(—)* Core

Like **'** but must be used in a colon definition. **[']** finds the next word in the dictionary and compiles its execution token as a literal. If *name* is not in the dictionary, **[']** aborts. **[']** calls **LITERAL**. **[']** is an **IMMEDIATE** word (executed, rather than compiled by the colon compiler; see the references section). "bracket-tick"

>BODY *(xt — a-addr)* Core

Given a word's execution token, return the address of the start of the parameter field in that word. "to-body"

FIND *(c-addr — c-addr 0 | xt 1 | xt -1)* Core, Search

Attempt to find a definition whose name is in a counted string at *c-addr*. If the definition is not found, return the address and zero; if the definition is found, return its execution token. If the definition is immediate, also return +1; otherwise, return -1.

References

['], Section 4.3.6
IMMEDIATE words, Section 4.4.1
Word lists, Section 4.6
WORD, Section 4.1.5.1

4.1.4 Input Number Conversion

Wherever possible, an application should be designed to take advantage of Forth's interactive nature. Thus, a hypothetical word **SCANS** whose function is to perform some user-specified number of scans (an application function) should expect only its parameter on the stack. Then, to perform 100 scans, the user could type:

```
100 SCANS
```

Such usage is natural and convenient for the operator, and requires no special programming to handle the input parameter.

There are occasions in which normal Forth syntax is inadequate. Some examples include:

- Parsing a text string that comes from a source other than a terminal, such as magnetic tape.
- Entry of numbers that must be in double-precision but are not punctuated (i.e., zip codes).
- Entry of numbers that must follow, rather than precede, the command.

Forth provides several words to enable the user to handle input numbers in a variety of circumstances. This section describes these methods.

>NUMBER is the basic input number conversion routine. If it encounters any non-numeric digit during the conversion, it stops with a pointer to the digit, rather than aborting. For this reason, **>NUMBER** is often used when a number is input by a program directly, without using the text interpreter.

>NUMBER expects a double-precision integer, and the address and count of the input string. It leaves a double-precision integer (the result of the conversion), and an address and count. The initial address into **>NUMBER** must point to the first digit of the string of numerals. The initial double-precision number is usually set to zero.

After **>NUMBER** stops, the address in the second stack item is the address of the first non-numeric character **>NUMBER** encountered or, if the string was entirely converted, is the first character past the end of the string. The double-precision integer will contain data from all digits converted thus far.

An example of the use of **>NUMBER** is:

```
: INPUT ( -- n)   PAD 5 BLANK  PAD 5 ACCEPT >R
      0. PAD R> >NUMBER  2DROP DROP ;
```

This definition initializes a region of **PAD** to blanks, and awaits up to five digits which will be stored there. **0.** provides an initial double-precision value, and **PAD R>** provides the address and actual count for **>NUMBER**. The **2DROP DROP** discards the address and count returned by **>NUMBER** and the high-order part of the converted number.

INPUT will not convert input strings with a leading minus sign, because a minus is not a digit. If negative input is necessary, the above definition can be extended to check the character upon which conversion stopped to see if it is a minus sign and, if it is, start **>NUMBER** again and negate the result.

>NUMBER returns the address of the string's next byte, so **>NUMBER** may be called in a loop. The text interpreter's number conversion routine calls **>NUMBER** in just this way. An application similar to this is parsing a packet of data received over a communications line, or from a tape record in which numeric fields are separated by an arbitrary delimiter such as **//**. To skip such items, or to skip fields that are not of interest, the appropriate count of bytes may simply be added to the address, which is carried on the stack.

In some cases, numbers may be in fields of known length but not separated by any delimiter. In such cases, the best solution may be to use **CMOVE** to move groups of digits to **PAD**, where they may be converted easily by **>NUMBER**.

Glossary

>NUMBER (ud_1 *c-addr*$_1$ u_1 — ud_2 *c-addr*$_2$ u_2) Core

Convert the characters in the string at *c-addr*$_1$, whose length is u_1, into digits, using the radix in **BASE**. The first digit is added to ud_1. Subsequent digits are added to ud_1 after multiplying ud_1 by the number in **BASE**. Conversion continues until a non-convertible character (including an algebraic sign) is encountered or the string is entirely converted; the result is ud_2. *c-addr*$_2$ is the location of the first unconverted character or, if the entire string was converted, of the first character beyond the string. u_2 is the number of unconverted characters in the string. "to-number"

References Numeric input, Section 1.1.6
CMOVE, Section 2.3.3
ACCEPT, Section 3.3.1
PAD, Section 2.3.1

4.1.5 Character String Processing

Character strings may be received or transmitted by using words defined in Section 3.3. Such input or output is only possible, of course, where the terminal device is capable of supporting the required operation.

4.1.5.1 Scanning Characters to a Delimiter

WORD is the main work-horse of Forth's text interpreter. It fetches characters from the input stream, starting at the offset given by the user variable **>IN**, until reaching a specified delimiter. **WORD** takes its input from the current input source; this is normally the terminal input buffer. During the time text is being interpreted from a source code file (either block or text file), the current input source is the designated file.

WORD expects the delimiter character in the low-order byte of the top item that is on the stack. **WORD** skips any leading occurrences of this character, searching for a non-delimiter character; if one is found, it is placed in a temporary storage area. Succeeding characters are then moved into this area until a delimiter character is encountered or until the specified end of the string is reached, which terminates the operation. The area where the characters are placed is not initialized, although **WORD** will insert one trailing blank after the string.

The maximum length of a string depends on the input source. If the source is the keyboard, the maximum length is set by the size of the terminal input buffer. If the source is a block, the maximum length is 1024. If the source is a file, the string expires at the end of the file.

WORD places on the stack the address of the string. The first byte of this string contains the number of input characters in the string, up to the occurrence of the delimiter—this is convenient for words that often follow it, such as **>NUMBER**.

The storage space used by **WORD** may be used by other Forth functions as well,

such as output number conversion. As a result, when you use **WORD** to pick up a string from the input stream, you should finish working with it or promptly move it to another area (such as **PAD**) to avoid confusion.

As an example of **WORD**'s use, consider the following simple **TEST** example (**COUNT** is a standard word whose definition is shown here for convenience. Its principle use is to convert a counted string to a plain character string, returning address and count on the stack):

```
: COUNT ( addr1 -- addr2 n)   DUP CHAR+  SWAP C@ ;
: TEST   32 WORD  COUNT TYPE ;
```

TEST would be used in the following way:

```
TEST ABC  (carriage return)  ABC ok
```

Because using a space for a delimiter is so common, the word **BL** (for *blank*) is defined, which returns the ASCII value for a space. Thus, phrases such as **32 WORD** can be replaced by **BL WORD**, which is more readable.

COUNT may also be used with plain strings. Successive calls to **COUNT** will "walk" through the string, returning each character and incrementing the address.

Glossary

>IN *(— a-addr)* Core

In many systems, this is the name given to the text interpreter pointer. It returns *a-addr*, the address of a cell containing the offset in characters from the start of the input buffer to the start of the current parse area. "to-in"

BL *(— char)* Core

Return *char*, the ASCII character value for a space (20_H). "B-L"

COUNT *($c\text{-}addr_1$ — $c\text{-}addr_2$ n)* Core

Return the length *n* and address $c\text{-}addr_2$ of the text portion of a counted string beginning at $c\text{-}addr_1$.

WORD <text> *(char — c-addr)* Core

Skip any leading occurrences of the delimiter *char*. Parse *text* delimited by *char*. Return *c-addr*, the address of a temporary location containing the parsed text as

a counted string. If the parse area was empty or contained only delimiter(s), the resulting string length is zero.

References Fetching input characters to **PAD**, Section 3.3.1
>NUMBER, Section 4.1.4
Text interpreter, Section 1.1.5
Character string output (**TYPE**), Section 3.3.2

4.1.5.2 Compiling and Interpreting Strings

There are two cases in which it is desirable to have text messages compiled in programs: to issue error messages and to communicate information during normal operation. The words in the following list provide support for compiled strings. These words may be used only inside a definition (except **S"**). In all cases, a quote mark delimits the end of the string.

S" also may be executed interpretively, if you need the address and count of a string outside of a definition. For example, **INCLUDED** loads a file, given the address and count of a string on the stack containing the filename. The syntax would be:

```
… S" <filename>" INCLUDED …
```

On many implementations, however, an interpreted **S"** uses a single buffer to hold the string. Therefore, successive uses of **S"** may over-write the buffer from a previous use.

Each of these words has functions to be performed both at compile time and at execute time. At compile time the address of the execute-time function is compiled, along with the string. At execute time the behavior differs. For **S"**, the address and length of the string must be pushed on the stack. For **ABORT"**, the test must be performed. For both **."** and **ABORT"**, the string must be typed out. The stack notation for the words below refers to the execution-time behavior.

Glossary

S" <string>" *(— c-addr u)* Core, File
Compile the following *string*, terminated by **"**. At run time, the address and length of the string (two stack items) will be pushed on the stack. "S-quote"

For example:

```
: "TEMP" ( n)   68 > IF
   S" WARM "  ELSE  S" COOL "
   THEN TYPE ;
```

This will display the message `WARM` if the temperature value on the stack is greater than 68, and will display `COOL` otherwise.

C" <string>**"** *(— c-addr)* Core Ext

Same as **S"**, but compiles a counted string (length stored in first byte). As with **S"**, the *string* is terminated by **"**. At run time, the address of the counted string (one stack item) will be pushed on the stack. "C-quote"

." <string>**"** *(—)* Core

Compile *string*, which will be typed when the word that contains it is executed. "dot-quote"

For example:

```
: GREETING   ." Hi there" ;
```

ABORT" <text>**"** *(i*x flag —); (R: j*x —)* Core, Exception Ext

Compile *text*, to be typed as an error message if the value on the stack, *flag*, is *true* when the phrase containing **ABORT"** is executed. "abort-quote"

If **ABORT"** finds its argument to be *true* (any non-zero value), it will echo the command most recently interpreted, issue the message, clear both data and return stacks, and return control to the operator. For example:

```
: CHECK ( n -- n)   1000 OVER <
   ABORT" TOO BIG" ;
```

References Error handling, Section 2.6
Compiling strings, Section 4.3.7

4.1.6 Text Interpreter Directives

It is useful to control the logical flow when compiling an application. You may wish, for example, to load a certain source code file only if a flag indicates the need for that file. A number of Forth words provide this kind of control. Although these words are almost always used outside of definitions, they are all **IMMEDIATE** (i.e., they will execute when encountered in compiling state) and so may be used in definitions, if needed.

Here are two examples:

```
<flag> [IF]  INCLUDE <file1>  [THEN]

[DEFINED] <word in file1> [IF]  INCLUDE <file2>
[ELSE] INCLUDE <file3> [THEN]
```

Glossary

[DEFINED] <name> *(— flag)* common usage
Search the dictionary for *name*. If *name* is found, return *true*; otherwise, return *false*. "bracket-defined"

[UNDEFINED] <name> *(— flag)* common usage
Search the dictionary for *name*. If the word is found, return *false*; otherwise, return *true*. "bracket-undefined"

[IF] *(flag —)* Tools Ext
Begin an interpretive branch. If the *flag* is *true*, do nothing (i.e., continue interpretation). If the *flag* is *false*, parse and discard words from the parse area (including nested occurrences of **[IF]** … **[THEN]** clauses) until either the word **[ELSE]** or the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, it is refilled as with **REFILL**. Because **[IF]** discards **[ELSE]** (if it is present) when *flag* is *false*, interpretation will continue after **[ELSE]**, thus interpreting the contents of the **[ELSE]** clause. "bracket-if"

[ELSE] *(—)* Tools Ext
Parse and discard words from the parse area (including nested occurrences of **[IF]** … **[THEN]** clauses) until the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, it is refilled as with **REFILL**.

[ELSE] is only executed if the *flag* for the associated **[IF]** was *true*; therefore, it always discards the words between **[ELSE]** and **[THEN]**. "bracket-else"

[THEN] *(—)* Tools Ext

Take no action. **[THEN]** has no function by itself, but must exist in the source code in order to mark the end of the parsing for **[IF]** or **[ELSE]**. "bracket-then"

References **REFILL**, Section 3.5.3

4.2 Defining Words

Forth provides a basic set of words used to define objects of various kinds. As with other features of Forth, the set of such commands may be expanded. Here we will present those which are standard in all Forth systems, exclusive of the defining words that are part of database support options and the assembler defining words (see your product documentation).

4.2.1 Creating a Dictionary Entry

A word is defined when an entry is created in the dictionary. **CREATE** is the basic word that does this; it may be used by **:**, **CODE**, **VARIABLE**, **CONSTANT**, and other defining words to perform the initial functions of setting up the dictionary entry. **CREATE** behaves as follows:

1. Memory is checked to see if a minimum amount remains. If not, there may be an abort. At the same time, the data-space pointer is aligned to an even cell address, if the system being used requires it.
2. **WORD** fetches the next word in the input stream. A dictionary entry is created for this word, with a pointer to the previous entry in this word list.
3. The code field of the new word is set to point to the run-time code of **CREATE**, which will push the address of this word's parameter field onto the stack when the word is executed. However, no data space is allocated by **CREATE**.

Other defining words that use **CREATE** may reset the new word's code field to define different run-time behavior by using the words **;CODE** or **DOES>**. Fig-

ure 9 shows an example of a dictionary entry built by **CREATE**.

CREATE is often used to mark the beginning of an array. The space for the rest of the array is reserved by incrementing the dictionary pointer with **ALLOT**, as in this example:

```
CREATE DATA   100 CELLS ALLOT
```

The example reserves a total of 100 cells for an array named **DATA**. When **DATA** is used in a colon definition, the address of the first byte of **DATA** will be pushed on the stack by the run-time behavior of **CREATE**. The array is not initialized. If you wish to set all the elements of the array to zero, you may use **ERASE** as in the following example:

```
DATA  100 CELLS ERASE
```

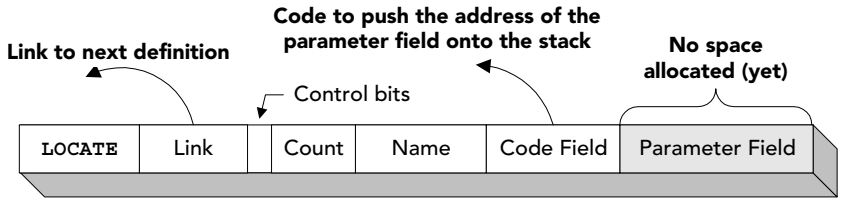

Figure 9. Dictionary entry built by CREATE

The word **UNUSED** places on the stack the number of bytes left in the memory area where dictionary entries are constructed. On some systems, this region of memory is also used for other purposes, with the dictionary starting at the bottom and growing towards high memory, and with something else starting at the top and growing towards low memory. On such systems, **UNUSED** may give different answers at different times, even though the dictionary pointer is not changed.

Glossary

ALLOT (*n* —) Core

If *n* is greater than zero, reserve *n* bytes of data space. If *n* is less than zero, release |*n*| bytes of data space. If the data-space pointer is initially aligned

and *n* is a multiple of the cell size, the data space pointer will remain aligned after the **ALLOT**.

CREATE <name> *(—)* Core

Construct a dictionary entry for *name*. Execution of *name* will return the address of its data space. No data space is allocated for *name*, however; this must be done by subsequent actions such as **ALLOT**.

UNUSED *(— u)* Core Ext

Return *u*, the number of bytes remaining in the memory area where dictionary entries are constructed.

References
:, Section 4.2.4
CODE, Sections 4.2.5, 5
CONSTANT, Section 4.2.3
ERASE, **BLANK**, **FILL**, Section 2.3.3
VARIABLE, Section 4.2.2

4.2.2 Variables

A **VARIABLE** is a named memory location whose value may be fetched onto the stack or stored into, with equal ease.

The definition of a **VARIABLE** takes the form:

```
VARIABLE <name>
```

This constructs a definition for *name*, with one cell allotted for a value. A single-cell value may be stored into the parameter field of the definition. For example:

```
VARIABLE DATA
6 DATA !
```

will store 6 in the parameter field of **DATA**.

When a **VARIABLE** is referenced by *name*, the address of its parameter field is pushed onto the stack. This address may be used with **@** or **!** to fetch or store, respectively, the variable's current value.

Similarly, the word **2VARIABLE** defines a variable whose parameter field is two cells long. Such a variable may contain one double-precision number or a pair of single-precision numbers (such as *x,y* coordinates), or even two unrelated values. **2VARIABLE** differs from **VARIABLE** only in the number of bytes allotted. The operators **2@** and **2!** are used with this format.

On some eight-bit and 16-bit CPUs, such as those used in embedded systems in which data space is limited, **CVARIABLE** defines a variable that is one byte long. The operators **C@** and **C!** are used with this format. Note that since **CVARIABLE** allots only one byte, it will leave the data space pointer unaligned. If you are concerned about alignment, you should either group **CVARIABLE**s so as to leave the space aligned, or use **ALIGN** afterwards.

In summary, to place the value of a **VARIABLE** on the stack, invoke its name and a fetch instruction. For example, you could type:

```
    <variable name> @
or  <variable name> 2@
```

To store a value into a variable, invoke its name and a store instruction. For example:

```
    <value> <variable name> !
or  <value1> <value2> <variable name> 2!
```

In a read-only-memory environment, **VARIABLE** is re-defined to allot space in read/write memory rather than in *name*'s parameter field; in this case, the assigned read/write memory address is compiled into the parameter field. The run-time behavior of a variable in ROM is to return the contents of its ROM parameter field (like a constant does); that value is the address of the variable's data space in RAM.

Glossary

VARIABLE <name> *(—)* Core

Define a single-cell variable. Execution of *name* will return the address of its data space.

2VARIABLE <name> *(—)* Double

Define a two-cell variable. Execution of *name* will return the address of its data

space. "two-variable"

CVARIABLE <name> (—) common usage
Define a one-byte variable. Execution of *name* will return the address of its data space. Typically available only on embedded systems. "C-variable"

References **@**, **!**, **2@**, and **2!**, Section 2.1.2
ALIGN, Section 4.3.2

4.2.3 CONSTANTs and VALUEs

The purpose of a **CONSTANT** is to provide a name for a value which is referenced often but may be changed seldom or never. There are both single- and double-precision versions. Figure 10 shows an example of a dictionary entry built by **CONSTANT**.

