SwiftForth®

Development System for Linux and macOS

Reference Manual

FORTH, Inc.

Software products and services since 1973
www.forth.com

Current revision: October, 2016

FORTH, Inc.
Los Angeles, California USA
www.forth.com

CONTENTS

List of Figures

List of Tables

WELCOME!

What is SwiftForth?

SwiftForth is FORTH, Inc.'s interactive development system for the Linux, macOS, and Windows environments. SwiftForth is based on the Forth programming language, which for over 30 years has been the language of choice for engineers developing software for challenging embedded and real-time control systems. SwiftForth uses the power and convenience of the Linux, macOS, and Windows operating systems to provide the most intimate, interactive relationship possible with your application, to speed the software development process, and to help ensure thoroughly tested, bug-free code. It also provides a number of libraries and other programming aids to speed your application development.

This manual describes the basic principles and features of the SwiftForth Interactive Development Environment (IDE), including features specific to the i386 (IA-32) processor family and to both the Linux and macOS operating systems.

Scope of this Manual

The purpose of this manual is to help you learn SwiftForth and use it effectively. It includes the basic principles of the compiler, multitasker, libraries, development tools, and recommended programming strategies.

This manual does not attempt to teach Forth. If you are learning Forth for the first time, install this system and then turn to *Forth Programmer's Handbook*, which also accompanies this system. Paper copies of this manual, as well as the tutorial textbook, *Forth Application Techniques,* may be purchased from FORTH, Inc. *Forth Programmer's Handbook* and *Forth Application Techniques* are also available from our partner Amazon.com.

Audience

This manual is intended for engineers developing software to run in a Linux or macOS environment. It assumes general knowledge of the host operating system, and some familiarity with the Forth programming language (which you can get by following the suggestions below).

How to Proceed

If you are not familiar with Forth, start by reading the first two sections of *Forth Programmer's Handbook*, or work through the first six chapters of Forth Application Techniques. Then experiment with this system by examining some of the sam-

ple programs described in Section 1.4.2 and by writing simple definitions and testing them. *Forth Programmer's Handbook* is included with SwiftForth as a PDF file. Paper copies may be purchased from our publising partner Amazon.com, as can the programming tutorial *Forth Application Techniques*. Links to order these books can be found on our website at http://www.forth.com/forth/forth-books.

Typographic Conventions

A **BOLDFACE** type is used to distinguish Forth words (including assembler mnemonics) from other words in the text of this document. This same type style is used to display code examples.

This manual is for both the Linux and macOS (formerly OS X) versions of SwiftForth, which share a common code base. When a path name is given that lists **<os>** as one of the components, this is either **osx** or **linux**, depending on the OS you are using.

Support

The support period included with the original purchase of a SwiftForth system or version upgrade is one year. During the support period, you are entitled to unlimited downloads of new releases as well as engineer-level technical support via email. Please submit support requests via our website: www.forth.com/forth-tech-support/product. The support period may be renewed in one-year increments. Please visit www.forth.com for details.

FORTH, Inc. maintains an online forum for SwiftForth and SwiftX users at www.forth.com/user-forums.

SECTION 1: GETTING STARTED

This section provides a general overview of SwiftForth for Linux and macOS, including installlation and configuration instructions.

1.1 Components of SwiftForth

SwiftForth consists of the following components:

- Executable image of the SwiftForth development system, including the Interactive Development Environment (IDE), OS and library interface functions, and programming aids including the disassembler and other tools.
- Forth source files for the entire system.
- A cross compiler for recompiling the SwiftForth kernel.
- On-line documentation, including PDF versions of all manuals.

The SwiftForth directory structure is shown in Figure 1. All of SwiftForth is contained within this directory structure, making it easy to navigate..

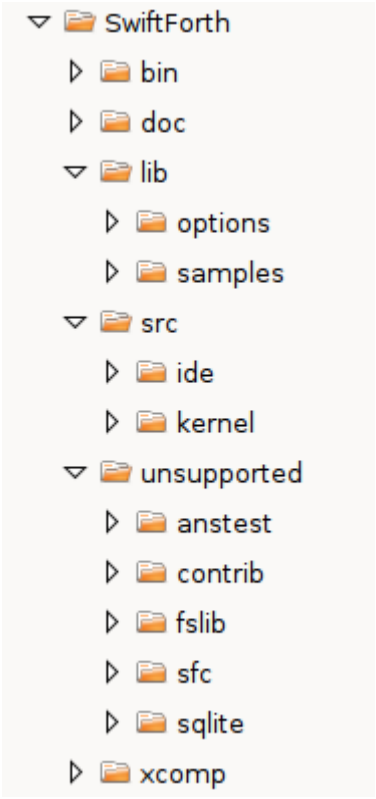

Figure 1. SwiftForth directory structure

1.2 SwiftForth System Requirements

In order to use SwiftForth, you need the following:

- PC running Linux[1]or and Intel-based Mac running OS X 10.4 or later. SwiftForth will occupy approximately 12 MB of disk space.
- A programmer's editor of your choice. Provision is made for linking interactive features of SwiftForth with most standard editors.