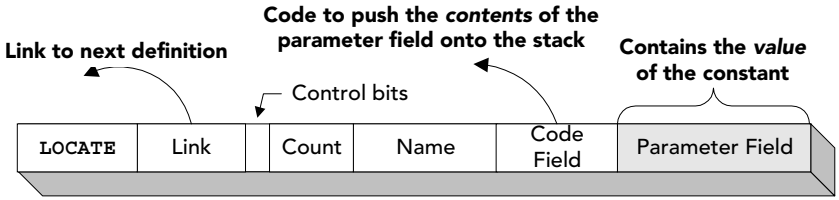

Figure 10. Dictionary entry built by CONSTANT

The syntax for defining constants is:

```
<value> CONSTANT <name>
```

For example, you may define:

```
1000 CONSTANT LIMIT
0 5000 2CONSTANT LIMITS
3141593. 2CONSTANT PI
```

When a **CONSTANT** is referenced by name, its value (not its address) is pushed onto the stack. Similarly, when a **2CONSTANT** is referenced, two stack items

are pushed onto the stack. In the case where a **2CONSTANT** is used for two values (as in **LIMITS**, above), the values are placed on the stack in the order specified (e.g., 5000 on top, 0 below). In the case of a double-precision number, the high-order part of the number is on top of the stack.

In order to change a **CONSTANT** you must first obtain its address. This is done by using **'** in interpretive mode or **[']** inside a colon definition, in either case followed by the name of the **CONSTANT** and **>BODY** to get its data space address. The command **!** will store into the address of a **CONSTANT** thus obtained, and **2!** stores into the fetched address of a **2CONSTANT**.

For example, you might type this:

```
100 CONSTANT SIZE
500 ' SIZE >BODY !
```

The first phrase creates a **CONSTANT** named **SIZE** whose value is 100. The second phrase changes the value to 500.

In many systems, such as those with a mix of RAM and ROM, you may not be permitted to store into constants at all. For example, the value of a **CONSTANT** may be in ROM.

The purpose of a **VALUE** is to provide a name for a single-precision value which is referenced often and but which may need to change. On systems with a mix of RAM and ROM, **VALUE**s are compiled into RAM. The procedure for defining values is to declare:

```
<initial value> VALUE <name>
```

For example, you may define:

```
1000 VALUE LIMIT
```

When a **VALUE** is referenced by *name*, its current value is pushed onto the stack. The word **TO** is used to change a value. The syntax is:

```
<new value> TO < name>
```

For example, you might type this:

```
1000 VALUE LIMIT   LIMIT .     500 TO LIMIT   VALUE .
```

The first phrase creates a **VALUE** named **LIMIT** whose value when defined is 1000. The second phrase changes the value to 500.

VALUE combines the convenience of a **CONSTANT**—it returns its value without requiring an explicit **@**—with the writeability of a **VARIABLE**.

Glossary

CONSTANT <name> (*x* —) Core

Define a single-precision constant *name* whose value is *x*.

2CONSTANT <name> (x_1 x_2 —) Core

Define a double-cell constant *name* whose value may be a double-precision number or a pair of single-precision numbers. "two-constant"

TO <name> (*x* —) Core Ext, Local

Store *x* in the data object *name*. An error will occur if *name* was not defined by **VALUE**.

VALUE <name> (*x* —) Core Ext

Define a single-precision data object *name* whose initial value is *x*.

4.2.4 Colon Definitions

The defining word **:** (colon) is discussed briefly in Section 1.1.7, and numerous examples appear in other sections. In this section, we describe the use and behavior of this important defining word.

The basic form of a **:** definition is:

```
: <name>   <action> ;
```

When the colon is executed, the system enters a compilation state. A dictionary entry is created for the word *name*. *action* represents a list of previously defined words that will be executed in sequence whenever *name* is invoked. The **;** terminates the definition and returns the system to interpretation state.

The variable **STATE** contains the compilation-state flag. The value of **STATE** is *true* (non-zero) when compiling (e.g., between **:** and **;**), and is *false* (zero) when

interpreting. **STATE** is only changed by the following seven Standard words: **:**, **;**, **ABORT**, **QUIT**, **:NONAME**, **[**, and **]**. Programs that comply with Standard Forth may not modify **STATE** directly.

Each of the words **:** and **;** has two types of behavior, one for compile time and another for run time.

At compile time, **:** constructs a dictionary entry (e.g., by using **CREATE**) and begins compiling. It also *smudges* the name so the word will not inadvertently compile a reference to itself. (The word **RECURSE** may be used where the definition of a word must call itself.)

The run-time behavior of a word defined by **:** is to execute the words whose execution tokens form the body of the definition.

The **;** ends compilation and compiles the run-time code for **;** (the word **EXIT** on most indirect-threaded implementations). This code pops the address on top of the return stack into the VM's instruction pointer. The effect is to return to the calling environment.

Most of the words that make up the content of a definition are not executed during compilation; instead, references to them are compiled in the parameter field of the definition. The exception to this procedure are the words which are compiler directives or literals. These generally have both compile-time and run-time behaviors, just as **:** and **;** do.

Every colon definition requires a minimum of three components: a colon, a name, and a semicolon. Such a minimum definition (commonly called a *null definition*) will execute properly, but will do no work. It does have useful purposes; for example, to provide *placeholder* definitions for routines to be written later, or to mark a location in the dictionary as the beginning of an overlay area.

It is possible to create high-level Forth definitions without associated names. This is an advanced technique, not commonly used in a Forth application (but see the example in Section 2.5.6). The word that makes nameless definitions is **:NONAME**, and the syntax is simply:

```
:NONAME   <action> ;
```

A piece of code created this way is an isolated fragment; it cannot be found in the dictionary and has no referenceable name to cause it to execute. To obtain

access to it, a **:NONAME** definition returns its execution token on the stack at the time it is created. The compiling program must take action *at that time* to store the execution token in a useful place, such as in a table or other data structure. **:NONAME** is mainly used to build definitions attached (via their execution tokens) to special mechanisms such as execution vectors or a push buttons.

Glossary

: <name> *(—)* Core

Create a definition for *name*, called a *colon definition*. Enter compilation state and start compiling the definition. The execution behavior of *name* will be determined by the previously defined words that follow, which are compiled into the body of the definition. *name* cannot be found in the dictionary until the definition is ended. At execution time, the stack effects of *name* depend on its behavior. "colon"

:NONAME *(— xt)* Core Ext

Create an execution token *xt* and place it on the stack. Enter compilation state and start compiling the definition. The execution behavior of *xt* will be determined by the words compiled into the body of the definition. This definition may be executed later by the phrase *xt* **EXECUTE**. "colon-no-name"

; *(—)* Core

End the current definition and enter interpretation state. If the data-space pointer is not aligned, reserve enough space to align it. "semi-colon"

RECURSE *(—)* Core

Append the execution behavior of the current definition to the current definition, so that it calls itself recursively.

References

Forth virtual machine, Section 1.1.7
Alignment, Section 4.3.2
Compiler directives, Section 4.4
EXIT, Section 2.5.5
EXECUTE, Section 2.5.6
Overlays, Section 4.5
Program structures, Section 2.5
STATE, Section 4.3.3

4.2.5 Code Definitions

The form of a **CODE** definition is:

```
CODE <name>  <assembler instructions>  <code-ending>
```

The word **CODE** performs the following functions at assembly time:

1. Constructs a standard dictionary entry for *name* using **CREATE**.
2. Sets the execution token for *name* to point to *name*'s parameter field.
3. Selects the **ASSEMBLER** word list.

The words used inside a **CODE** definition are executed directly, with the effect of assembling machine instructions into the parameter field of the word being defined. There is nothing analogous to the compilation state that exists between **:** and **;**. When high-level Forth words are encountered, they are executed directly as well—thus, when used in a **CODE** definition, words such as **SWAP** and **DUP** manipulate the stack during assembly.

Macros can be defined as colon definitions containing assembler words, provided you first select the **ASSEMBLER** word list. This works because of the normal consequence of putting executable words in a colon definition: they will be executed when the definition is executed. Thus, one 8051 assembler defines:

```
\ Subtract without borrow
: SUB ( r1 r2 -- )   C CLRB   SUBB ;
```

The new "mnemonic," used in the form:

```
<r1> <r2> SUB
```

will assemble instructions that clear the carry bit before subtracting r_2 from r_1.

The basic principles of Forth assemblers are covered in Section 5. Assembler mnemonics, addressing modes, and conventions are covered in the documentation for your Forth system.

References Assembler code endings, Section 5.2

4.2.6 Custom Defining Words

One of the most powerful capabilities in Forth is the ability to define new defining words. Thus, the programmer may create new data types with characteristics peculiar to the application, new generic types of words, and even new classes of words with a specified behavior that is common to each class.

In creating a custom defining word, the programmer must specify two separate behaviors:

- The compile-time behavior of the defining word (creating the dictionary entry, compiling parameters, etc.).
- The run-time behavior (the action to be performed by words created by the new defining word).

In the cases discussed in the next two sections, compile-time behavior is described in high-level Forth. Several methods for specifying run-time behavior are also discussed.

4.2.6.1 Basic Principles of Defining Words

There are two ways to create new defining words in Forth. In the one case (using **DOES>**), the run-time behavior is described in high-level Forth; in the other (using **;CODE**), the run-time behavior is described in assembler code. The basic principles are the same.

In Forth, a *defining word* will create a new dictionary entry when executed. All words defined by the same defining word share a common compile-time and run-time behavior. For example, **VARIABLE** is a defining word; all words defined by **VARIABLE** share two common characteristics:

- Each has one cell allotted in which a value may be stored. These bytes may be initialized to zero in some systems.
- When executed, each of these words will push onto the stack the address of this one-cell reserved area.

On the other hand, all words defined by **CONSTANT**, which is another defining word, share two other behaviors:

- Each has compiled into its parameter field a single-precision value, which was on the stack when **CONSTANT** was executed.
- When a word defined by **CONSTANT** executes, it puts its value on the stack.

In each of these examples, the first behavior (the compile-time action) relates to the physical construction of the word, which is determined when the word is compiled. The second behavior describes what all defined words of that type do when executed. All defining words must have a compile-time behavior and a run-time behavior.

The general definition of a defining word looks like:

```
: <name>   <compile-time behavior>   <transition word>
   <run-time behavior> <ending>
```

The *transition word* ends the specification of compile-time behavior and begins the specification of run-time behavior. There are two such transition words: **;CODE** begins run-time behavior described in code (assembler), and **DOES>** begins run-time behavior described in high-level Forth. Each of the transition words requires a different ending; in the case of **DOES>**, it is **;** (semi-colon); in the case of **;CODE**, it is an implementation-defined code ending.

The exact behavior of these two words is discussed in the following sections. The description of compile-time behavior is the same, regardless of which transition word is used. In fact, if you change the transition word and run-time behavior from **DOES>** plus high-level to **;CODE** plus equivalent code, no change to the compile-time behavior is necessary.

The compile-time portion of a defining word must contain **CREATE** (or a defining word that calls **CREATE**) to create the dictionary entry. If one or more parameters are to be compiled, or if space for variable data is to be allocated, it is convenient to use a previously defined defining word which takes care of that.

Every defining word must provide space for data or code belonging to each instance of the new class of words. For example, when a variable is defined, space is allotted for its parameter field. If more space is needed, the usual approach is to use **CREATE** followed by **ALLOT**.

After a new defining word has been created, it can be used to create specific instances of its class, with the syntax:

```
<parameters> <defining word> <instance1>
<parameters> <defining word> <instance2>
```

and so forth. The *instance1* and *instance2* are names that would be specified in an application. The *parameters* are optional, depending on the defining word, and are specific to each instance.

When a defining word is executed, it may be followed by any number of words—such as **,** (to compile a single-precision value) or **C,** (to compile an eight-bit value) to fill the allotted storage area with explicit values.

Glossary

;CODE *(—)* Tools Ext

Begin run-time behavior, specified in assembly code. "semi-colon-code"

DOES> *(—)* Core

Begin run-time behavior, specified in high-level Forth. At run time, the address of the parameter field of the particular instance of the defining word is pushed onto the stack before the run-time words are executed. "does"

References **,** and **C,**, Section 4.3.2
;CODE, Section 5.2
ALLOT, Section 4.3.1
CONSTANT, Section 4.2.3
CREATE, Section 4.2.1
DOES>, Section 4.2.6.2
VARIABLE, Section 4.2.2

4.2.6.2 High-level Defining Words

New defining words whose run-time behavior is specified in high-level Forth may be created by using a technique similar to that used for **;CODE**. For these definitions, the word **DOES>** terminates the compile-time portion of the definition and introduces the run-time portion. The form of a **DOES>** definition is:

```
: <name>   <compile-time words>
   DOES> <run-time words> ;
```

After such a definition is compiled, *name* can be used to define a new instance of this class of words. Here, however, the run-time behavior of this class is described in high-level Forth.

At run time, the address of *name*'s parameter field is pushed onto the stack before the *run-time words* are executed. This provides easy access to the parameter field.

An example of a **DOES>** definition is the word **MSG**, which might be used to type short character sequences:

```
: MSG   ( -- ) CREATE
   DOES>  ( -- ) COUNT TYPE ;
```

Here is an example of how **MSG** would be used (assuming **HEX** base):

```
MSG (CR)   2 C, 0D C, 0A C,
```

(CR) is a specific instance of the **MSG** class: it uses the same code (the **DOES>** phrase) as other words defined by **MSG**, but uses that code to emit its own unique character string.

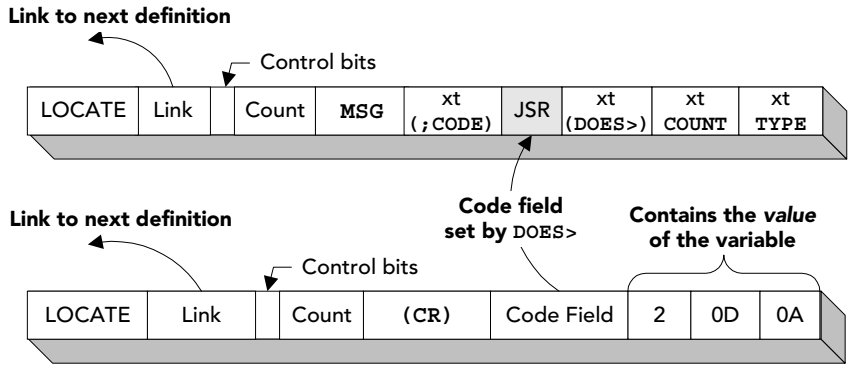

Figure 11. Example of structures defined by using DOES>

The values that comprise the string are kept in the parameter field of the word—in this case, **(CR)**—defined by **MSG**. At execution time, the defining word's **DOES>** puts the address of the instance's parameter field (which, here, is used to store the string) on the stack to serve as the parameter for **COUNT**,

which returns the string's length and byte address as arguments for **TYPE**.

Figure 11 shows a possible implementation of **DOES>**, which works like this:

1. The **:** compiler executes **DOES>**. The compile-time behavior of **DOES>** is to compile code that resets the code field of the new word being defined (the instance of the defining word containing **DOES>**) to point to the cell following the compiled address of **(;CODE)**.
2. After the address of **(;CODE)**, **DOES>** compiles a subroutine call to the run-time code for **DOES>**. The compiler then proceeds to finish compiling addresses in the new defining word. (The use of a subroutine call in the defining word is system dependent. However, all implementations of **DOES>** compile something in the defining word which will allow the run-time code for **DOES>** to find the defining word's high-level code without losing the defined word's data space address.) When the new defining word is executed, its last step will be to change the execution token of the entry it creates to point to the jump-to-subroutine created by **DOES>** in the defining word.
3. When one of the instances created by the new defining word is executed, the virtual machine jumps to the subroutine call in the defining word. Then the subroutine call saves the address of the cell following itself, in some CPU-dependent way, and jumps to the run-time code for **DOES>**. That code uses the address from the subroutine linkage to find the execution token for the defining word. The run-time code for **DOES>** also pushes the address of the defined word's parameter field onto the data stack.

References
, and **C,**, Section 4.3.2
;CODE, Section 5.2
CONSTANT, Section 4.2.3
CREATE, Section 4.2.1
TYPE, Section 3.3.2

4.3 Compiling Words and Literals

A compiling word stores addresses or values into the dictionary, and allots space for definitions and data.

A literal is a number that is compiled directly into a definition or in some other

unnamed form. Covered in this section are several Forth words for compiling literals, including **LITERAL** and **[']**.

4.3.1 ALLOTing Space in the Dictionary

The resident version of **ALLOT** reserves a specified number of bytes in the dictionary by adding to the dictionary pointer. The dictionary usually grows from low memory toward the "top" of the downward-growing data stack. **ALLOT** ensures that some system-specific minimum amount of memory is available for work space. If not enough space remains, **ALLOT** aborts the compilation and issues the message `Dictionary Full`. If the minimum amount is available, **ALLOT** adds the argument on the stack to the address of the next free dictionary byte—this prevents other compiling words from compiling into this portion of memory.

An example of **ALLOT**'s use to create a 200-byte array is:

```
CREATE ARRAY 200 ALLOT
```

The target compiler's version of **ALLOT** differs from the resident version—it allots space in the target system's RAM, rather than in the target dictionary (which is presumed to be in ROM).

Glossary

ALLOT (*n* —) Core

Increment the dictionary address pointer by *n* number of bytes.

References **CREATE** and arrays, Section 4.2.1

4.3.2 Use of , and C, to Compile Values

The word **,** ("comma") stores the top stack item into the next available dictionary location, and increments the dictionary pointer by one cell.

The most common use of **,** is to put values into a table whose starting address is defined by using **CREATE**; **CREATE** defines a word that behaves identically

to **VARIABLE**, in that, when the new word is executed, its address is returned. **CREATE** differs from **VARIABLE** only in that it does not allot any space.

Consider this example:

```
CREATE TENS  1 , 10 , 100 , 1000 , 10000 ,
```

This establishes a table whose starting address is given by **TENS** and which contains powers of ten from zero through four. Indexing this table by a power of ten will give the appropriate value. A possible use might be:

```
: 10** ( n1 n2 -- n)  CELLS  TENS + @ * ;
```

Given a single-precision number n_1 on the stack, with a power of ten n_2 on top, **10**** will multiply the number by the power of ten to yield the product.

When a single byte of data is sufficient, **C,** performs for bytes the same function that **,** performs for cells. On processors that do not tolerate addresses that are not cell-aligned (e.g., 68000), uses of **C,** must be for strings of even cell length, or some other action must be taken to re-align the dictionary pointer.

Even on processors that allow references to any byte address in data space, there usually is an execution penalty for addresses that are not cell-aligned (even addresses in a 16-bit system, and addresses divisible by four in a 32-bit system). Most dictionary entries, such as those created by a colon definition, contain only cell-sized items, so if the dictionary pointer is aligned to begin with, it will stay aligned. However, if words such as **C,** or string-compiling words are used, subsequent unaligned addresses may result.

Two words facilitate alignment in such cases. **ALIGN** takes no stack arguments; when executed, it examines the dictionary pointer and, if it is not cell-aligned, reserves enough additional bytes to align it. **ALIGNED** takes an arbitrary address and returns the first aligned address that is greater than or equal to the given address.

Dictionary entries made by **CREATE**, and by words that use **CREATE**, are aligned. Data laid down by **,** are not automatically aligned, but cell-sized words that access data (such as **@**) may require alignment. Therefore, if you are mixing uses of **,** and **C,** you must manually perform the alignment, e.g.:

```
CREATE TEST 123 C, ALIGN 1234 ,
```

so the phrase **TEST CELL+ @** will properly return 1234.

Glossary

, *(x —)* Core
Reserve one cell of data space and store *x* in the cell. If the data-space pointer is initially aligned, it will remain aligned after **,** has executed. "comma"

ALIGN *(—)* Core
If the data-space pointer is not aligned, reserve enough space to align it.

ALIGNED *(addr — a-addr)* Core
Return *a-addr*, the first aligned address greater than or equal to *addr*.

C, *(char —)* Core
Reserve one byte of data space and store *char* in the byte. "C-comma"

References **CODE**, Sections 4.2.5, 5.1
CONSTANT, Section 4.2.3
CREATE, Section 4.2.1
LITERAL, Section 4.3.5

4.3.3 The Forth Compiler

When a high-level definition is created in the dictionary for a given *name*, it is the task of the Forth compiler to produce a series of executable references, one for each of the previously compiled words that appears in the body of *name*'s definition. The word **COMPILE,** ("compile-comma") is a generic word used by the compiler to create those executable references. **COMPILE,** usually is invoked after the compiler finds a word in the dictionary. It expects the execution token of a word to be on the stack, and it adds the behavior of that word to the definition that currently is being compiled.

Exactly how **COMPILE,** constructs a reference to the word depends on the implementation. In an indirect-threaded model, the references are the actual addresses of the words; in a direct-threaded model, they are jumps; and so forth.

The compiler must handle two special cases besides references to previously

compiled words. The first case occurs when numbers are included in a high-level definition. The compiler handles numbers much like the standard Forth text interpreter does. When a dictionary search fails, the compiler attempts to convert the ASCII string into a number. When conversion succeeds, the number is compiled in-line with a reference to code which will push the number's binary value onto the stack at run time. When the numeric conversion fails, the conversion word aborts and prints an error message.

The second special case occurs with words that must be executed at compile time by the compiler. These words are called *compiler directives*. **IF**, **DO**, and **UNTIL** are examples of compiler directives. After the word is found in the dictionary, the compiler checks the precedence bit in the header of the word's dictionary entry. If the precedence bit is set (i.e., 1), the word is executed, not compiled. If the precedence bit is reset (i.e., 0), a reference to the word is compiled. The precedence bit of any word may be set by placing **IMMEDIATE** directly after the word's definition.

Additionally, sometimes it is necessary to explicitly force the system into interpretation or compilation state. This is done by the words **[** (enter interpretation state, pronounced "left-bracket"), and **]** (enter compilation state, pronounced "right-bracket"). These words set the value of a system variable called **STATE**. **STATE** is *true* (non-zero) when in compilation state, and *false* (zero) otherwise. The only other words that modify **STATE** are **:** (colon), **;** (semicolon), **ABORT**, **QUIT**, and **:NONAME**. It is a violation of Standard Forth to modify the value of **STATE** directly.

The most common use of **[** and **]** is to leave compile-mode temporarily to perform some run-time operation at compile time. For example, in a definition containing numbers most naturally thought of in decimal, suppose you wish to refer to an ASCII code in hex:

```
: GAP ( n)   10 0 DO  [ HEX ] 0A
   EMIT  LOOP ;
```

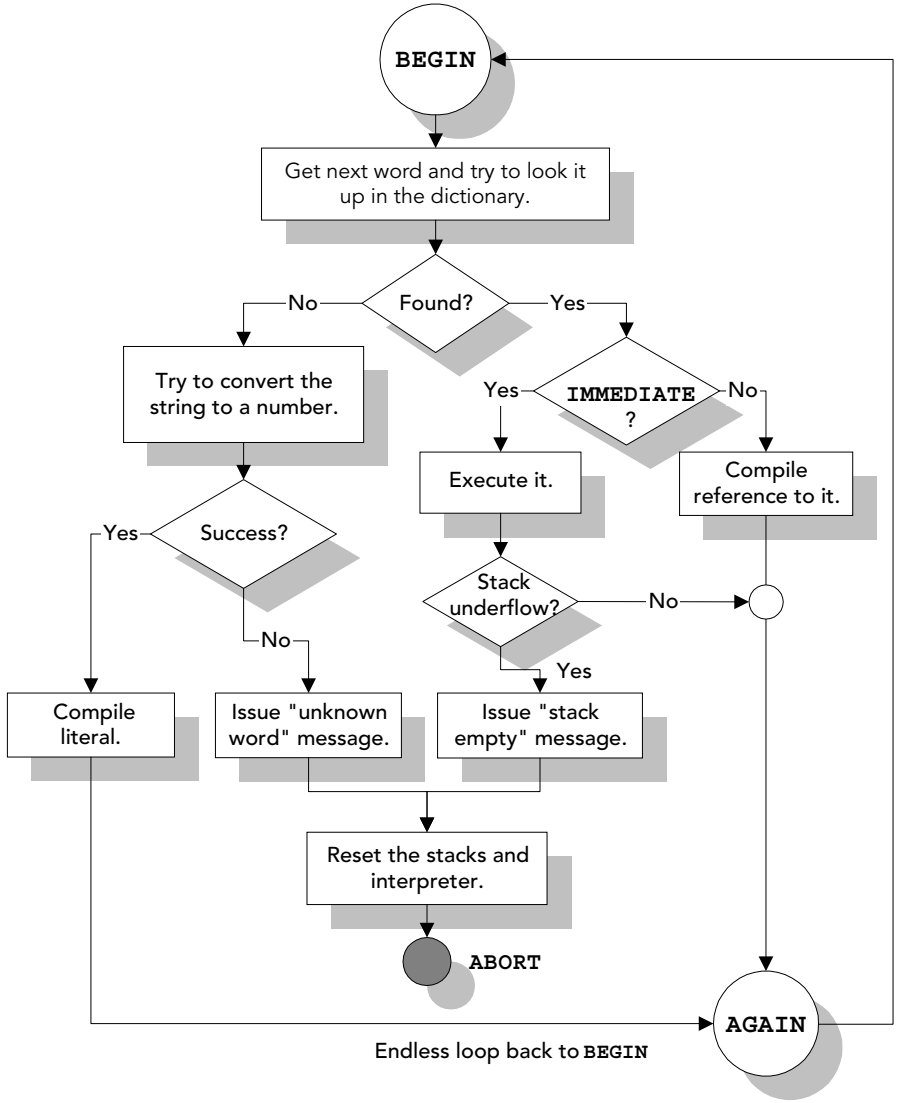

Figure 12. Action of the Forth compiler

Because the words that control **BASE** aren't **IMMEDIATE**, it is necessary to leave compile mode and execute **HEX** before compiling the hex code. **[** is an **IMMEDIATE** word which leaves the compiler and resumes interpretation. **]** returns to compile mode.

Sometimes, when high-level Forth code is necessary but a dictionary header is not (as in some power-up code), the word **]** is used rather than **:**. (This is similar to **:NONAME** but does not leave an execution token on the stack.) Similarly, where high-level Forth is necessary but no address for **EXIT** needs to be compiled on the end of the definition (as when compiling endless loops), **[** may be used instead of **;** to save memory.

Consider, for example, the following possible response to a Break key in an indirect-threaded implementation on an Intel 8086:

```
:NONAME ." Break"  CR ABORT [
ASSEMBLER BEGIN   SWAP # I MOV   STI
   NEXT   0A INTERRUPT
```

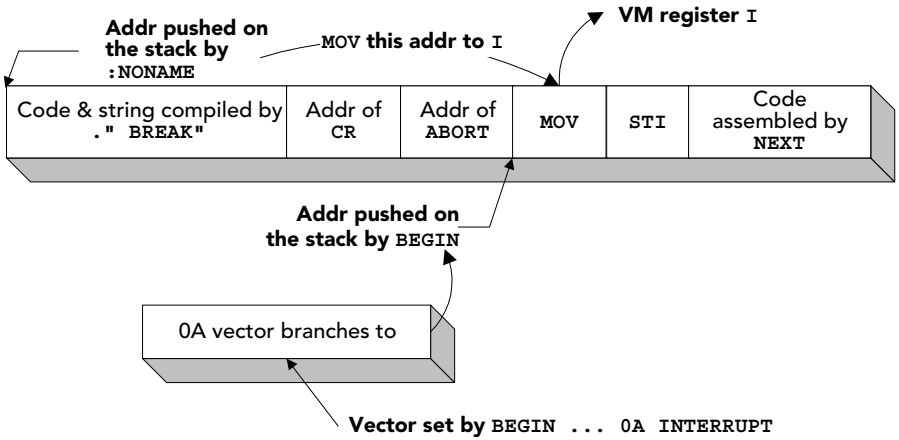

Figure 13. "Break key" response example

At compile time: **:NONAME** compiles the **."** message followed by the references to **CR** and **ABORT**, leaving the address of the beginning of this definition fragment on the stack. **ABORT** aborts the operation of the terminal task that initi-

ated the interrupt, and returns control to the keyboard. Immediately after the address of **ABORT** is the assembler **MOV** instruction, followed by the rest of the code through **NEXT**. The **BEGIN** pushed the address of the **MOV** on the stack; this address and **0A** (the interrupt vector) are the arguments to **INTERRUPT**, which stores the address in the interrupt vector.

At run time: When the user presses the Break key, the interrupt causes a branch through the vector to the **MOV** instruction, which will set Forth's interpreter pointer to the beginning of the high-level phrase starting with **."**. The **NEXT** at the end of the code will start execution of the high-level phrase, terminating with the **ABORT**. Because the phrase is only entered in this way (never called from another high-level word, for example), there is no need to begin it with **:** <name> and since it terminates in **ABORT** there is no need for an **EXIT** (compiled by **;**) at the end.

In a multitasking environment, only rarely can you know which task is controlling the CPU at the time an interrupt occurs. The technique used in this example is, therefore, appropriate only in a narrow range of applications.

Glossary

COMPILE, *(xt —)* Core Ext

Append the execution behavior of the definition represented by the execution token *xt* to the execution behavior of the current definition. "compile-comma"

STATE *(— a-addr)* Core, Tools Ext

Return *a-addr*, the address of a cell containing the compilation-state flag: a non-zero value (interpreted as *true*) when in compilation state, *false* (zero) otherwise.

[*(—)* Core

Enter interpretation state. **[** is an immediate word. "left-bracket"

] *(—)* Core

Enter compilation state. **]** is an immediate word. "right-bracket"

References

ABORT, Section 2.6
Forth virtual machine, indirect-threaded implementations, Section 1.1.7
Colon definitions, Section 4.2.4
Compiler directives, Section 4.4

Dictionary searches, Section 4.1.3
IMMEDIATE, Section 4.4.1
Input number conversion, Section 4.1.4
Interrupts, Section 5.11

4.3.4 Use of Literals and Constants in : Definitions

When the Forth compiler encounters a number in a **:** definition, the number is converted to binary and is compiled as a literal. The compiled form of a literal in a **:** definition has two parts: the number itself, and a reference to code which, when executed, will push the number onto the stack. When Forth is compiling a definition and a number is encountered, this form is automatically compiled. There are other ways in which a literal in a definition may be generated (discussed in the following section), but this is the most common situation.

On many systems, the size of a literal is optimized by the compiler (for example, a literal less than 256 will be compiled as a byte).

Because a literal requires both an in-line number and a reference to run-time code for it, it usually is larger than a **CONSTANT**, which needs only the reference. Therefore, a generic number that will be used frequently (e.g., more than six times) should be defined as a **CONSTANT** to save space. There is not much difference between the time required to execute a **CONSTANT** and a literal. Numbers which have specific meanings (e.g., **86400 CONSTANT SECONDS/DAY**) should always be defined as **CONSTANT**s for program readability.

References Explicit literals, Section 4.3.5
Literal addresses, Section 4.3.6

4.3.5 Explicit Literals

The word **LITERAL** compiles into a definition the number that was placed on the stack at compile time. When the definition is executed, that number will be pushed onto the stack. The compiled result of **LITERAL** is identical to that of a literal number, described in the previous section. **LITERAL** is useful for compiling a reference to an address or to a number that may be computed at compile time.

A common usage of **[** and **]** (leaving and entering compiling state) combined with **LITERAL** is to compile the results of complex calculations that only need to be performed once. As a trivial example, disk status information might be stored in the third cell of an array of disk data named **DISK**. A word to retrieve that information could be:

```
: STATUS   [ DISK 2 CELLS + ]  LITERAL  @ ;
```

The **[** stops compilation, and **]** restarts compilation. During this hiatus, the words **DISK 2 CELLS +** are interpreted and executed, leaving on the stack the address of the status cell, which, after compilation resumes, is compiled into the definition by **LITERAL**. If the calculations are in an inner loop, time savings can be large compared to performing them at run time.

The word **2LITERAL** functions exactly the same as **LITERAL** but requires two values on the stack at compile time and will return those values, in the same order on the stack, at execution time.

SLITERAL is for use with strings. This word requires an address and length of a string on the stack at compile time. The string is compiled into the definition and, at execution time, **SLITERAL** returns the address where the string was compiled and its length. See Section 4.3.7 below for a fuller description.

Glossary

LITERAL *(— x)* Core

At compile time, remove the number that is on top of the stack and compile it into the current definition. At run time, return the number to the stack.

2LITERAL *(— x_1 x_2)* Double

At compile time, remove the top two items on the stack and compile them into the current definition. At run time, return the items to the stack in the same order. "two-literal"

References **[** and **]**, Section 4.3.3
Compilation of literals, Section 4.3.4

4.3.6 Use of ['] to Compile Literal Addresses

The word **[']** ("bracket-tick") is used inside a definition to compile as a literal the execution token of the word that follows it at compile time. The most common use of **[']** is to obtain the address of either a **CONSTANT** or a **2CONSTANT** (on systems that have the 32-bit option or on 32-bit machines). Consider the following example:

```
0 500 2CONSTANT LIMITS
: RANGE ( d -- )  ['] LIMITS >BODY 2! ;
```

Given these definitions, the phrase:

```
0 2000 RANGE
```

resets the values of **LIMITS** to zero and 2000. The address of the beginning of the double-precision parameter field for **LIMITS** is compiled as a literal in **RANGE** and is pushed onto the stack when **RANGE** is executed, to provide the address for **2!**.

References **[']**, Section 4.1.3

4.3.7 Compiling Strings

Forth provides two methods for compiling strings. The most generally useful word is **S"** (pronounced "s-quote"). It compiles a string, using a quotation mark as a delimiter, and stores it in the dictionary. When the word containing this string is executed, the address and length of the string are pushed on the stack.

A similar word, **C"**, compiles a *counted string* (compiled with its count in the first byte, a common practice in Forth). At execution time, **C"** returns the address of the length byte. Therefore, frequently it is useful to use **COUNT** to fetch the address of the first byte of the string and the string's length.

For example, consider a word **?NO** that compares a data string—whose address and length are on the stack—to a string that is compiled as part of the definition of **?NO**, and returns *true* if they match:

```
: ?NO ( addr n -- flag)   C" no" COUNT COMPARE ;
```

?NO takes the address and length of an input string. When **?NO** is executed, **C"** will push the address of a compiled counted string on the stack. **COUNT** converts that address to a string address and length, leaving the appropriate arguments for **COMPARE** (which performs the comparison).

Here is a word which will search for a compiled string in a longer string whose address and count are on the stack:

```
: ?DUCK ( addr n -- flag)   S" duck"
   SEARCH  NIP NIP ;
```

The **NIP**s discard the address and character count where the match (may have) occurred.

In cases similar to the examples above, you might need to allow the test string to contain an arbitrary mixture of upper- and lower-case characters. If so, you should set or clear the appropriate bit in each byte of the test string, to standardize on all upper or all lower case, before making your comparison.

SLITERAL is the low-level compiling word used by **S"**, **C"**, and similar string-handling words. Just as **LITERAL** compiles into a definition the number found on the stack at compile time, and returns that number at execution time, **SLITERAL** compiles into a definition a string, characterized by an address and length on the stack at compile time, and returns the string's address and length at execution time. The address at compile time is not the same as the address at execution time—the former typically is an address in the input buffer, and the latter is an address connected with the definition using **SLITERAL**.

Consider how you might define the word **S"** to begin compiling a string, which will be terminated by a second quote, and which will leave the string's address and count on the stack at execution time. It could be used as follows:

```
: ALARM-MESSAGE   S" Too Hot!" TYPE ;
```

A possible definition for **S"** would be:

```
: S"   [CHAR] " WORD  COUNT  POSTPONE SLITERAL ; IMMEDIATE
```

When **S"** executes (which is at compile time, since it is marked **IMMEDIATE**), the phrase **[CHAR] "** returns the ASCII value for the quote character, which is passed to **WORD** for use as the delimiter. **WORD** parses the input stream and returns the address of a counted string in the input buffer consisting of all the

characters between the **S"** (the name of the executing word) and the delimiting **"**. All spaces are included, even leading spaces, because in this case a space is not the delimiter. **COUNT** converts the counted string address to a character string address and length; these two parameters are passed to **SLITERAL**, which compiles the string into the definition. The **POSTPONE** command preceding **SLITERAL** causes **SLITERAL**'s compilation behavior to occur rather than its execution behavior. When **ALARM-MESSAGE** executes, the run-time behavior of **SLITERAL** returns the address and count of the stored message `Too Hot!` for **TYPE** to display.

Glossary

SLITERAL *(— c-addr u)* String

Compile into a definition a string that is characterized by the starting address and length on the stack at compile time. At run time, return the string's address and length to the stack. In general, the run-time address will not be the same as the compile-time address. "S-literal"

References
S" and **C"**, Section 4.1.5.2
Defining words, Section 4.2
String comparisons, Section 2.3.4
POSTPONE, Section 4.4.1

4.4 Compiler Directives

A *compiler directive* in Forth is a word that is *executed* at compile time, i.e., during a **:** compilation. Many such words exist: **DO**; **LOOP** and **+LOOP**; **BEGIN** and **UNTIL**; **IF**, **ELSE**, and **THEN**; literals; and others. It is rare that a user needs to add compiler directives; it is not difficult, but requires mastery of **IMMEDIATE** and **POSTPONE**.