1.3 Installation Instructions

This section describes how to install and run your SwiftForth development system. It also describes basic procedures for compiling and testing programs, including the demos and sample code provided with your system.

If you have previously installed SwiftForth, you may wish to move older copies of files being installed to a backup directory or rename its directory (or the new one).

1.3.1 Installing SwiftForth

SwiftForth for Linux and MacOS is distributed as a tarball and installation is a manual procedure. But don't worry, it's easy.

In the text below, the tag <version> is replaced by the current SwiftForth version number (e.g. "3.2.0"). The tag <os> is one of "osx" or "linux" (without the quotes).

Download **SwiftForth-linux-macos-<version>.tgz** from the download link provided by FORTH, Inc. Extract the file into the installation directory with:

```
$ tar xzf .../SwiftForth-linux-macos-<version>.tgz
```

where "$" represents the shell command prompt and "..." is the directory in which you saved the SwiftForth tarball. This will the directory "SwiftForth" along with its subdirectories in your current working directory.

In order to run SwiftForth, you can either add **.../SwiftForth/bin/linux** (for Linux systems) or **.../SwiftForth/bin/osx** (for MacOS systems) to your **PATH**, or make symbolic links to the executable files "**sf**" (SwiftForth) and "**sfk**" (SwiftForth kernel) in a directory that's on your **PATH**.

Most Linux distributions include **$HOME/bin** in **$PATH** if it exists, so if this is the case in your system and you have unpacked SwiftForth in your home directory, you can simply do the following:

```
$ cd $HOME/bin
$ ln -s ../SwiftForth/bin/linux/sf
$ ln -s ../SwiftForth/bin/linux/sfk[2]
```

1.SwiftForth has been tested on many Linux distributions but is not guaranteed to work with all of them.

2.The SwiftForth kernel binary is not included in the evaluation version.

If your MacOS installation does not have **$HOME/bin**, you can create one:

```
cd ~
mkdir bin
```

Then add the symlinks as above:

```
$ cd $HOME/bin
$ ln -s ../SwiftForth/bin/osx/sf
$ ln -s ../SwiftForth/bin/osx/sfk
```

For both Linux and MacOS, make sure **$HOME/bin** is in your **PATH** (in **$HOME/.profile**). If you have to add it, log out and back in with a new terminal session and check it:

```
echo $PATH
```

If you installed SwiftForth in **/usr/local**, you may choose to perform the same procedure shown above, but with the symlinks in **/usr/local/bin** (which is normally in all users' paths).

Now you should be able to run SwiftForth:

```
$ sf
SwiftForth i386-<OS> <rel> <date>
```

Where <OS> is the operating system (Linux or MacOS), <rel> is the release version and <date> is the date the release was built.

Type **BYE** to exit SwiftForth.

After installation, PDF copies of the *SwiftForth Reference Manual*, *Forth Programmer's Handbook*, and the ANS Forth Standard can be found in the **SwiftForth/doc** directory.

1.3.2 Configuring Your Editor

SwiftForth invokes an external editor for **EDIT** and **G** by executing the shell script **SwiftForth/bin/editor**, passing the filename as the first argument and the line number as the second.

If the file **$HOME/.SwiftForth-editor** exists and is executable, then this is executed with the same arguments (by **SwiftForth/bin/editor**).

If **$HOME/.SwiftForth-editor** doesn't exist, then **SwiftForth/bin/editor** invokes the editor specified by the **VISUAL** environment variable, or **EDITOR** if **VISUAL** is not set.

SwiftForth/bin/editor assumes that the editor specified by **VISUAL** or **EDITOR** takes line numbers as a *+n* argument. If that isn't the case, you need to create your own **$HOME/.SwiftForth-editor** file.