Some compiler directives have only compile-time behavior (such as **BEGIN**). Other directives need to perform some actions at compile time and other actions at run time. For example, at compile time **DO** must mark the position to which **LOOP** or **+LOOP** will return; at run time, it must push the index and limit for the loop onto the return stack.

The way these functions are managed is to define (usually with **CODE**) the run-time activity as a separate word and then to have the compile-time definition, which is **IMMEDIATE**, compile the address of the run-time code (in addition to its other activities).

4.4.1 Making Compiler Directives

The word **IMMEDIATE** is used directly after a definition. It signals the compiler that this definition is to be *executed* at compile time (when all non-immediate words are being *compiled*). This is done by setting the word's precedence bit (usually the high-order bit in the count field).

The word **POSTPONE** is used inside **IMMEDIATE** definitions. It has the opposite function from **IMMEDIATE**. Used in the form **POSTPONE** `<name>`, it causes the *compilation* behavior of *name*, rather than the *execution* behavior of *name*, to be added to the current definition. **POSTPONE** can be used with **IMMEDIATE** words (such as compiler directives) and with normal, non-immediate words, as shown in the following examples.

Consider a common definition of **BEGIN**:

```
: BEGIN   HERE ;  IMMEDIATE
```

This definition of **BEGIN** is simply an **IMMEDIATE** version of **HERE**. The difference is that, when **HERE** appears in a normal definition, its address is compiled; it will push the value of the dictionary pointer onto the stack when the word that contains **HERE** is executed. **BEGIN**, on the other hand, compiles nothing; it pushes the dictionary pointer onto the stack at compile time, to serve as the address **UNTIL** (or a similar structure word) needs to compile a conditional return to that location.

S*tructure* words such as **IF** provide classic examples of the use of **POSTPONE**. Most of these words have a *run-time behavior* that we usually think of. For example, **IF** checks the truth of the top stack item, and conditionally branches. But there is also a *compile-time behavior* for **IF**, which is to compile a reference to the run-time behavior and also to provide for whatever branching is associated with the structure.

If the run-time behavior is defined as a word called **(IF)**, we could define the

compiler directive **IF** this way:

```
: IF   ( -- addr )
   POSTPONE (IF)  HERE 0 , ;
IMMEDIATE
```

When executed during compilation of a word, this **IF** will compile the reference to **(IF)** and leave a one-cell space in the definition, placing the address of that cell on the stack, as shown in Figure 14. Subsequent execution of **ELSE** or **THEN** will resolve the branch by storing an appropriate offset in that space.

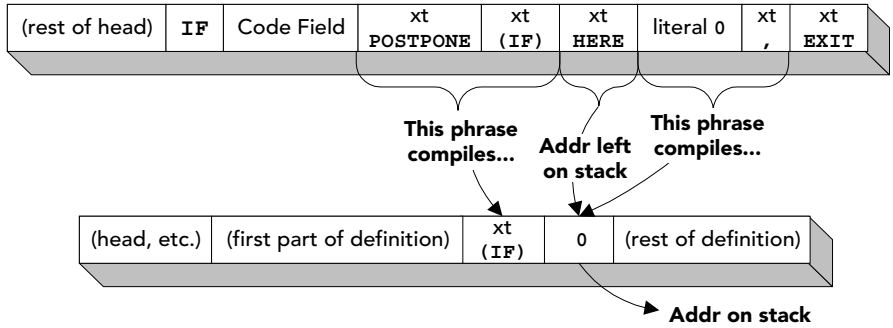

Figure 14. Compile-time action of IF

As a second example, suppose you often use the phrase … **?DUP IF** … in definitions, and want to create a word **?IF** that performs both functions. Here is how **?IF** would need to be defined:

```
: ?IF   POSTPONE ?DUP  POSTPONE IF ; IMMEDIATE
```

?IF is an **IMMEDIATE** word because it needs to set up a conditional branch at compile time. However, we do not want the run-time behavior for **?DUP** and **IF** to execute at compile time; instead, we want these words' compilation behaviors to occur. Hence, each must be preceded by **POSTPONE**. **?DUP** is non-immediate, and **IF** is **IMMEDIATE**, but the syntax for **POSTPONE** is identical.

POSTPONE is very similar to **[']** except, whereas **[']** compiles as a literal the execution token of the word that follows (so the address will be pushed onto the stack at run time), **POSTPONE** lays down a pointer to the execution token, so the word can be executed by the Forth virtual machine.

Glossary

IMMEDIATE *(—)* Core

Make the most recent definition an immediate word. When the compiler encounters an immediate word it causes it to execute at that time rather than compiling a reference to it.

POSTPONE <name> *(—)* Core

At compile time, add the *compilation* behavior of *name*, rather than its execution behavior, to the current definition. Usually used in **IMMEDIATE** definitions.

References

Colon definitions, Section 4.2.4
DO ... **LOOP**, program structure words, Sections 2.5, 5.8
Literals, Section 4.3.5
The Forth compiler, Section 4.3.3
Use of **BEGIN**, Section 2.5.1
Word lists, Section 4.6
['], Section 4.3.6
Compiler directives, Section 4.4

4.4.2 The Control-flow Stack and Custom Compiling Structures

The standard branching constructs in Forth (**IF** … **ELSE** … **THEN**, **BEGIN** … **UNTIL**, **BEGIN** … **AGAIN**, and **DO** … **LOOP**) are examples of *control-flow words*. In direct management of control flow, every branch must terminate at some destination. An *origin* (abbreviated *orig* in Table 8) is the location of the branch itself; a *destination* (abbreviated *dest* in Table 8) is where control will continue if the branch is taken. A natural implementation to manage control flow uses a stack to remember the origin of forward branches and the destination of backward branches. This is the *control-flow stack* in Forth. How it is implemented is system dependent, and generally is not of concern to the user; in virtually all implementations, it is the data stack at compile time.

This section describes some additional primitive words which directly access the control-flow stack. With these words, a programmer can create branching structures of any needed degree of complexity. The abilities required are compilation of forward and backward conditional and unconditional branches,

and compile-time management of branch origins and destinations. These are provided by just three new words: **AHEAD**, **CS-PICK**, and **CS-ROLL**. Table 8 summarizes the compilation behavior of these words and of the other basic Forth words that affect control flow.

Table 8: Summary of compile-time branch words

Word	Control-flow stack	Function
IF	*(— orig)*	Marks the origin of a forward conditional branch.
THEN	*(orig —)*	Resolves the branch originated by **IF** or **AHEAD**.
BEGIN	*(— dest)*	Marks the destination of a backward branch.
AGAIN	*(dest —)*	Resolves a backward unconditional branch.
UNTIL	*(dest —)*	Resolves a backward conditional branch.
AHEAD	*(— orig)*	Marks the origin of a forward unconditional branch.
CS-PICK	*(i*x u — i*x x_u)*	Copies item on control-flow stack.
CS-ROLL	*(i*x u — (i-1)*x x_u)*	Reorders items on control-flow stack.

All other branching words—such as **WHILE**, **REPEAT**, and **ELSE**—can be defined in terms of the primitive words in Table 8. For example:

```
: ELSE ( addr1 -- addr2 )  \ Resolve IF, set up for THEN
   POSTPONE AHEAD          \ Set up forward branch
   1 CS-ROLL               \ Get addr of IF's branch
   POSTPONE THEN ;         \ Resolve IF's branch
IMMEDIATE
```

In this definition, the phrase **POSTPONE AHEAD** marks the origin of an unconditional branch (around the "false clause") to be taken at the end of the "true clause." This will later be resolved by the **THEN** which occurs at the end of the **IF** statement. Since **POSTPONE AHEAD** places one item on the control-flow stack, the phrase **1 CS-ROLL** (the equivalent of **SWAP**) is needed to restore the previous origin which was placed there by the **IF**. Next, **POSTPONE THEN** compiles the branch resolution for this origin, providing entry to the "false clause" following **ELSE** if the conditional branch at **IF** was taken.

Glossary

AHEAD *(— orig)* Tools Ext

At compile time, begin an unconditional forward branch by placing *orig* on the control-flow stack. The behavior is incomplete until the *orig* is resolved, e.g., by **THEN**. At run time, resume execution at the location provided by the resolution of this *orig*.

CS-PICK *(i*x u — i*x x_u)* Tools Ext

Place a copy of the *u*th control-stack entry on the top of the control stack. The *zero*th item is on top of the control stack; i.e., **0 CS-PICK** is equivalent to **DUP** and **1 CS-PICK** is equivalent to **OVER**. "C-S-pick"

CS-ROLL *(i*x u — (i-1)*x x_u)* Tools Ext

Move the *n*th control-stack entry to the top of the stack, pushing down all the control-stack entries in between. The *zero*th item is on top of the stack; i.e., **0 CS-ROLL** does nothing, **1 CS-ROLL** is equivalent to **SWAP**, and **2 CS-ROLL** is equivalent to **ROT**. "C-S-roll"

References Indefinite loops, Section 2.5.1
DO … **LOOP**s, Section 2.5.2
IF … **ELSE** … **THEN**, Section 2.5.3

4.5 OVERLAYS

Because of Forth's compilation speed, there is rarely need for a dynamic runtime overlay capability. Many resident applications have several functionally independent subsets, however, and it is conventional to organize these as mutually exclusive overlays, any one of which may be loaded into each terminal's private dictionary. This is done by explicit command. Once loaded, such an overlay will remain resident until replaced by another.

Examples of such overlay categories in a business environment might include order entry, payroll, and general ledger. In a scientific laboratory system, there may be several different data acquisition and analysis modes.

Overlays are enabled with **MARKER**. The phrase **MARKER** <name> creates a dictionary entry for *name*. When *name* is executed, it will discard the definition

name and all words defined after *name* in a user's partition. The user's dictionary pointer will be reset to the last definition in the vocabulary before *name*. Because the dictionary pointer is reset, the dictionary is truncated spatially as well as logically. Other system-dependent actions may be taken as well, such as restoration of interrupt vectors (see your product documentation).

MARKER has two uses:

- To discard only part of your definitions. For example, when testing, you may wish to reload only the last block, not your entire application.
- To create additional levels of overlays.

Suppose your application includes an overlay called **GRAPHICS**. After **GRAPHICS** is loaded, you want to be able to load one of two additional overlays, called **COLOR** and **B&W**, thus creating a second level of overlay. Here is the procedure to follow:

1. Define a marker as the final definition of **GRAPHICS**, using any word you want as a dictionary marker. For example:

```
MARKER OVERLAY
```

   Preferably, such a definition would be placed at the bottom of the **GRAPHICS** load block.

2. Execute **OVERLAY** and then redefine it (since it forgets itself) on the first line of the load block of each level-two overlay. For instance,

```
( COLOR)    OVERLAY    MARKER OVERLAY
```

   Thus, when you execute the phrase:

```
COLOR LOAD
```

   the system will *forget* any definitions which may have been compiled after **GRAPHICS** and will restore the marker definition of **OVERLAY** in the event you want to load an alternate level-two definition, such as **B&W**.

   By using different names for your markers, you may create any number of overlay levels.

Glossary

MARKER <name> (—) Core Ext

Create a dictionary definition for *name*, to be used as a deletion boundary. When *name* is executed, remove the definition of *name* and all subsequent definitions from the dictionary. Restore all dictionary allocation and search order pointers to the state they had just prior to the definition of *name*.

4.6 WORD LISTS

Word lists are collections of definitions residing within the dictionary. ANS Forth guarantees there will be at least eight wordlists available to the user. Dictionary searches proceed from one word list to another in a specified sequence. This mechanism allows you to control which list or lists are to be searched. Within each word list, the search is from newest to oldest.

Word lists have three principal uses:

- In the resident system, to segregate special-purpose words such as those in the **ASSEMBLER**, to allow them to have the same names as standard Forth words.
- In a target compiler environment where two types of CPU exist, to segregate target versions of **FORTH** and **ASSEMBLER** words from the host versions.
- In applications running in the host system, to protect against accidental misuse of words only intended for programmers.

4.6.1 Basic Principles

The standard word lists provided by typical Forth systems are:

FORTH
ASSEMBLER (on most systems)
EDITOR (on systems with an internal editor)

Other lists may be created, as described below. The **FORTH** word list contains most of the familiar words such as **DUP**, **SWAP**, **DO**, etc. Another word list on most systems is **ASSEMBLER**, containing words used to assemble machine

code. **EDITOR** contains the editing commands for editing source text.

The use of separate word lists makes it possible, for instance, for the word **I** to supply a loop index in one context (**FORTH**), to insert a string in another context (**EDITOR**), or to name a register in yet another (**ASSEMBLER**).

When the Forth interpreter receives a word, whether it is one you have typed at the keyboard or one it gets from a file, it looks for that word in an ordered sequence of word lists. The sequence is called the *search order*. A word will not be found unless it is contained in a word list in the search order. The search order may be changed at any time. A pointer to the first word list in the search order is kept in the variable **CONTEXT**.

To display the search order, use **ORDER**.

When a Forth word is compiled, it will be placed in the current *compilation word list*. This is not necessarily the same word list that is first in the interpretation search order. A pointer to the current compilation word list is kept in the variable **CURRENT**. Words are provided, as described below, for manipulating both the interpretation search order and the compilation word list.

You may change the contents of **CONTEXT** (i.e., select the word list to search first) simply by naming the desired word list. For example, the word:

```
ASSEMBLER
```

changes **CONTEXT** so future searches will begin with the **ASSEMBLER** word list. (**CONTEXT** is set to **ASSEMBLER** by the defining words **CODE** and **;CODE**.)

Similarly, you may employ the word:

```
EDITOR
```

to set **CONTEXT** to begin by searching the **EDITOR** word list. In many cases, **EDITOR** commands are found in **FORTH** and automatically set **CONTEXT** to the **EDITOR** word list.

The contents of **CURRENT**, which selects the compilation word list, may also be changed. The word **DEFINITIONS** sets **CURRENT** to the word list indicated by **CONTEXT**. For example, in the phrase:

```
EDITOR DEFINITIONS
```

EDITOR sets the value in **CONTEXT** to be the **EDITOR** word list. **DEFINITIONS** then sets **CURRENT** also to **EDITOR**. Thereafter, any future definitions will be linked according to the **EDITOR** word list. Subsequent changes in the search order will change **CONTEXT**, but **CURRENT** remains as set until explicitly changed. When the system starts, or following an **EMPTY**, the default word list for both **CONTEXT** and **CURRENT** is **FORTH**.

Invoking the name of a word list always *replaces* the word list that previously was at the head of the search order. To add a word list to the head of the search order and still retain the previous word list in the search order, use **ALSO** (see below) followed by the name of the word list you want to add.

4.6.2 Managing Word Lists

Here are some words for manipulating word lists:

Glossary

ALSO *(—)* Search Ext

Duplicate the first word list in the search order, increasing the number of word lists in the search order by one. Commonly used in the phrase **ALSO** *name*, which has the effect of adding *name* to the top of the search order.

ASSEMBLER *(—)* Tools Ext

Set future dictionary searches to begin with the **ASSEMBLER** word list (available on most systems).

CONTEXT *(— a-addr)* Core

Return *a-addr*, the address of a cell that contains a pointer to the first word list in the search order.

CURRENT *(— a-addr)* common usage

Return *a-addr*, the address of a cell that contains a pointer to the current compilation word list.

DEFINITIONS *(—)* Search

Change the compilation word list to be the same as the current first word list in the search order. Set a pointer to this word list in the variable **CURRENT**.

Subsequent changes to the interpretation search order will not affect the compilation word list; this word list remains in effect until explicitly changed.

EDITOR *(—)* Tools Ext

Set future dictionary searches to begin with the **EDITOR** word list.

FORTH *(—)* Search Ext

Set future dictionary searches to begin with the **FORTH** word list, which contains all standard words provided by the system implementation.

ONLY *(—)* Search Ext

Reduce the search order to the minimum word list(s), usually just **FORTH**.

ORDER *(—)* Search Ext

Display names of the word lists in the search order, in their present search order sequence. Also, display the word list into which new definitions will be placed (the **CURRENT** word list).

PREVIOUS *(—)* Search Ext

Remove the first word list (the one in the **CONTEXT** position) from the search order. This may be used to undo the effect of an **ALSO**.

VOCABULARY `<name>` *(—)* common usage

Create a word list *name*. Subsequent execution of *name* replaces the first word list in the search order with *name*. When *name* becomes the compilation word list, new definitions will be appended to *name*'s list.

WORDS *(—)* Tools

Displays the names of all the words in the first word list of the search order.

4.6.3 Sealed Word Lists

The word list mechanism offers an exceptionally powerful security technique. You can implement this by setting up a special application word list consisting of a limited number of commands guaranteed to be safe for users. You then ensure that no application word can change **CONTEXT**, and that **CONTEXT** is set so the text interpreter will only search the application word list.

5. THE ASSEMBLER

Forth is one of the fastest, most efficient high-level languages, and is used extensively in real-time programming and applications programming. Application programs usually are written in the extensible Forth word set. However, for low-level words that will be executed a very large number of times, or anywhere there are particular time constraints, Forth can assemble machine-language definitions of Forth words. Among the many examples of low-level words defined by machine-language instructions in the Forth nucleus are the operations:

```
+   -   SWAP   DROP   2DUP
```

The assembler for your particular CPU is explained in the product documentation. This section provides a general overview of the assembler on any Forth system. Note that the assembler is not used in ordinary high-level Forth programming, only in **CODE** definitions. Assembler code is, by definition, machine dependent. However, many characteristics of Forth assemblers are relatively consistent across all the processors on which Forth has been implemented. This set of common characteristics is discussed in this section. Examples will be given using code for some of the most popular processors; unfortunately, space will not permit providing versions of each example for all processors. However, the principles should be clear.

5.1 CODE DEFINITIONS

The Forth defining word **CODE** creates a standard dictionary entry whose code address field contains the address of the byte that follows, which is the first byte of the parameter field where machine instructions are assembled. See Figure 15 for a diagram of this dictionary entry. The form of a **CODE** definition is:

```
CODE <name> <instructions> <code ending>
```

CODE creates a definition with the given *name*. It also selects the **ASSEMBLER** vocabulary, in which the various instruction mnemonics, addressing modes, etc., are defined. These are used to build actual machine instructions, which are laid down in subsequent locations in the dictionary. The *code ending* is one of several macros, each of which ultimately returns to Forth's virtual machine.