Below is a sample **.SwiftForth-editor** file, which invokes gvim if running under X, and vim if not:

```
#!/bin/sh
if [ -n "$DISPLAY" ]; then
    exec gvim --remote-silent +$2 "$1"
else # no X
    exec vim +$2 "$1"
fi
```

1.4 Development Procedures

Here we provide a brief overview of some development paths you might pursue. You may wish to:

- Run sample programs installed with your system.
- Write and test application routines.
- Prepare a custom image of your SwiftForth system with added routines (for details, see Section 4.1.2).

Guidelines for doing these things are given in the following sections. Further details about SwiftForth's interactive development aids are given in Section 2.4.

1.4.1 Exploring SwiftForth

Run SwiftForth by typing **sf** at the shell command prompt. SwiftForth will attempt to execute everything you type on its command line.

When you type in the command window, SwiftForth will wait to process what you've typed until you press the Enter key. Before pressing Enter, you may backspace or use the left- and right-arrow keys to edit your command line. The up- and down-arrow keys page through previous command lines. The Tab key attempts to complete partial entries from its command-line history.

Forth commands are strings separated by spaces. The default behavior of SwiftForth is to be case-insensitive; that is, it treats upper-case and lower-case letters the same. For consistency, we will use upper case for most SwiftForth words. Linux and macOS libraries, file, and path names, however, are case sensitive. SwiftForth compiles all word names preserving their original case. You may temporarily set SwiftForth to be case sensitive by using the command **CASE-SENSITIVE**, and return to case insensitivity by using the command **CASE-INSENSITIVE**.

If you are new to the Forth programming language, we recommend you start your exploration by looking at some of the sample programs provided with the system. These are described in Section 1.4.2. As you look at the source for these applications in your editor, you may go to the SwiftForth command window to use **LOCATE** to find the source for words that are not part of the application file; the manuals provided with this system provide discussion of generic Forth words.

If you are an experienced Forth programmer, you will also benefit from looking at the sample programs to see how SwiftForth performs system and library calls. You may also wish to **LOCATE** various low-level words to familiarize yourself with this

implementation, which is discussed in Section 5.

References **LOCATE**, Section 2.4.1

1.4.2 Running the Sample Programs

SwiftForth is supplied with a number of sample programs. These may be found in the **Swiftforth/lib/samples** directory. All are provided as source files that you may **INCLUDE** and may be run according to their instructions.

References **INCLUDE** (loading source files), Section 3.1

1.4.3 Developing and Testing New Software

You may add new definitions to SwiftForth in four ways:

- Type them directly from the keyboard in the command window.
- Copy them from a source file and paste them into the console window (you can also copy definitions from the console window and paste them into files).
- Interpret an entire source file by typing **INCLUDE** `<filename>`. See Section 3.1 for details.
- Load them from Forth blocks, if the block-handling options are loaded.

The first two are extremely convenient for exploring a problem and for testing new ideas. Later stages of development generally involve editing a file and repeatedly loading it using **INCLUDE**. However, to maximize your debugging efficiency, remember to keep your definitions short (typically a few lines) and always follow the practice of bottom-up testing of individual, low-level words before trying the higher-level functions that depend on them.

References Source in text files, Section 3.1
Source in Forth blocks, Section A.1
Keyboard history, Section 2.3.1

1.5 Licensing Issues

You may find a copy of the license agreement in **SwiftForth/doc/license.pdf**.

SwiftForth is an unusual product, in that it is a development environment that can also produce program images containing the development environment itself. In this regard, SwiftForth differs dramatically from development systems such as Visual Basic which only produce executables. Because of this, we want to be very clear as to what is and is not permitted under this license.

1.5.1 Use of the Run-time Kernel

The purpose of SwiftForth is to enable you to develop Linux and macOS programs. Your programs may incorporate the SwiftForth run-time kernel as a component of a turnkey application in which the compiler and assembler or other programming aids are not available to *any* user of the program. If you need distribution rights other than these, please contact FORTH, Inc.

1.5.2 Use of the SwiftForth Development System

This section describes how you may use the SwiftForth development environment. You may:

- use the SwiftForth development system on any single computer;
- use the SwiftForth development system on a network, provided that each person accessing the Software through the network must have a copy licensed to that person;
- use the SwiftForth development system on a second computer as long as only one copy is used at a time;
- copy SwiftForth for archival purposes, provided that any copy must contain all of the original Software's proprietary notices; or
- distribute the run-time kernel provided with this system in accordance with the terms described in Section 1.5.1 above.

If you have purchased licenses for multiple copies of SwiftForth, all copies must contain all of the original Software's proprietary notices. The number of copies is the total number of copies that may be made for all platforms.