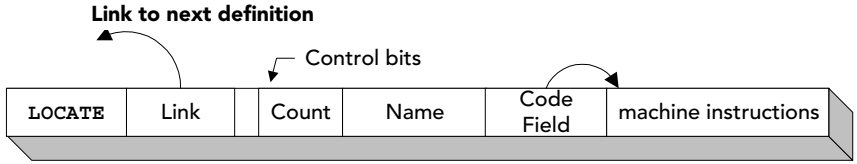

Figure 15. Diagram of a dictionary entry for a CODE entry

Aside from the dictionary entry header, there is no high-level language overhead, in size or speed, within a code definition. All instructions are executed at full machine speed.

As a general rule, Forth programs are written first in high-level language. Then a time analysis is performed to locate the most frequently executed words, which are then re-written as **CODE** definitions. Two examples of such words might be:

- The portion of an interrupt routine that actually moves data to and from a device.
- The innermost loop of a routine where the computer spends a significant portion of its time (for example, the word **NEXT** in a Forth kernel).

Glossary

CODE <name> (*i*x — j*x*) Tools Ext

Create a definition for *name*, called a *code definition*, and start its definition. The execution behavior of *name* will be determined by the assembly-language words that follow and which are compiled into the body of the definition. The *name* cannot be found in the dictionary until the definition is ended. At execution time, the stack effects of *name* depend on its behavior.

References Code endings, Section 5.2
Macros, Section 5.7

5.2 Code Endings

Most Forth code routines end with some formal ending. The most common code ending on direct-threaded and indirect-threaded implementations is a routine called **NEXT**, which is sometimes implemented as a macro. This and the other most common code endings are summarized below. On some processors, these endings assemble code or branches to the appropriate code; on others, they return addresses which may be used as arguments to a **JMP**. Many other systems have a code ending word that explicitly leaves the Assembler word list and otherwise completes the definition; the most common name for this function is **END-CODE**.

Refer to your product documentation for a list of the code endings for your processor.

Glossary

NEXT *(—)* common usage

Exit to the next high-level definition (via the Forth virtual machine).

END-CODE *(—)* common usage

Terminate a code definition in an implementation-dependent manner, typically leaving the Assembler word list and rendering the newly completed definition available to dictionary searches.

INTERRUPT *(addr i*x —)* common usage

Set up an interrupt vector to the code at *addr*. The parameters *i*x* represent system-dependent device and interrupt vector locations. This is very implementation-dependent. On some systems it is called **EXCEPTION**.

5.3 Assembler Instructions

To compile a colon definition, the interpreter enters a special compile mode in which the words of the input string are not executed (unless designated as **IMMEDIATE**). Instead, their addresses are placed sequentially in the dictionary. During assembly, however, the interpreter remains in execute mode. The mnemonics of the processor instructions are defined as words which,

when executed, assemble the corresponding operation code at the next location in the dictionary. Operands (addresses or registers) precede instruction mnemonics, in order to leave information on the stack that will be used by the mnemonic to assemble the instruction.

The Forth assembler for each processor defines words for the available instruction and addressing formats. Those words may then be used to assemble instructions, along with the operation code and any required parameters. The new instruction is assembled into the next available dictionary location.

For example, the Intel 8080 processor has an ALU reference instruction format for instructions that perform arithmetic computations. The Forth assembler defines the command **ALU**, which is used to define mnemonics of the ALU class, which in turn assemble ALU reference instructions. For example, the mnemonic ADD is defined on 8080 systems by:

```
80 ALU ADD
```

ADD is an operation which assembles an ALU-type instruction whose numeric code is 80_H and whose operand will be on the stack. In use,

```
L ADD
```

assembles an instruction which, when executed, will add the contents of Register L into the accumulator.

5.4 Notational Conventions

Although each Forth assembler uses the manufacturer's mnemonics, some standard Forth notational conventions are shared by many Forth assemblers. Registers for the Forth virtual machine (which may be in actual CPU registers or in memory) have the standard names given in Table 9.

These pointers are often kept in registers, but may reside in memory in some computers. Refer to your product documentation for a discussion of their locations on your system. Wherever these pointers reside, the standard names may be used in code to refer to them.

Registers are numbered in a way that reflects the manufacturer's usage and

the actual bits used in assembled instructions. In addition, for convenience and readability, some registers are given names by using **CONSTANT**. Thus, on the Intel 8051:

```
108 CONSTANT R0
109 CONSTANT R1
10A CONSTANT R2
…etc.
```

enables you to refer to the machine registers by familiar names. In addition to defining the manufacturer's names, the Forth virtual machine's registers are sometimes named, e.g., **S**, **R**, and **I**. This is helpful to Forth programmers who work on a variety of CPUs.

Table 9: Forth assembler notation conventions

Name	Description
S	Address of the top of the parameter stack.
W	Address of the parameter field or code field of the current definition.
I	Interpreter pointer.
R	Address of the top of the return stack.
U	Beginning of the user area.

Forth code routines tend to be extremely short, averaging under a dozen instructions on 16-bit processors. Moreover, Forth assembler code is entirely structured. For such short routines, the conventional vertical format with comments on each line is not needed, and takes considerable space. FORTH, Inc. recommends, and uses internally, a horizontal format, with three spaces between each instruction and one space between each component of an instruction (address specifiers, the mnemonic itself, etc.). In this format, the average code definition occupies only two or three lines, and is still readable. For example, on an 80386 one might find:

```
CODE +    ( n1 n2 -- n3 )    0 POP    0 S ) ADD    NEXT
```

References Documentation facilities, Section 1.4
Forth virtual machine, Section 1.1.7

5.5 Use of the Stack in Code

When using code, it is necessary to distinguish between how the stack is used at assembly time and at execution time. The words in a code entry are executed at assembly time to create machine instructions, which are placed in the dictionary to be executed later. Thus, for example,

```
HERE 2- TST
```

at assembly time places the current dictionary location on the stack (**HERE**) and decrements it by two. The resulting number is the parameter for **TST**, which assembles a machine instruction that is the equivalent of:

```
TST *-2
```

in conventional assembler notation. Similarly, such words as **SWAP** and **DUP** are executed at assembly time to manipulate the parameters being used by assembler words, although such stack words would be compiled into the dictionary in a **:** definition. For example, in the 8080:

```
0  HERE SWAP  H LXI  JMP
```

assembles an endless loop that loads zero into the accumulator. **HERE** pushes the address of the next free byte of dictionary space onto the stack. The phrase **H LXI** takes the zero from the top of the stack (at assembly time) and assembles a "load index immediate" that will load zero into the **HL** register pair. The **JMP** uses the address left on the stack to assemble a jump to the first byte of the load.

In high-level definitions, the run-time use of the stack is implicit: numbers you type are placed there, routines naturally leave their results there, etc. Code, however, requires that parameters be handled explicitly, using **S** (the parameter stack pointer) and the code-endings that push or pop the stack before executing **NEXT**.

5.6 Addressing Modes

In general, Forth assemblers implement the processor manufacturer's mnemonics, but many standardize notational conventions for specifying address-

ing modes. Obviously, not all processors have all addressing modes, nor do they interpret terms such as "relative" identically. Nonetheless, certain basic concepts do exist and it's helpful, when you're working with several processors, to have these concepts expressed in standard ways.

Refer to your product documentation for the specific addressing modes implemented in your system.

Typical Forth addressing notation includes the right parenthesis, which indicates relative addressing (when it is by itself) or indexing (when it is combined with an index register designation). Some examples:

Notation	Addressing mode
`S )`	Addressing relative to the top of the stack.
`S)`	Indexed by `S`.
`1)`	Indexed by Register 1.

On machines with automatic incrementing or decrementing, the parenthesis may be combined with `+` or `-`. On the Motorola 68000 family, for example:

Notation	Function
`S )+`	Refers to the number on top of the stack, *popping* it off at the same time—that is, incrementing the stack pointer.
`S -)`	Refers to the next available stack location—a *push* operation.

The position of the sign indicates when the increment or decrement takes place in the computation of the effective address; the two preceding examples show post-incrementing and pre-decrementing.

Immediate addressing is indicated by `#` and memory-indirect by the right parenthesis; the assembler can determine from the address whether `)` means register-relative or memory-relative (indirect). In addition, specific notation for each processor are described in the product documentation.

Parameters may be taken directly from memory, if this is permitted by the architecture of the processor. The assembler will check to determine whether the address of the argument permits a short format instruction. If it will not,

an extended format will be used. Often, parameters may be supplied to an operation without being named. As long as an address is on the stack, it doesn't matter how it got there:

```
HERE 55 , ... LDA
```

will enter the literal number 55 in the dictionary and leave its address on the stack at assembly time. (The operation puts the number that is on the stack into the dictionary at **HERE** and increments **H**, the dictionary pointer, by one cell.) The **LDA** instruction encounters the address on the stack and assembles an instruction to move its contents to Register A.

References **,** (comma), Section 4.3.2

5.7 MACROS

Macros are easily defined in Forth by using **:** definitions that contain assembler instructions. For example, on the RCA 1802 one frequently uses the operations **DEC** and **STR** successively on the same register. For convenience, the following macro has been defined:

```
: DST ( r)    [ ASSEMBLER ] DUP DEC  STR ;
```

Thus, **S DST** could be used to assemble the two instructions:

```
S DEC  S STR
```

Note the way **DUP** in the definition of **DST** allows the single parameter **S** to be used by both the **DEC** and **STR** mnemonics.

Macros are mainly a notational convenience; **DST** assembles two instructions, as if the expressions had been written out in full.

The words used to implement the assembler structures (loops and conditionals) are defined as macros, as are the code endings.

5.8 PROGRAM STRUCTURES

Control of logical flow is handled by Forth's assembler using the same structured approach as high-level Forth, although the implementation of the commands is necessarily different. The commands even have the same names as their high-level analogues (e.g., **BEGIN** … **UNTIL**, **IF** … **ELSE** … **THEN**); ambiguity is prevented by use of separate word lists.

In conditional branches, the **ELSE** clause in an **IF** … **ELSE** … **THEN** construct may be omitted entirely. This construction is functionally analogous to the **IF** … **ELSE** … **THEN** construction provided by Forth's compiler. For instance,

```
0= IF <code for 0> ELSE <code for not 0> THEN …
0= IF <code for 0> THEN …
```

Please note, however, that whereas the **IF** and **UNTIL** in high-level Forth remove an item from the stack and test it, the corresponding assembler words assemble conditional branches whose action will depend on condition codes set by the result of a previous instruction.

Because the locations or destinations of branches are left on the stack at assembly time, the structures **BEGIN** … **UNTIL** and **IF** … **ELSE** … **THEN** may be nested naturally. By manipulating the stack during assembly, however, you can assemble any branching structure.

To branch forward, use **IF** to leave the location of the branch's address field on the stack. At the branch's destination, bring the location back to the top of the stack (if it is not there already) and use **ELSE** or **THEN** to complete the branch (by filling in the branch's destination at the location that is on the top of the stack).

To branch back to an address, leave it on the stack with **BEGIN**. At the branch's source, bring the address to the top of the stack and use **UNTIL** or a jump mnemonic to assemble a conditional or unconditional branch back. Be sure to manipulate the branch address before the condition mnemonic, because each condition code adds one item to the stack.

Suppose, for example, you wish to define a word **LOOK**, which takes two parameters (a delimiter on top of the stack with a starting address beneath it) and which scans successive bytes until it finds the delimiter or a zero. The number of characters scanned is returned. Here is a definition for the Motorola 6800:

```
CODE LOOK ( a c -- n)   B PUL  A PUL  TSX
 0 ) LDX  BEGIN   0 ) TST   0= NOT IF
    0 ) A CMP   0= NOT IF   INX   B INC
    ROT JMP  THEN THEN   A CLR
 TSX   PUT JMP
```

Here the phrase **0= NOT IF** (used twice) assembles two conditional forward jumps which will be executed if the character scanned is the same as one of the delimiters. If the loop is to be repeated, after **B INC** a **JMP** is needed back to the **BEGIN**. Because the intervening **IF**s have left their locations on the stack, the backwards branch must be assembled by **ROT JMP**. The **ROT** (executed at assembly time) pulls the address left by **BEGIN** to the top of the stack, where it is used as **JMP**'s destination. Finally, the **THEN**s fill in the destination of the **IF**s.

Glossary

BEGIN *(— addr)* common usage

Push the address of the top of the dictionary onto the stack.

UNTIL *(addr x —)* common usage

Assemble a conditional jump back to the address left by **BEGIN**, using the system-dependent condition specifier *x*. The jump is taken if the condition is met. Common condition codes are **0=** and **0<**, as appropriate to various CPUs.

NOT *(x_1 — x_2)* common usage

Invert the action taken for a condition code.

IF *(x — addr)* common usage

Assemble a conditional forward jump, using the system-dependent condition specifier *x*, to be taken if the preceding condition is *false*, leaving the address of this instruction on the stack.

ELSE *($addr_1$ — $addr_2$)* common usage

Resolve the destination of **IF**'s jump (at $addr_1$) and assemble an unconditional forward jump (whose location $addr_2$ is left on the stack).

THEN *(addr —)* common usage

Resolve the destination for a jump instruction whose location is on the stack at assembly time (left by **IF** or **ELSE**).

5.9 LITERALS

Some processors allow you to define instructions to reference literals. For these, the standard Forth word for identifying a literal is **#**. Thus the instruction:

```
1000 # 0 MOV
```

would move the literal 1000 into Register 0. A few processors allow a short instruction format for small literals and a long format for larger ones. In such cases, the Forth assembler automatically examines the literal and generates the appropriate format.

On processors that do not support direct reference to literals, one technique for supplying them is to compile a literal into the dictionary, then pass the literal's address to an instruction that references it by **HERE**. For example:

```
HERE 1000 ,
CODE FIX   0 MOV …
```

In this example the literal 1000 is placed in memory and its address is left on the stack by **HERE**. The **MOV** instruction assembles a reference to that address. When executed, the effect will be to move 1000 into Register 0.

5.10 DEVICE HANDLERS

Device handlers should be kept extremely short, including only the instructions required to pass a value to or from the stack, or to issue a command. Consider, for example, a self-scan character display interfaced to an RCA 1802 as Device 2. This is all that is needed to output one character from the top of the stack:

```
CODE (EMIT) ( c)   S INC   S SEX
   2 OUT   NEXT
```

In this example, **S INC** increments the stack pointer (to get the low-order byte), **S SEX** sets **S** as the output register, and **2 OUT** sends the character to the device, incrementing **S** again to complete a **POP**.

Given this hypothetical definition of **(EMIT)**, you could define **(TYPE)** in high-level Forth to display a string of characters whose byte address and

length are on the stack:

```
: (TYPE) ( a n)   0 DO  PAUSE  DUP C@
   (EMIT)  1+ LOOP  DROP ;
```

To convert and display a number on the stack, you could define **SHOW**:

```
: SHOW ( n)   (.) (TYPE) ;
```

Here **(.)** performs the conversion, leaving the address and length of the resulting string for **(TYPE)**. The point here is that, given the simple code definition **(EMIT)**, full control of the display is available in high-level Forth.

Device drivers are highly variable in nature, depending upon both the processor and the actual device. You'll find a discussion of drivers for your processor in your product documentation and useful examples in the system listings.

5.11 INTERRUPTS

A multitasked Forth (plus a few standard conventions) makes dealing with interrupts* relatively simple. The principle strategy is to perform only the most time-critical actions at interrupt time, to notify the task responsible for the interrupting device that the interrupt has occurred, and to defer all complex logic to high-level routines executed by that task. The notification may take the form of setting a flag, incrementing or decrementing a counter, or modifying the task's status such that it will become active at the next opportunity in the multiprogrammer cycle.

The basic form of an interrupt handler is as follows:

```
ASSEMBLER BEGIN <code instructions> <dev#> INTERRUPT
```

where **ASSEMBLER** selects the assembler vocabulary (**CODE** would do this for you, but should not be used because it builds an unneeded header); **BEGIN** pushes onto the stack the address of the beginning of the code (which will be used by the word **INTERRUPT**); the *code instructions* perform the necessary work of the routine; *dev#* is the device code or interrupt vector to which the

* Most Motorola processors, as well as others, use the term "exceptions." On such processors, the word **INTERRUPT** would be replaced by **EXCEPTION**.

routine will respond, and **INTERRUPT** is a special code ending macro that assembles the appropriate return-from-interrupt instruction and puts the address of the code supplied by **BEGIN** into the interrupt vector.

The actual implementation of **INTERRUPT** is highly processor dependent. On machines with hardware-vectored interrupts, the implementation merely stores the address of the code in the specified vector address. On such machines, interrupts incur no additional overhead: only the instructions in the interrupt routine itself are executed. On machines in which software must identify the interrupting device, the identification method is system specific—consult your product documentation. However, one popular method is to put the polling routines in a chain, with the CPU's interrupt vector pointing to the first polling routine. If the device served by the first routine did not generate the interrupt, the first routine executes a jump to the second routine.

On every system, conventions are established for the use of registers at interrupt time. On most systems, you may not use any registers without saving and restoring them. Save and restore only the registers you are actually going to use! The usual place to save registers is on the return stack; on systems with only one hardware stack, the parameter stack becomes the place of choice. On systems with software vectors and few registers, one or two registers are routinely saved and restored so that you may use them freely. Consult your product documentation for details.

5.12 EXAMPLE

As an example of the action of the assembler, consider the definition of the high-level comparison operator **0=**. This word expects a value on the stack. If the value is non-zero, it will be replaced by a zero (*false*); if it is zero, it will be replaced by a negative one (*true*). The code for this routine in SwiftX for the Motorola 68HC12 is:

```
CODE 0= ( n -- flag )    0 # LDX    0 ,Y LDD
        0= IF    DEX    THEN
        0 ,Y STX
        RTS    END-CODE
```

The example below shows the processor during compilation and execution of

this routine (see your product documentation for details of the disassembler).

```
SEE 0=
81CE    0 # LDX
81D1    0 ,Y LDD
81D3    81D6 BNE
81D5    DEX
81D6    0 ,Y STX
81D8    RTX
```

6. Programming Style and Editing Standards

In this section, we will explore some of the issues that make Forth code easier to read and to maintain, notably source formatting standards and naming conventions. In addition, we are reprinting a set of "rules for readable Forth," published by Leo Wong on the Internet newsgroup comp.lang.forth.

Successful Forth programming groups generally acknowledge the importance of agreeing within the group on a single set of coding standards. This contributes significantly to long-term code maintainability, and facilitates code-sharing within the group, because all group members become comfortable reading the group's code.

Two sets of source guidelines are provided, one for **BLOCK**-based source (described in Section 3.4) and one for file-based source (described in Section 3.5). The file-based source guidelines are recommended by FirmWorks for use with Open Firmware. Open Firmware (IEEE Std. 1275-1994) is a Forth-based system for use in boot firmware used on SPARC systems, PowerPC PCI bus systems, and others. You will notice that, in Section 6.2, Forth words are spelled in lower case. This is conventional in Open Firmware and some Forth systems, although traditionally (and elsewhere in this book) upper case has been used for standard Forth words. This issue should be addressed in your group's coding standards.

Style and readability are highly subjective matters. We encourage you to modify the guidelines in this section to suit your own taste and the consensus of your group. The important thing is to have *some* set of standards, and follow it consistently!

6.1 GUIDELINES FOR BLOCK-BASED SOURCE

The purpose of this section is to describe a set of standards used for editing block-based Forth source code, to ensure readability and notational consistency.

6.1.1 Stack Effects

1. Any colon or code definition which expects or leaves data stack arguments must include a comment identifying them.
2. The format of the comment is:

   ```
   ( input - output )
   ```

   with the rightmost item on each side of the dash representing the top item on the stack. If there is input but no output, you may omit the dash.

Example 1	`: TYPE ( a n)`	(input only)
Example 2	`: -FOUND ( - a a' t)`	(output only)
Example 3	`CODE @ ( a - n)`	(input and output)

3. The stack comment begins one space after the name of the word. The terminating parenthesis should follow the last character, without a space. Exactly three spaces follow the right parenthesis before the code begins, if it begins on the same line. Remember to leave one space after the opening `(`.
4. The specific notation used to represent each stack item should follow these conventions:

`a`	address
`b`	eight-bit byte
`c`	ASCII character
`n`	single-precision number, usually signed
`u`	single-precision unsigned number
`t`	Boolean truth flag (0=*false*)

   The following special cases should be used when appropriate:

`l c`	Screen position, in lines and columns (always in that order).
`s d n`	Source, destination, count (always in that order).
`y x`	2-vector (*x,y* coordinate pairs, e.g., for graphics).

`f l` First, last limits, inclusive.
`f l+1` First, last limits, exclusive at end.
`c t` Cylinder, track (for disk drivers).

Other special situations may be dealt with similarly if necessary to improve clarity, but use single characters where possible.

5. Where several arguments are of the same type and clarity demands that they be distinguished, use `'` (prime) or suffix numerals. For example:

```
CODE RSWAP ( n a a' - n a)
CODE RSWAP ( n a1 a2 - n a1)
```

both show that the address returned is the same as the first one input.

6.1.2 General Comments

1. All source files must start with a comment which succinctly describes their contents. Some examples of good and bad style follow:

good: **`( Double-precision arithmetic)`**
wordy: **`( This code contains double-precision operators)`**
useless: **`( Misc. OPS)`**

2. Comments within source (other than stack effects) should be restricted to situations in which a serious ambiguity needs to be resolved.

good: **`177566 ( Send +2)`** and **`177562 ( RCV+2)`**
redundant: **`DUP 0= ABORT" Value is zero" ( Aborts if zero)`**
unhelpful: **`S ) 0 MOV ( Move top of stack to R0)`**

3. Comments should begin with a capital letter and otherwise be lower case, except as standard usage indicates, e.g.,

```
( Defining words)
( RX01 Bootstrap)
```

6.1.3 Spacing Within Source

1. Blank lines within source are valuable. Use them to separate definitions or groups of definitions. Avoid a dense clump of lines at the top of a file with a

lot of blank lines below, unless the clump is a single definition. Never have two blank lines together except at the end.

2. Definitions should begin in the left-most column of a line, except that two or three related **VARIABLE**s, **CONSTANT**s, or other data items may share a line if there is room for three spaces between them.
3. The name of a definition must be separated from its defining word by only one space. If it is a **CONSTANT** or other data item with a specified value, the value must be separated from the defining word by only one space.
4. Within a colon definition, three spaces are required after the stack comment. Thereafter, words are separated by one space, except when a second space is added between groups of closely related words.
5. Second and subsequent lines of colon and **CODE** definitions must be indented by multiples of three spaces (e.g., 3, 6, 9). Indentation beyond one set of three spaces indicates nested structures.

Examples of Forth in documentation should conform to these rules.

6.2 Open Firmware Coding Style

This section describes the coding style in some Open Firmware implementations. These guidelines are a "living" document that first came into existence in 1985. By following these guidelines in your own code development, you will produce code that is similar in style to a large body of existing Open Firmware work. This will make your code more easily understood by others within the Open Firmware community.

6.2.1 Typographic Conventions

The following typographic conventions are used in this document:

- The symbol _ is used to represent space characters (i.e., ASCII 0x20).
- The symbol ... is used to represent an arbitrary amount of Forth code.
- Within prose descriptions, Forth words are shown in **this** font.

6.2.2 Use of Spaces

Since Forth code can be very terse, the judicious use of spaces can increase the readability of your code.

Two consecutive spaces are used to separate a definition's name from the beginning of the stack diagram, another two consecutive spaces (or a new line) are used to separate the stack diagram from the word's definition, and two consecutive spaces (or a new line) separate the last word of a definition from the closing semi-colon. For example:

```
: new-name__(_stack-before_--_stack-after_)__foo__bar__;
: new-name__(_stack-before_--_stack-after_)
__foo_bar_framus_dup_widget_foozle_ribbit_grindle
;
```

Forth words are usually separated by one space. If a phrase consisting of several words performs some function, that phrase should be separated from other words/phrases by two consecutive spaces or a new line.

```
: name__(_stack before_--_stack after_)__xxx_xxx__xxx_xxx__;
```

When creating multiple line definitions, all lines except the first and last should be indented by three (3) spaces. If additional indention is needed with control structures, the left margin of each additional level of indentation should start three (3) spaces to the right of the preceding level.

```
: name (stack before -- stack after)
__xxx...
_____xxx...
_____xxx...
__xxx...
```

6.2.3 Conditional Structures

In **if...then** or **if...else...then** control structures that occupy no more than one line, two spaces should be used both before and after each **if**, **else**, or **then**.

```
__if__xxx__then__
__if__xxx__else__xxx__then__
```

Longer constructs should be structured like this:

```
<code to generate flag>__if
___<true clause>
then
<code to generate flag>__if
___<true clause>
else
___<false clause>
then
```

6.2.4 `do…loop` Structures

In **`do…loop`** constructs that occupy no more than one line, two spaces should be used both before and after each **`do`** or **`loop`**.

```
<code to calculate limits>__do__xxx__loop__
```

Longer constructs should be structured like this:

```
<code to calculate limits>__do
___<body>
loop
```

The longer **`+loop`** constructs should be structured like this:

```
<code to calculate limits>__do
___<body>
<incremental value>_+loop
```

6.2.5 `begin…while…repeat` Structures

In **`begin…while…repeat`** constructs that occupy that occupy no more than one line, two spaces should be used both before and after each **`begin`**, **`while`**, or **`repeat`**.

```
__begin__<flag code>__while__<body>__repeat__
```

Longer constructs:

```
begin__<short flag code>__while
___<body>
repeat
begin
___<long flag code>
while
___<body>
repeat
```

6.2.6 `begin…until…again` Structures

In **`begin…until`** and **`begin…again`** constructs that occupy no more than one line, two spaces should be used both before and after each **`begin`**, **`until`**, or **`again`**.

```
__begin__<body>__until
__begin__<body>__again
```

Longer constructs:

```
begin
___<body>
until
begin
___<body>
again
```

6.2.7 Block Comments

Block comments begin with `\_`. All text after the space is ignored until after the next new line. It would be possible to delimit block comments with parentheses, but the use of parentheses is reserved by convention for stack comments.

Precede each non-trivial definition with a block comment giving a clear and concise explanation of what the word does. Put more comments at the very

beginning of the file to describe external words which could be used from the User Interface.

6.2.8 Stack Comments

Stack comments begin with **(_** and end with **)**. Use stack liberally within definitions. Try to structure each definition so that, when you put stack comments at the end of each line, the stack picture makes a nice pattern.

```
: name (stack before -- stack after)
___xxx xxx bar ( stack condition after the execution of bar)
___xxx xxx foo ( stack condition after the execution of foo)
___xxx xxx dup ( stack condition after the execution of dup)
```

6.2.9 Return Stack Comments

Return stack comments are also delimited with parentheses. In addition, the notation **r:** is used at the beginning of the return stack comment to differentiate it from a data stack comment.

Place return stack comments on any line that contains one or more words that cause the return stack to change. (This limitation is a practical one; it is often difficult to do otherwise due to lack of space.) The words **>r** and **r>** must be paired inside colon definitions and inside **do…loop** constructs.

```
: name ( stack before -- stack after )
___xxx >r ( r:addr )
___xxx r> ( r: )
```

6.2.10 Numbers

Hexadecimal numbers should be typed in a lower case. If a given number contains more the four digits, the number may be broken into groups of four digits with periods. For example:

```
dead.beef
```

All literal numbers should have a preceding h# (for hex) or d# (for decimal). The only exception is in tables, where the number base is explicitly specified. For example:

```
hex
create foo
   1234 ,  abcd ,  56ab ,  8765 ,
   0023 ,  ...
```

6.3 Wong's Rules for Readable Forth

Following is a set of rules for readable Forth posted in `comp.lang.forth` by:

Leo Wong (`hello@albany.net`)
New York State Department of Civil Service
Albany, NY 12239

with additional commentary by Wil Baden, including quotes from Leo Brodie (author of the popular tutorial *Starting Forth;* see Appendix A *Bibliography)*. These rules are not provided here as definitive guidelines—they are presented in order to provoke thought about which approaches may be most useful in your own programming practice.

1. Use the word that fits the data.
2. Do not use ASCII codes (or other "magic numbers") in colon definitions.
3. Do not factor just to factor.
4. Get all three right: code, comment, and name.
5. Do not use syntactic sugar.
6. Eschew sophistication.
7. Test it, even if it's obvious.
8. Do not shun Scylla by falling into Charybdis*.
9. Feature the stack machine.

* In the *Odyssey*, the mighty Ulysses was required, at one point, to sail through a strait that was guarded on one side by the many-headed monster Scylla, and on the other by Charybdis, the whirlpool of oblivion. Many sailors tried so hard to avoid the one that they succumbed to the other.

10. Pattern names after other names.

These rules are not for beginners learning their Forth ABCs, but might be helpful to a person who has written a program and wants to make it clearer.

6.3.1 Example: Magic Numbers

Here are some examples:

```
: STAR   42 EMIT ;
: STAR   ." *" ;
: STAR   [CHAR] * EMIT ;
```

Each of these definitions has a fault:

- The first forces the reader to know that ASCII 42 means * (although this could be remedied by a comment).
- The second uses a word intended for strings.
- The third is wordy in the source, although it compiles the same result as the first.

I don't consider the lack of a stack comment in these definitions a fault. I find the third **STAR** to be the most readable and, hence, preferable.

Two rules:

1. Use the word that fits the data.
2. Do not use ASCII codes or other unidentified numbers in colon definitions.

6.3.2 Example: Factoring

From the first edition of *Starting Forth* (p. 43):

```
: QUARTERS   4 /MOD  . ." ONES AND "  . ." QUARTERS " ;
```

From the second edition (p. 40):

```
: $>QUARTERS  ( dollars -- quarters dollars)  4 /MOD ;
: .DOLLARS  ( dollars -- )  .  ." dollar bills" ;
: .QUARTERS  ( quarters -- )  .  ." quarters " ;
: QUARTERS  ( dollars -- )
   $>QUARTERS  ." Gives " .DOLLARS  ." and " .QUARTERS ;
```

In the second edition, a name and two stack comments are wrong (as is the output, not shown here). In addition, this approach is both larger and slower without contributing significantly to readability or functionality.

Rules:

3. Do not factor just to factor.
4. Get all three right: code, comment, and name.

6.3.3 Example: Simplicity

Two solutions adapted from *Starting Forth*, 2nd edition (pp. 277–278):

(1)

```
: bdot"   BRIGHT  R>  COUNT  2DUP + >R  TYPE  -BRIGHT ;
: B."   POSTPONE bdot"  [CHAR] " WORD  C@ 1+ ALLOT ;
IMMEDIATE
```

Brodie: "The foregoing solution is messy and probably not transportable."

[Note: transportability is limited by the assumptions this approach makes about the implementation]

(2)

```
: B."  POSTPONE BRIGHT  POSTPONE ."  POSTPONE -BRIGHT ;
IMMEDIATE
```

Brodie: "The disadvantage of this solution over the previous one is that every invocation of **`B."`** compiles two extra addresses. The first solution is more efficient and therefore preferable if you have the system source listing and lots of invocations of **`B."`**. The second solution is simpler to implement, and adequate for a small number of invocations.

"Other languages may be easier to learn; but what other languages let you extend the compiler like this?"

An alternative that doesn't include the compilation features might be:

```
: .BRIGHT  ( a u)  BRIGHT TYPE -BRIGHT ;
```

Rules:

5. Do not use syntactic sugar.
6. Eschew sophistication.

6.3.4 Example: Testing Assumptions

In *Thinking Forth* (reprint edition, p. 219), Brodie quotes Moore:

"In books you often see a lot of piece-wise linear approximations that fail to express things clearly. For instance the expression

x = 0 for t < 0
x = 1 for t ≥ 0

"This would be equivalent to:

```
t 0< 1 AND
```

"as a single expression, not a piece-wise expression."

Rule:

7. Test it even if it's obvious.

6.3.5 Example: `IF` Avoidance

Forth programmers strive to avoid **`IF`**, some going so far as to use **`CASE`** whenever possible. Here are two examples, from *Starting Forth,* of **`IF`**-avoidance:

First the **IF** versions (second edition, p. 183):

```
: CATEGORY  ( weight-per-dozen -- category#)
   DUP 18 < IF  0  ELSE
   DUP 21 < IF  1  ELSE
   DUP 24 < IF  2  ELSE
   DUP 27 < IF  3  ELSE
   DUP 30 < IF  4  ELSE
                5
   THEN THEN THEN THEN THEN  SWAP DROP ;
```

(Note: the "official table" on which **CATEGORY** is based is ambiguous. See p. 85.)

```
: LABEL  ( category# -- )
   DUP  0= IF  ." Reject "       ELSE
   DUP 1 = IF  ." Small "        ELSE
   DUP 2 = IF  ." Medium "       ELSE
   DUP 3 = IF  ." Large "        ELSE
   DUP 4 = IF  ." Extra Large "  ELSE
                ." Error "
   THEN THEN THEN THEN THEN  DROP ;
```

Now the "simple and elegant for experts" versions (pp. 189 and 253):

```
CREATE SIZES  18 C,  21 C,  24 C,  27 C,  30 C,  255 C,
: CATEGORY  ( weight-per-dozen -- category# )
   6 0 DO  DUP  SIZES I + C@
   < IF  DROP  I LEAVE  THEN  LOOP ;

CREATE "LABEL"
ASCII " STRING Reject  Small   Medium  Large   Xtra LrgError  "
: LABEL  (  category# -- )
   8 *  "LABEL"  +  8 TYPE SPACE ;
: LABEL   0 MAX  5 MIN  LABEL ;
```

It may seem unfair of me to give the code without the explanations, but:

- Experts wouldn't need the explanations.
- I would have to mention the bugs in the elegant **LABEL**.

Which versions would you rather maintain?

Rule:

8. Do not shun Scylla by falling into Charybdis.

6.3.6 Example: Stack Music

What is stack noise to you and me is music to a stack machine. It is time to face the music.

In *Thinking Forth,* Brodie gives a solution (reprint edition, p. 222) to a phone-rate problem posed and analyzed earlier in the book (pp. 45–51):

```
\ Telephone rates                              03/30/84
CREATE FULL     30 ,  20 ,  12 ,
CREATE LOWER    22 ,  15 ,  10 ,
CREATE LOWEST   12 ,   9 ,   6 ,
VARIABLE RATE   \ Points to FULL, LOWER or LOWEST
                \ depending on time of day
FULL RATE !     \  For instance
: CHARGE   ( o -- )  CREATE ,
   DOES>   ( -- rate )  @  RATE @ +  @ ;
0 CHARGE 1MINUTE    \ Rate for first minute
2 CHARGE +MINUTES   \ Rate for each additional minute
4 CHARGE /MILES     \ Rate per each 100 miles

\ Telephone rates                              03/30/84
VARIABLE OPERATOR?  \ 90 if operator assisted; else 0
VARIABLE #MILES     \ Hundreds of miles
: ?ASSISTANCE  ( Direct-dial charge -- total charge)
   OPERATOR? @ + ;
: MILEAGE  ( -- charge )  #MILES @  /MILES * ;
: FIRST  ( -- charge )  1MINUTE  ?ASSISTANCE  MILEAGE + ;
: ADDITIONAL  ( -- charge)  +MINUTES  MILEAGE + ;
: TOTAL  ( #minutes -- total charge)
   1- ADDITIONAL *  FIRST + ;
```

No stack noise. Readable?

Here's a try at a solution that requires stack manipulations:

```
\ Phone-rate table from Brodie, Thinking Forth,
\ reprint edition, p. 51
```

```
\ Rates are used as offsets into arrays
0 CELLS CONSTANT FULL
1 CELLS CONSTANT LOWER
2 CELLS CONSTANT LOWEST
\ Array-defining word
: FOR   CREATE  DOES>  ( rate - charge-per-minute)  + @ ;
\ Table comprises three arrays
\ Charge-per-minute at FULL  LOWER LOWEST  rate
    FOR FIRST           30 ,  22 ,  12 ,
    FOR +MINUTES        20 ,  15 ,   9 ,
    FOR DISTANCE        12 ,  10 ,   6 ,

90 CONSTANT ASSISTANCE  \ Charge for operator assistance
: ?ASSISTANCE  ( flag - charge)  ASSISTANCE AND ;
: ADDITIONAL  ( #minutes-1 rate - charge)  +MINUTES * ;
: MINUTES  ( #minutes rate - charge)
   DUP FIRST  ROT 1- ROT  ADDITIONAL + ;
: MILES  ( distance #minutes rate - charge)  DISTANCE * * ;
: TOTAL  ( distance #minutes rate assistance-flag - charge)
   ?ASSISTANCE >R  2DUP MINUTES >R  MILES  2R> + + ;
```

Stack music. Unreadable?

Rule:

9. Feature the stack machine.

6.3.7 Summary

How do we feel about these rules? Are any of them helpful? Hurtful? Are there better rules? Do we want rules anyway? These are questions for you to answer, should you so choose.

6.4 Naming Conventions

Table 10 presents some naming conventions that have been widely used in Forth for many years. These take advantage of Forth's flexible naming rules to use special characters to convey additional meaning.

In this table, the word *name* refers to some word the programmer has chosen to

represent a Forth routine.

Where possible, a prefix before a name indicates the type or precision of the value being operated on, whereas a suffix after a name indicates what the value is or where it's kept.