You may not:

- permit other individuals to use the SwiftForth development system except under the terms listed in Section 1.5.1 above;
- permit concurrent use of the SwiftForth development system;
- modify, translate, reverse-engineer, or create derivative works based on the SwiftForth development system except as provided in Section 1.5.1 above;
- copy the Software other than as specified above;
- rent, lease, grant a security interest in, or otherwise transfer rights to the Software without first obtaining written permission from FORTH, Inc.; or
- remove any proprietary notices or labels on the Software.

SECTION 2: USING SWIFTFORTH

SwiftForth supports interactive development and testing of Linux and macOS programs that can deliver very fast performance, and full access to standard system functions, shared objects, dynamic libraries, and other features found in these environments.

This introductory section gives a general view of the design of the SwiftForth interactive development environment. We recommend that you read this, even if you are already familiar with the Forth language.

If you are a Forth beginner, read *Forth Programmer's Handbook* carefully, and consider ordering *Forth Application Techniques*, a tutorial workbook offered by FORTH, Inc. The online version of the classic tutorial *Starting Forth* is also available on our web site. Review some of the demo applications supplied with your system in the directory **SwiftForth/lib/samples**. Find out what software is available by looking through the source files supplied with SwiftForth. Finally, write to the SwiftForth email group or to support@forth.com with any questions or problems (see "Support" on page 10). Forth programming courses are also available from FORTH, Inc., and can help shorten the learning process.

2.1 SwiftForth Programming

SwiftForth is a powerful and flexible system, supporting software development for a wide variety of programs. Although the internal principles of SwiftForth are simple, a necessary side-effect of its power is that it has a large number of commands and capabilities. To get the most benefit, allocate some time to become familiar with this system before you begin your project. This will pay off in your ability to get results quickly.

2.2 System Organization

SwiftForth, like most Forth development systems, is a single, integrated package that includes all the tools needed to develop applications. SwiftForth adds to the normal Forth toolkit special extensions for Linux and macOS programming. The complete system includes:

- Forth language compiler
- i386-family assembler
- Dynamically loaded library interface
- Libraries
- Interactive development aids

SwiftForth complies with ANS Forth, including the Core wordset, most Core Extensions, and most optional wordsets. Details of these features are given in Section 4.8

and Appendix B.

SwiftForth has two main components: a pre-compiled kernel and a set of options you may configure to suit your needs. In addition to these, you may add your application functions. When your application is complete, you may use the turnkey utility, described in Section 4.1.2, to prepare an executable binary image that you may distribute as appropriate

Be sure to read and understand "Licensing Issues" on page 15.

A purchased SwiftForth includes source for all extensions and most kernel functions, as this can be valuable documentation, including a cross-compiler that can be used to modify the kernel.

2.3 IDE Quick Tour

The SwiftForth Interactive Development Environment (IDE) presents a traditional command line interface (CLI). This section summarizes its principal features.

2.3.1 The SwiftForth Command-Line Interface

Your main interface with SwiftForth is through its command-line interface, typically via a terminal emulator window (referred to here as the *console window*). Commands that you type in the console window are executed by SwiftForth.

SwiftForth keeps a history of the command-line input. The up-arrow and down-arrow keys allow you to retrieve these command lines from the history buffer. You may edit them by using the left-arrow and right-arrow keys and typing; the Insert key toggles between insert and overwrite mode (as does clicking on the mode area of the status bar, discussed above). Press Enter to execute the entire line, or press Esc to leave the line without executing it.

The command-line input processor provides smart command completion. For instance, if you had previously typed **`INCLUDE FOO`**, typing **`INC`** and pressing the Tab key will complete the phrase **`INCLUDE FOO`** for you. Successive presses of the Tab key step through entries in the command-line history buffer.

Table 1 summarizes the special keyboard functions available in the command window.

Table 1: Command window keyboard controls

Key	Action
Up arrow Down arrow	Retrieve commands you have typed.
Left arrow Right arrow	Move cursor on command line.
Insert	Toggle the insert/overwrite mode of typing.

Table 1: Command window keyboard controls *(continued)*

Key	Action
Enter	Execute the command line the cursor is on.
Delete	Delete next character
VERASE[a]	Delete previous character
VKILL[b]	Delete the entire line
Home (also Ctrl-A)	Move to beginning of line
End (also Ctrl-E)	Move to end of line
Ctrl-Home	Show history
Ctrl-Shift-Del	Delete to end of line
F3	Append lastto EOL

a.VERASE is the system-defined erase key (e.g. Backspace)
b.VKILL is the system-define kill key (e.g. Ctrl-U)

2.4 Interactive Programming Aids

This section describes the specific features of SwiftForth that aid development. These tools will typically be used from the keyboard in the command window.