Table 10: Naming conventions

Prefixes	Meaning	Examples
`!name`	Store into name	**`!DATA`**
`#name`	Size or quantity	**`#PIXELS`**
	Output numeric operator	**`#S`**
	Buffer name	**`#I`**
`'name`	Address of name	**`'S`**
	Address of pointer to name	**`'TYPE`**
`(name)`	Internal component of name, not normally user-accessible	**`(IF)`**
		`(FIND)`
	Run-time procedure of name	**`(:)`**
	File index	**`(PEOPLE)`**
`*name`	Multiplication	**`*DIGIT`**
	Takes scaled input parameter	**`*DRAW`**
`+name`	Addition	**`+LOOP`**
	Advance	**`+BUF`**
	Enable	**`+CLOCK`**
	More powerful	**`+INITIALIZE`**
	Takes relative input parameters	**`+DRAW`**
`-name`	Subtract, remove	**`-TRAILING`**
	Disable	**`-CLOCK`**
	`not` name (opposite of name)	**`-DONE`**
	Returns reversed truth flag (1 is *false*, 0 is *true*)	**`-MATCH`**
	Pointers, especially in files	**`-JOB`**

Table 10: Naming conventions *(continued)*

Prefixes	Meaning	Examples
.name	Print named item	.S
	Print from stack in named format	.R .$
	Print following string	." string"
	May be further prefixed with data type	D. U. U.R
/name	Division	/DIGIT
	Initialize routine or device	/COUNTER
	"per"	/SIDE
1name	First item of a group	1SWITCH
	Integer 1	1+
	One-byte size	1@
2name	Second item of a group	2SWITCH
	Integer 2	2/
	Two-cell size	2@
;name	End of something	;S
	End of something, start of something else	;CODE
<name	Less than	<LIMIT
	Open bracket	<#
	From device name	<TAPE
<name>	Name of an internal part of a device driver routine	<TYPE>
>name	Towards name	>R, >TAPE
	Index pointer	>IN
	Exchange, especially bytes	>< (swap bytes)
		>MOVE< (move, swapping bytes)
?name	Check condition, return *true* if yes	?TERMINAL
	Conditional operator	?DUP
	Check condition, abort if bad	?STACK
	Fetch contents of name and display	?N

Table 10: Naming conventions *(continued)*

Prefixes	Meaning	Examples
@name	Fetch from name	**@INDEX**
Cname	One-byte character size, integer	**C@**
Dname	Double-cell integer	**D+**
Mname	Mixed single and double operator	**M***
Tname	Three-cell size	**T***
Uname	Unsigned encoding	**U.**
[name]	Executes at compile time	**[']**
\name	Unsigned subtraction (ramp-down)	**\LOOP**
name!	Store into name	**B!**
name"	String follows, delimited by "	**ABORT" xxx"**
name,	Put something into dictionary	**C,**
name:	Start definition	**CASE:**
name>	Close bracket	**#>**
	Away from name	**R>**
name?	Same as ?name	**B?**
name@	Fetch from name	**B@**

APPENDIX A: BIBLIOGRAPHY

American National Standard For Information Systems: Programming Languages – Forth (ANSI X3.215–1994). American National Standards Institute, 11 W. 42nd St., New York, NY 10036, (212) 642-4900.

Bailey, G., Sanderson, D., Rather, E. "clusterFORTH, A High-Level Network Protocol" *Proceedings of the 1984 FORTH Conference*. Rochester, NY: The Institute for Applied Forth Research, 1984.

Brodie, L. *Starting FORTH*, Englewood Cliffs, NJ: Prentice-Hall, 1981, 2nd ed. 1987. Contact: Forth Interest Group, 100 Dolores St., Suite 183, Carmel, California 93923.

Brodie, L. *Thinking FORTH*, Englewood Cliffs, NJ: Prentice-Hall, 1984, reprinted by the Forth Interest Group, 100 Dolores St., Suite 183, Carmel, California 93923. 1994.

ISO/IEC 15145:1997: *Information technology—Programming languages—FORTH*. This is the International Standard equivalent of ANS Forth. In the U.S., it is available through the American National Standards Institute, 11 W. 42nd St., New York, NY 10036, (212) 642-4900. For sources in other countries or on-line ordering, see http://www.iso.ch.

Kelly, M.G., and Spies, N. *FORTH: A Text and Reference*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Koopman, P. *Stack Computers, The New Wave*. Chichester, West Sussex, England. Ellis Horwood Ltd. 1989

Martin, T. *A Bibliography of Forth References*, 3rd ed. Rochester, NY: Institute for Applied Forth Research, 1987.

Moore, C.W. "The Evolution of Forth — An Unusual Language" *Byte*, August 1980.

Noble, J.V. *Scientific Forth*. Charlottesville, VA: Mechum Banks Publishing, 1992.

Pountain, R. *Object Oriented Forth*. New York: Academic Press, 1987.

Rather, E.D. "Forth Programming Language" *Encyclopedia of Physical Science & Technology*

(V. 5) Academic Press, Inc., 1987, 1992.

Rather, E.D. "Fifteen Programmers, 400 Computers, 36,000 Sensors and Forth" *Journal of Forth Application and Research* (V. 3, #2, 1985), P.O. Box 27686, Rochester, NY 14627.

Rather, E.D., Colburn, D.R., and Moore, C.W. "The Evolution of Forth" *ACM SIGPLAN Notices*, Vol. 28, No. 3, March 1993.

Terry, J.D. *Library of Forth Routines and Utilities*. New York: Shadow Lawn Press, 1986.

Tracy, M. and Anderson, A. *Mastering Forth* (2nd ed.). New York: Brady Books, 1989.

APPENDIX B: GLOSSARY & NOTATION

This section describes technical terms and notational conventions used in this manual. Additional notation specific to certain sections is described in those sections.

In this manual, the words "shall" and "must" indicate mandatory behavior. The word "will" indicates predicted or consequential behavior. The word "may" indicates permitted or desirable, but not mandatory, behavior. The phrase "may not" indicates prohibited behavior.

B.1 ABBREVIATIONS

ANS	American National Standard
BCD	Binary Coded Decimal
CPU	Central Processing Unit
H	Hexadecimal (base 16), when used as a subscript
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
K	1024
LSB	Least-Significant Bit
N/A	Not Applicable
OS	Operating System
PC	Personal Computer
PROM	Programmable Read-Only Memory
RAM	Random-Access Memory
ROM	Read-Only Memory

B.2 GLOSSARY

address unit — In Standard Forth, the units in which the length of a region of memory is expressed, or the units into which the region is divided for the purpose of locating data objects. These are nearly always bytes, and in this manual will be referred to as simply bytes.

aligned address — The address of a memory location at which a character, cell, cell pair, or double-cell integer can be accessed. For cell-aligned addresses, the address is evenly divisible by the cell size in bytes.

ANS Forth — The Forth programming language as defined by the American National Standard X3.215, 1994.

ASCII string — A string whose data contains one ASCII character per byte. An ASCII string is specified by a cell pair representing its starting address and length in bytes.

big-endian — Describes a CPU's byte-ordering system in which the highest-order byte of a cell is at the lowest address (i.e., appears first in a data stream). *Little-endian* is the converse of this. Motorola processors are big-endian and Intel processors are little-endian.

cell — The primary unit of information storage in the architecture of a Forth system. The word length of the processor is always referred to as a cell. This is also the size of an address, and is the size of a single item on Forth's stacks.

cell pair — Two cells that are treated as a single unit. The cells may contain a double-length number, two related single-length numbers (such as a 2-vector), or two entirely unrelated values. In memory, a cell pair is contiguous; the cell at the lower address is the "first" cell, and its address identifies the pair. Unless otherwise specified, a cell pair on the stack has the first cell immediately above the second cell.

character — In Standard Forth, one meaning of this word is the number of address units needed to store a character. In this manual, characters are assumed to occupy one byte each. The length of a character string in bytes is, therefore, equal to the number of characters in it (plus one if it is a *counted string*—see below).

character-aligned address — In Standard Forth, the address of a memory location at which a character can be accessed. In nearly all implementations, a character occupies a single byte and, thus, this will be an arbitrary byte address.

code space The logical area of the dictionary in which word definitions are implemented.

compile Transform source code into dictionary definitions.

compilation behavior The behavior of a Forth definition when its name is encountered by the text interpreter in compilation state.

counted string A data structure consisting of one character containing the length followed by 0–255 data characters. A counted string in memory is identified by the address of its length character.

cross-compilation Generation of an executable program for a target CPU on a host system that may be based on a different CPU.

data field The data space associated with a word defined via **`CREATE`**.

data space The logical area of the dictionary that can be accessed.

data space pointer The address of the next available data space location. Also called the *dictionary pointer*. The Forth word **`HERE`** returns this value.

data stack A stack that may be used for passing parameters between procedures. When there is no possibility of confusion, the data stack is referred to simply as "the stack." Contrast with *return stack*.

defining word A Forth word that creates a new definition when executed.

definition A Forth execution procedure compiled into the dictionary.

dictionary An extensible structure containing definitions and associated data space.

double-cell integer A double-precision integer, signed or unsigned, occupying two cells. On the stack, the most-significant cell is above the least-significant cell. In memory, the most-significant cell is normally at the lower address, independent of processor type (see *big-endian*, above). Placing a single-cell integer zero on the stack above a single-cell unsigned integer produces a double-cell unsigned integer with the same value.

exception frame The implementation-dependent set of information recording the current execution state, necessary for exception processing using the Forth words **`CATCH`** and **`THROW`**.

exception stack A stack used for nesting exception frames. It may be, but need not be, implemented using the return stack.

execution behavior The behavior of a Forth definition when it is executed.

execution token A single-cell value that identifies the execution behavior of a procedure. Multiple definitions may have the same execution token if the definitions have equivalent execution behaviors.

flag A single-cell Boolean true/false value. A word using a flag as input treats zero as *false*, and any non-zero value as *true*. A word returning a flag returns either all bits zero (*false*) or all bits one (*true*).

immediate word A Forth word whose compiling behavior is to perform its execution behavior. Commonly used to compile program-flow structures.

input stream ASCII string data input to the host interpreter. It may come from an input device (such as a keyboard) or from a file. The input stream is the vehicle by which user commands, program source, and other data are provided to the host system.

interpretation behavior The behavior of a Forth definition when its name is encountered by the text interpreter in interpretation state.

keyboard event A value received by a Forth system as a result of a user action at the user input device. This manual's use of the word *keyboard* does not exclude other types of user input devices.

name space The logical area of the dictionary in which definition names are stored during compilation and testing in the host computer.

number In this manual, *number* used without qualification means "integer." *Double number* or *double-precision number* means "double-cell integer."

parse To select and exclude a character string from the parse area using a specified set of delimiting characters, called delimiters.

parse area The portion of the input stream that has not yet been processed by the host interpreter and is, therefore, available for processing by the host interpreter and other parsing operations.

return stack A stack that may be used for program execution nesting, `DO` loop execution, temporary storage, and other purposes.

stack An area in memory containing a last-in, first-out list of items.

Standard Forth The term used to refer to a Forth system that complies with the *ISO/IEC 15145:1997* and *ANSI X3.215:1994* standards for the Forth programming language.

variable A named region of data space, located and accessed by its memory address.

whitespace character	A blank or non-graphic character encountered by the Forth interpreter while processing source in a text file.
word	The name of a Forth definition. In the text interpreter, *word* can also refer to a sequence of non-space characters to be processed.
word list	A list of related Forth definition names that may be examined during a dictionary search. A word list is a subset of the entire Forth dictionary.

B.3 Data Types in Stack Notation

Table 11 gives a description of the Standard Forth notation used to refer to the different data types that may appear in stack notation or descriptions in this manual. Additional tables in this section describe other notational conventions.

Table 11: Notation for the data type of stack arguments

Symbol	Data type	Size on stack
a-addr	A byte address that is cell-aligned (the address is evenly divisible by the cell size in bytes).	1 cell
addr	Address.	1 cell
b	A byte, stored as the least-significant eight bits of a stack entry. The remaining bits of the stack entry are zero in results, and are ignored in arguments.	1 cell
c or *char*	An ASCII character, stored as a byte (see above) with the parity bit reset to zero.	1 cell
c-addr	A byte address that is character-aligned (on current systems a character is always one byte, so this amounts to an arbitrary byte address).	1 cell
d	A double-precision, signed, two's complement integer, stored as two stack entries (least-significant cell underneath the most-significant cell). On 16-bit machines the range is from -2**31 through +2**31-1. On 32-bit machines, the range is from -2**63 through +2**63-1.	2 cells
dest	Control-flow destination.	Implementation dependent

Table 11: Notation for the data type of stack arguments *(continued)*

Symbol	Data type	Size on stack
echar	Extended character (occupying the two low-order bytes of a stack item).	1 cell
flag	A single-precision, Boolean truth flag (zero means *false*, non-zero means *true*). See Section B.4 for details.	1 cell
*i*x, j*x, etc.*	Zero or more cells of unspecified data type; normally used to indicate that the state of the stack is preserved during, or is restored after, an operation.	Varies
ior	Result of an I/O operation. See Section 3.5.1 for use of *iors* in the file access words.	1 cell
len	Length of a string (0–65535). A counted string may not contain more than 255 characters plus count.	1 cell
loop-sys	Loop-control parameters. These include implementation-dependent representations of the current value of the loop index, its upper limit, and a pointer to a *termination location* where execution continues following an exit from the loop.	Implementation dependent
nest-sys	Implementation-dependent information for procedure calls. It may be kept on the return stack.	Implementation dependent
n	A signed, single-precision, two's complement number. On 16-bit machines, the range is -2**15 through +2**15-1. On 32-bit machines, the range is -2**31 through +2**31-1. If a stack comment is shown as *n*, *u* is implied, unless specifically stated otherwise (e.g., **+** may be used to add signed or unsigned numbers). If there is more than one input argument, signed and unsigned types may not be mixed.	1 cell
+n	A single-precision number that may not be negative and has the same positive upper limit as *n*, above. An input stack argument shown as *+n* must not be negative.	1 cell
orig	Control-flow origin.	Implementation dependent
u	An unsigned, single-precision integer, with the range 0 to 2**16 on 16-bit machines, or 0 through 2**32 on 32-bit machines.	1 cell

Table 11: Notation for the data type of stack arguments *(continued)*

Symbol	Data type	Size on stack
ud	An unsigned, double-precision integer, with the range 0 to 2**32 on 16-bit machines, or 0 through 2**64 on 32-bit machines.	2 cells
x	A cell (single stack item), otherwise unspecified.	1 cell
xt	Execution token. This is a value that identifies the execution behavior of a definition. When this value is passed to **EXECUTE**, the definition's execution behavior is performed.	1 cell

Some data types are sub-types of other data types. Figure 16 shows the hierarchy for single-cell and double-cell types. Any Forth definition that accepts an argument of a type shown in the figure must also accept all the subtypes below it. For example, a word with an input stack argument of type *n* also accepts arguments of type *+n* and *char*.

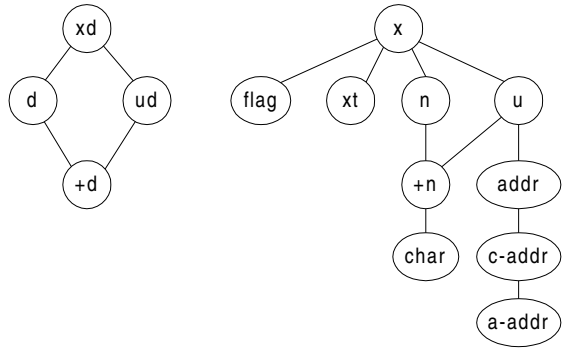

Figure 16. Hierarchy of data types

Standard Forth does not require data-type checking, and most implementations do not include it among their standard functions. Also, most implementations do not include arithmetic error checking on standard math functions (such as overflow on a multiply operation). The reason for both of these restrictions is that error checking and data-type checking on low-level functions could be prohibitively costly in execution time. Most Forth implementations do support whatever hardware error-detection functions exist, such as a trap for divide by zero, or the various exceptions signaled by the 80387/80486 floating-point processor. An application may, of course, build in error checking and/or type checking at any level deemed necessary, simply by redefining the words in question and adding an outer layer of protection.

B.4 Flags and IOR Codes

Procedures that accept flags as input arguments shall treat zero as *false*, and any non-zero value as *true*. A flag returned as an argument is a *well-formed* flag with all bits zero (*false*), or all bits one (*true*).

Certain device control and other functions return an *ior* (I/O Result) to report the results of an operation. An *ior* may be treated as a flag, in the sense that a non-zero value is *true*; however, it is not necessarily a well-formed flag, because its specific value often is used to convey additional information. A returned value of zero for an *ior* shall mean successful completion (i.e., no error); non-zero values may indicate an error condition or other abnormal status, and are device dependent.

B.5 Forth Glossary Notation

Words described in this manual are grouped functionally. An alphabetical list of all words is given in Appendix C.

Each entry consists of two parts: an index line and a semantic (behavioral) description of the word. The index line is a single-line entry containing, from left to right:

- Definition name, in upper-case, monospaced, boldface letters;
- Stack behavior (the notation follows the conventions described in Sections B.3 and B.4, above).

The first paragraph of the behavioral description concludes with the natural-language pronunciation of the word (shown in distinctive type), if it is not obvious.

A word's behavior may be context dependent. The behavior(s) for each such word are described, as applicable, for:

Compiling An action taken by the system when adding to the Forth dictionary.

***name* Execution** The behavior of *name* when executed, where *name* is an instance of a class of words created by a defining word (see Section 4.2).

Interpreting An action taken by the system when the name of a word is encountered by the text interpreter in interpretation state.

Run-time The behavior executed by the system.

While many words (such as defining words and compiler directives) possess specific compiling behaviors, the default compilation behavior of a word is to append its *execution behavior* to the current definition. Separate behaviors in different modes will be shown, where they differ.

Some words will be executed (i.e., will perform their behavior) when encountered in compiling mode. In Forth, these are known as *immediate* words. If execution of such a word will cause some run-time action in the word being compiled, this is shown as a separate run-time behavior.

APPENDIX C: INDEX TO FORTH WORDS

This section provides an alphabetical index to the Forth words appearing in the glossaries in this book. Each is shown with its stack arguments and a page reference, where you may find more information.

Stack operations are described in Section 2.1. The stack-argument notation is described in Appendix A, Table 11. Where several arguments are of the same type, and clarity demands that they be distinguished, numeric subscripts are used.