2.4.1 Interacting With Program Source

The command:

```
LOCATE <name>
```

If *name* is defined in the current search order, this will display several lines from the source file from which *name* was compiled, with *name* highlighted. The amount of text actually displayed depends on the current command window size.

LOCATE will work for all code compiled from available source files; source is not available for:

- code typed directly into the SwiftForth command window
- source code copied from a file and pasted into the command window
- code from files not supplied with the version of SwiftForth you are using
- words in the SwiftForth metacompiler

LOCATE may also fail if the source file has been altered since the last time it was compiled, since each compiled definition contains a reference to the position in the file that produced it.

For example, to view the source for the word **DUMP**:

```
LOCATE DUMP
```

The **LOCATE** command opens the correct source file, and displays the source:

```
/usr/local/SwiftForth/src/ide/tools.f
   49: -? : DUMP ( addr u -- )
   50:    BASE @ >R HEX  /SCROLL
   51:    BEGIN ( a n)  2DUP 16 MIN DUMPLINE
   52:       16 /STRING  DUP 0 <= UNTIL 2DROP
   53:    R> BASE ! ;
   54:
```

After you have displayed source for a word in the command window, typing **N** (Next) or **B** (Back) will display adjacent portions of that source file.

Follow a **LOCATE** command with **EDIT** to launch your text editor positioned at the beginning of the first line displayed by **LOCATE**. You can also type **EDIT** <name> to launch your editor in the source positioned at the line on which *name* is defined.

If the compiler encounters an error and aborts, you may directly view the file and line at which the error occurred by typing **L** for a **LOCATE** display (as shown above) or **G** to launch your editor positioned on the error line.

Glossary

LOCATE <name> *(—)*

Display the source from which *name* was compiled, with the source path and definition line number, in the SwiftForth command window. *name* must be in the current scope. Equivalent to double-clicking on *name* and selecting the Locate option from the menu generated by a right-click. The number of lines displayed depends on the current size of the SwiftForth command window.

N *(—)*

Display the Next lines of source following a **LOCATE** display.

B *(—)*

Display the previous (Back) lines of source following a **LOCATE** display.

L *(—)*

Following a compiler error, display the line of source at which the error occurred, along with the source path and line number, in the SwiftForth command window.

G *(—)*

Following a compiler error, open your editor positioned at the line of source at which the error occurred..

EDIT <name> *(—)*

Launches or switches to a linked editor, passing it appropriate commands to open the file in which *name* is defined, positioned at its definition. If *name* cannot be found in the dictionary, **EDIT** will abort with an error message.

2.4.2 Listing Defined Words

The command **WORDS** displays a list of all defined words in the current search order (i.e., currently accessible in the dictionary). You may see words in a particular vocabulary by specifying a vocabulary before **WORDS**. For example:

```
FORTH WORDS
```

will show only those in the **FORTH** vocabulary. Alternatively, you may specify **ALL WORDS** to get all defined words in all current vocabularies. You may search for words with a particular character sequence in their names by following **WORDS** with the search string. For example, if you type:

```
ALL WORDS M*
```

you will get this response:

```
M*/ M* UM*  ok
```

These three words contain the sequence **M*** in their names.

References

Search orders, wordlists, and vocabularies, *Forth Programmer's Handbook*
Wordlists in SwiftForth, Section 5.5.2

2.4.3 Cross-references

SwiftForth provides tools to enable you to find all references to a word, and also to identify words that are never called in the currently compiled program.

To find all the places a word is used, you may type:

```
WHERE <name>
```

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

If the same name has been redefined, **WHERE** gives the references for each definition separately. The short form **WH** is a synonym for **WHERE**:

```
WH <name>
```

does the same thing. This command is not the same as a source search—it is based on the code you are running on right now. This means you will be spared any instances of *name* in files you aren't using. However, it's also different from a normal dictionary search: it searches *all* wordlists, regardless of whether they are in the current search order. This is to reveal any definitions or usages of the word that may be currently hidden and, therefore, the source of subtle bugs.