On the following pages, the "Wordset" column identifies the Standard Forth word list in which each word appears. "Core" words are required in all Standard Forth systems. Words marked "common usage" are not mentioned in Standard Forth, but may be found in many Forth systems. All other designations represent optional Standard Forth wordsets (groupings by logical function) that may be present in some systems. You may use **`ENVIRONMENT?`** (Section 3.2) to determine whether a particular optional wordset is present.

Name	Stack	Wordset	Page
`(`	(—)	Core, File	**23**
`.(`	(—)	Core Ext	**23**
`+`	(n_1 n_2 — n_3)	Core	**40**
`-`	(n_1 n_2 — n_3)	Core	**40**
`.`	(n —)	Core	**53**
`,`	(x —)	Core	**149**
`1+`	(n_1 — n_2)	Core	**41**
`1-`	(n_1 — n_2)	Core	**41**
`2+`	(n_1 — n_2)	common usage	**41**
`2-`	(n_1 — n_2)	common usage	**41**

Name	Stack	Wordset	Page
!	(*x a-addr* —)	Core	**35**
#	(ud_1 — ud_2)	Core	**56**
#>	(*ud* — *c-addr u*)	Core	**57**
#S	(ud_1 — ud_2)	Core	**56**
' <name>	(— *xt*)	Core	**38, 124**
*****	(n_1 n_2 — n_3)	Core	**40**
***/**	(n_1 n_2 n_3 — n_4)	Core	**40**
***/MOD**	(n_1 n_2 n_3 — n_4 n_5)	Core	**40**
+!	(*n a-addr* —)	Core	**35**
+LOOP	(*n* —)	Core	**65**
-TRAILING	(*c-addr* u_1 — *c-addr* u_2)	String	**50**
." <string>**"**	(—)	Core	**130**
.'	(*addr* —)	common usage	**27**
.R	(n_1 +n_2 —)	Core Ext	**53**
.S	(—)	Tools	**38**
/	(n_1 n_2 — n_3)	Core	**41**
/MOD	(n_1 n_2 — n_3 n_4)	Core	**41**
/STRING	(*c-addr*$_1$ u_1 +*n* — *c-addr*$_2$ u_2)	String	**50**
0<	(*n* — *flag*)	Core	**45**
0<>	(*n* — *flag*)	Core Ext	**46**
0=	(*n* — *flag*)	Core	**46**
0>	(*n* — *flag*)	Core Ext	**46**
2!	(x_1 x_2 *a-addr* —)	Core	**35**
2*	(x_1 — x_2)	Core	**41**
2/	(x_1 — x_2)	Core	**41**
2>R	(x_1 x_2 —) (*R:* — x_1 x_2)	Core Ext	**37**
2@	(*a-addr* — x_1 x_2)	Core	**35**
2CONSTANT <name>	(x_1 x_2 —)	Core	**138**
2DROP	(x_1 x_2 —)	Core	**33**

Name	Stack	Wordset	Page
2DUP	*(x_1 x_2 — x_1 x_2 x_1 x_2)*	Core	**33**
2LITERAL	*(— x_1 x_2)*	Double	**155**
2OVER	*(x_1 x_2 x_3 x_4 — x_1 x_2 x_3 x_4 x_1 x_2)*	Core	**33**
2R>	*(— x_1 x_2) (R: x_1 x_2 —)*	Core Ext	**37**
2R@	*(— x_1 x_2) (R: x_1 x_2 — x_1 x_2)*	Core Ext	**37**
2ROT	*(x_1 x_2 x_3 x_4 x_5 x_6 — x_3 x_4 x_5 x_6 x_1 x_2)*	Double Ext	**33**
2SWAP	*(x_1 x_2 x_3 x_4 — x_3 x_4 x_1 x_2)*	Core	**34**
2VARIABLE <name>	*(—)*	Double	**135**
: <name>	*(—)*	Core	**140**
:NONAME	*(— xt)*	Core Ext	**140**
;	*(—)*	Core	**140**
;CODE	*(—)*	Tools Ext	**144**
<	*(n_1 n_2 — flag)*	Core	**46**
<#	*(ud — ud)* or *(n ud — n ud)*	Core	**56**
<>	*(n_1 n_2 — flag)*	Core Ext	**46**
=	*(n_1 n_2 — flag)*	Core	**46**
>	*(n_1 n_2 — flag)*	Core	**46**
>BODY	*(xt — a-addr)*	Core	**124**
>FLOAT	*(c-addr u — true | false); (F: — r |)*	Floating	**116**
>IN	*(— a-addr)*	Core	**121, 128**
>NUMBER	*(ud_1 c-$addr_1$ u_1 — ud_2 c-$addr_2$ u_2)*	Core	**126**
>R	*(x —) (R: — x)*	Core	**37**
?	*(a-addr —)*	Tools	**38, 54**
?DO	*(n_1 n_2 —)*	Core Ext	**64**
?DUP	*(x — 0 | x x)*	Core	**34**
@	*(a-addr — x)*	Core	**35**
[	*(—)*	Core	**153**

Name	Stack	Wordset	Page
BLK	*(— a-addr)*	Block	**93, 120**
BLOCK	*(u — addr)*	Block	**91**
BUFFER	*(u — addr)*	Block	**91**
C!	*(b c-addr —)*	Core	**35**
C" <string>"	*(— c-addr)*	Core Ext	**130**
C+!	*(b c-addr —)*	common usage	**35**
C,	*(char —)*	Core	**149**
C@	*(c-addr — b)*	Core	**35**
CASE	*(—)*	Core Ext	**68**
CATCH	*(i*x xt — j*x 0 | i*x n)*	Exception	**76**
CELL+	*(a-addr$_1$ — a-addr$_2$)*	Core	**41**
CELLS	*(n$_1$ — n$_2$)*	Core	**41**
CHAR	*(— char)*	Core	**58**
CHAR+	*(c-addr$_1$ — c-addr$_2$)*	Core	**41**
CHARS	*(n$_1$ — n$_2$)*	Core	**41**
CLOSE-FILE	*(fileid — ior)*	File	**97**
CMOVE	*(c-addr$_1$ c-addr$_2$ u —)*	String	**51**
CMOVE>	*(c-addr$_1$ c-addr$_2$ u —)*	String	**51**
CODE <name>	*(i*x — j*x)*	Tools Ext	**170**
COMPARE	*(c-addr$_1$ u$_1$ c-addr$_2$ u$_2$ — n)*	String	**52**
COMPILE,	*(xt —)*	Core Ext	**153**
CONSTANT <name>	*(x —)*	Core	**138**
CONTEXT	*(— a-addr)*	Core	**167**
COUNT	*(c-addr$_1$ — c-addr$_2$ n)*	Core	**128**
CR	*(—)*	Core	**85**
CREATE <name>	*(—)*	Core	**134**
CREATE-FILE	*(c-addr u fam — fileid ior)*	Core	**98**
CS-PICK	*(i*x u — i*x x$_u$)*	Tools Ext	**163**
CS-ROLL	*(i*x u — (i-1)*x x$_u$)*	Tools Ext	**163**

Name	Stack	Wordset	Page
CURRENT	*(— a-addr)*	common usage	**167**
CVARIABLE <name>	*(—)*	common usage	**136**
D+	*(d_1 d_2 — d_3)*	Double	**42**
D-	*(d_1 d_2 — d_3)*	Double	**42**
D.	*(d —)*	Double	**54**
D.R	*(d +n —)*	Double	**54**
D0<	*(d — flag)*	Double	**46**
D0=	*(d — flag)*	Double	**46**
D2*	*(xd_1 — xd_2)*	Double	**42**
D2/	*(xd_1 — xd_2)*	Double	**42**
D<	*(d_1 d_2 — flag)*	Double	**46**
D=	*(d_1 d_2 — flag)*	Double	**46**
D>F	*(d —); (F: — r)*	Floating	**109**
D>S	*(d — n)*	Double	**42**
DABS	*(d — +d)*	Double	**45**
DASM	*(addr —)*	common usage	**27**
DECIMAL	*(—)*	Core	**14**
DEFER <name>	*(—)*	common usage	**73**
DEFINITIONS	*(—)*	Search	**167**
DELETE-FILE	*(c-addr u — ior)*	File	**98**
DEPTH	*(— +n)*	Core	**34**
DF!	*(df-addr —); (F: r —)*	Floating Ext	**108**
DF@	*(df-addr —); (F: — r)*	Floating Ext	**108**
DFALIGN	*(—)*	Floating Ext	**115**
DFALIGNED	*(addr — df-addr)*	Floating Ext	**115**
DFLOAT+	*($df\text{-}addr_1$ — $df\text{-}addr_2$)*	Floating Ext	**115**
DFLOATS	*(n_1 — n_2)*	Floating Ext	**115**
DMAX	*(d_1 d_2 — d_3)*	Double	**45**
DMIN	*(d_1 d_2 — d_3)*	Double	**45**

Name	Stack	Wordset	Page
F.	*(F: r —)*	Floating Ext	**106**
F/	*(F: r_1 r_2 — r_3)*	Floating	**110**
F0<	*(— flag); (F: r —)*	Floating	**111**
F0=	*(— flag); (F: r —)*	Floating	**111**
F<	*(— flag); (F: r_1 r_2 —)*	Floating	**111**
F>D	*(— d); (F: r —)*	Floating	**109**
F@	*(f-addr —); (F: — r)*	Floating	**108**
FABS	*(F: r_1 — r_2)*	Floating Ext	**110**
FACOS	*(F: r_1 — r_2)*	Floating Ext	**112**
FACOSH	*(F: r_1 — r_2)*	Floating Ext	**112**
FALIGN	*(—)*	Floating	**115**
FALIGNED	*(addr — f-addr)*	Floating	**115**
FALOG	*(F: r_1 — r_2)*	Floating Ext	**112**
FALSE	*(— flag)*	Core Ext	**46**
FASIN	*(F: r_1 — r_2)*	Floating Ext	**112**
FASINH	*(F: r_1 — r_2)*	Floating Ext	**112**
FATAN	*(F: r_1 — r_2)*	Floating Ext	**112**
FATAN2	*(F: r_1 r_2 — r_3)*	Floating Ext	**112**
FATANH	*(F: r_1 — r_2)*	Floating Ext	**113**
FCONSTANT `<name>`	*(F: r —)*	Floating	**107**
FCOS	*(F: r_1 — r_2)*	Floating Ext	**113**
FCOSH	*(F: r_1 — r_2)*	Floating Ext	**113**
FDEPTH	*(— +n)*	Floating	**109**
FDROP	*(F: r —)*	Floating	**109**
FDUP	*(F: r — r r)*	Floating	**109**
FE.	*(F: r —)*	Floating Ext	**106**
FEXP	*(F: r_1 — r_2)*	Floating Ext	**113**
FEXPM1	*(F: r_1 — r_2)*	Floating Ext	**113**
FILE-POSITION	*(fileid — ud ior)*	File	**101**

Name	Stack	Wordset	Page
FILE-SIZE	*(fileid — ud ior)*	File	**101**
FILE-STATUS	*(c-addr u — x ior)*	File Ext	**101**
FILL	*(c-addr u b —)*	Core	**51**
FIND	*(c-addr — c-addr 0 | xt 1 | xt -1)*	Core, Search	**124**
FLITERAL	*(F: r —)*	Floating	**107**
FLN	*(F: r_1 — r_2)*	Floating Ext	**113**
FLNP1	*(F: r_1 — r_2)*	Floating Ext	**113**
FLOAT+	*(f-addr$_1$ — f-addr$_2$)*	Floating	**115**
FLOATS	*(n_1 — n_2)*	Floating	**115**
FLOG	*(F: r_1 — r_2)*	Floating Ext	**113**
FLOOR	*(F: r_1 — r_2)*	Floating Ext	**110**
FLUSH	*(—)*	Block	**91**
FLUSH-FILE	*(fileid — ior)*	File Ext	**98**
FM/MOD	*(d n_1 — n_2 n_3)*	Core	**42**
FMAX	*(F: r_1 r_2 — r_3)*	Floating	**110**
FMIN	*(F: r_1 r_2 — r_3)*	Floating	**110**
FNEGATE	*(F: r_1 — r_2)*	Floating	**110**
FORTH	*(—)*	Search Ext	**168**
FOVER	*(F: r_1 r_2 — r_1 r_2 r_1)*	Floating	**109**
FREE	*(a-addr — ior)*	Memory	**103**
FROT	*(F: r_1 r_2 r_3 — r_2 r_3 r_1)*	Floating	**109**
FROUND	*(F: r_1 — r_2)*	Floating	**111**
FS.	*(F: r —)*	Floating Ext	**106**
FSIN	*(F: r_1 — r_2)*	Floating Ext	**113**
FSINCOS	*(F: r_1 — r_2 r_3)*	Floating Ext	**113**
FSINH	*(F: r_1 — r_2)*	Floating Ext	**113**
FSQRT	*(F: r_1 — r_2)*	Floating Ext	**111**
FSWAP	*(F: r_1 r_2 — r_2 r_1)*	Floating	**109**

Name	Stack	Wordset	Page
M*/	(d_1 n_1 +n_2 — d_2)	Double	**43**
M+	(d_1 n — d_2)	Double	**43**
M-	(d_1 n — d_2)	common usage	**43**
M/	(d n_1 — n_2)	common usage	**43**
MARKER <name>	(—)	Core Ext	**165**
MAX	(n_1 n_2 — n_3)	Core	**44**
MIN	(n_1 n_2 — n_3)	Core	**44**
MOD	(n_1 n_2 — n_3)	Core	**42**
MOVE	($addr_1$ $addr_2$ u —)	Core	**51**
MS	(u —)	Facility Ext	**102**
NEGATE	(n — -n)	Core	**44**
NEXT	(—)	common usage	**171**
NIP	(x_1 x_2 — x_2)	Core Ext	**34**
NOT	(x — *flag*)	common usage	**47**
NOT (assembler)	(x_1 — x_2)	common usage	**178**
OF	(x_1 x_2 — | x_1)	Core Ext	**68**
ONLY	(—)	Search Ext	**168**
OPEN-FILE	(*c-addr u fam — fileid ior*)	File	**98**
OR	(x_1 x_2 — x_3)	Core	**44**
ORDER	(—)	Search Ext	**168**
OVER	(x_1 x_2 — x_1 x_2 x_1)	Core	**34**
PAD	(— *addr*)	Core Ext	**48**
PAGE	(—)	Facility	**85**
PARSE	(*char — c-addr u*)	Core Ext	**122**
PICK	(+n — x)	Core Ext	**34**
POSTPONE <name>	(—)	Core	**161**
PRECISION	(— u)	Floating Ext	**107**
PREVIOUS	(—)	Search Ext	**168**
QUIT	(*i*x* —); (*R: j*x* —)	Core	**122**

Name	Stack	Wordset	Page
R/O	*(— fam)*	File	**101**
R/W	*(— fam)*	File	**101**
R>	*(— x) (R: x —)*	Core	**37**
R@	*(— x) (R: x — x)*	Core	**37**
READ-FILE	*(c-addr u_1 fileid — u_2 ior)*	File	**99**
READ-LINE	*(c-addr u_1 fileid — u_2 flag ior)*	File	**99**
RECURSE	*(—)*	Core	**140**
REFILL	*(— flag)*	Block Ext, Core Ext, File Ext	**100, 122**
RENAME-FILE	*(c-addr$_1$ u_1 c-addr$_2$ u_2 — ior)*	File Ext	**98**
REPEAT	*(—)*	Core	**62**
REPOSITION-FILE	*(ud fileid — ior)*	File	**101**
REPRESENT	*(c-addr u — n flag$_1$ flag$_2$); (F: r —)*	Floating	**117**
RESIZE	*(a-addr$_1$ u — a-addr$_2$ ior)*	Memory	**103**
RESIZE-FILE	*(ud fileid — ior)*	File	**98**
RESTORE-INPUT	*(xn … x$_1$ n — flag)*	Core Ext	**122**
ROLL	*(+n)*	Core Ext	**34**
ROT	*(x_1 x_2 x3 — x_2 x_3 x_1)*	Core	**34**
RSHIFT	*(x_1 u — x_2)*	Core	**42**
S" <string>"	*(— c-addr u)*	Core, File	**101, 129**
S>D	*(n — d)*	Core	**43**
SAVE-BUFFERS	*(—)*	Block	**91**
SAVE-INPUT	*(— xn … x$_1$ n)*	Core Ext	**123**
SCR	*(— a-addr)*	Block Ext	**96**
SEARCH	*(c-addr$_1$ u_1 c-addr$_2$ u_2 — c-addr$_3$ u_3 flag)*	String	**52**
SEE <name>	*(—)*	Tools	**26**
SET-PRECISION	*(u —)*	Floating Ext	**107**
SF!	*(sf-addr —); (F: r —)*	Floating Ext	**108**

Name	Stack	Wordset	Page
SF@	*(sf-addr —); (F: — r)*	Floating Ext	**108**
SFALIGN	*(—)*	Floating Ext	**115**
SFALIGNED	*(addr — sf-addr)*	Floating Ext	**115**
SFLOAT+	*(sf-addr$_1$ — sf-addr$_2$)*	Floating Ext	**116**
SFLOATS	*(n_1 — n_2)*	Floating Ext	**116**
SIGN	*(n —)*	Core	**57**
SLITERAL	*(— c-addr u)*	String	**158**
SM/REM	*(d n_1 — n_2 n_3)*	Core	**43**
SOURCE	*(— c-addr u)*	Core	**123**
SOURCE-ID	*(— n)*	Core Ext, File	**120**
SPACE	*(—)*	Core	**86**
SPACES	*(u —)*	Core	**86**
STATE	*(— a-addr)*	Core, Tools Ext	**153**
SWAP	*(x_1 x_2 — x_2 x_1)*	Core	**34**
T*	*(d n — t)*	common usage	**43**
T/	*(t +n — d)*	common usage	**43**
THEN	*(—)*	Core	**67**
THEN (assembler)	*(addr —)*	common usage	**178**
THROW	*(k*x n — k*x | i*x n)*	Exception	**76**
THRU	*(i*x u_1 u_2 — j*x)*	Block Ext	**93**
TIME&DATE	*(— u_1 u_2 u_3 u_4 u_5 u_6)*	Facility Ext	**102**
TO <name>	*(x —)*	Core Ext, Local	**138**
TRUE	*(— flag)*	Core Ext	**47**
TUCK	*(x_1 x_2 — x_2 x_1 x_2)*	Core Ext	**35**
TYPE	*(c-addr u —)*	Core	**85**
U.	*(u —)*	Core	**54**
U.R	*(u +n —)*	Core Ext	**54**
U<	*(u_1 u_2 — flag)*	Core	**47**
U>	*(u_1 u_2 — flag)*	Core Ext	**47**

GENERAL INDEX

See also *Appendix C: Index to Forth Words*.