Here's an example of a display produced by **WH** (used on a word defined in the SwiftForth kernel):

```
WH NUMBER?
```

```
WORDLIST: FORTH
/usr/local/SwiftForth/src/kernel/number.f
 21  53| : NUMBER? ( c-addr u -- d 2 | n 1 | 0)
 21  71| BASE @ >R  OVER C@ TEMP-BASE /STRING  NUMBER?  R> BASE ! ;
```

The first line shows the wordlist in which the word was found. The next lines show where it was defined, and the lines following show references to that definition. The numbers to the left of the vertical bar are file and line numbers. You can locate or edit the source referred to by one of these file:line pairs like this:

```
LOCATE 21 53|
EDIT 21 53|
```

Make sure you have the vertical bar at the end of the line number so **LOCATE** and **EDIT** will know that you're talking about a file:line reference.

Conversely, you may be interested in identifying words that have *never* been used, perhaps to prune some of them from your program. To do this, type **UNCALLED**. As with the cross-reference display, you may **LOCATE** or **EDIT** file:line number pairs to display or edit the source. Note that the fact that a word appears in the **UNCALLED** list doesn't necessarily mean you want to get rid of it.

Glossary

WHERE <name>
WH <name> *(—)*

Display a cross-reference showing the definition(s) of *name* and each line of source in which *name* is used in the currently compiled program. **WH** and **WHERE** are synonyms.

UNCALLED *(—)*

List all words that have never been called in a colon definition.

References Wordlists, *Forth Programmer's Handbook*
Wordlists and vocabularies in SwiftForth, Section 5.5.2

2.4.4 Disassembler

The disassembler is used to reconstruct readable source code from compiled definitions. This is useful as a cross-check, whenever a new definition fails to work as expected, and also shows the results of SwiftForth's code optimizer.

The single command **SEE** *name* disassembles the code generated for *name*. Since SwiftForth compiles actual machine code, it is unable to reconstruct a high-level definition. You may use **LOCATE** to display the source.

For example, the definition of **TIMER** (see Section 4.4.2) is:

```
: TIMER ( ms -- )
   COUNTER SWAP - U. ;
```

It disassembles as follows:

```
SEE TIMER
45A353   454883 ( COUNTER ) CALL        E82BA5FFFF
45A358   0 [EBP] EBX SUB                2B5D00
45A35B   4 # EBP ADD                    83C504
45A35E   407BC3 ( U. ) JMP              E960D8FAFF ok
```

The leftmost column shows the memory location being disassembled; the rightmost column shows the actual instruction.

An alternative is to disassemble or decompile from a specific address:

```
<addr> DASM
```

This is useful for disassembling code from an arbitrary address. The address must be an *absolute* address such as may be returned by a **LABEL** definition, a data structure, or obtained from an exception (see Section 4.7).

Glossary

SEE <name> *(—)*

Disassemble *name*.

DASM *(addr —)*

Disassemble code starting at *addr*.

References Using **LOCATE** to display source, Section 2.4.1
CODE and **LABEL**, Section 6.2

2.4.5 Viewing Regions of Memory

SwiftForth provides two basic ways to view memory: by dumping a region of memory, or by setting up *watch points* in a window.

2.4.5.1 Static Memory Dumps

Regions of memory may be dumped by using the commands **DUMP**, **IDUMP, UDUMP, and HDUMP.** All take as arguments a memory address and length in bytes; they differ in how they display the memory:

- **DUMP** displays successive bytes in hex, with an ASCII representation at the end of each line.
- **IDUMP** displays successive single-cell signed integers in the current number base.
- **UDUMP** displays single-cell unsigned numbers in the current number base.
- **HDUMP** displays unsigned cells in hex.

Data objects defined with **VARIABLE**, **CREATE**, or defining words built using **CREATE** will return suitable memory addresses. If you get an address using **'** <name> and wish to see its parameter field or data area, you may convert the address returned

by **'** to a suitable address by using **>BODY**.

Example 1: Dumping a string

```
CREATE MY-STRING   CHAR A C,  CHAR B C,  CHAR C C,
MY-STRING 3 DUMP
```

displays:

```
  45F860 41 42 43                                              ABC
```

Example 2: Dumping a 2CONSTANT

```
5000 500 2CONSTANT FIVES
' FIVES >BODY 8 IDUMP
```

displays:

```
0045F87C:        500       5000  ok
```

Glossary

DUMP *(addr u —)*
Displays *u* bytes of hex characters, starting at *addr*, in the current section, which may be either code or data. An attempt at ASCII representation of the same space is shown on the right.

IDUMP *(addr u —)*
Displays *u* bytes, starting at *addr*, as 32-bit integers in the current base.

UDUMP *(addr u —)*
Displays *u* bytes, starting at *addr*, as 32-bit unsigned numbers in the current base.

HDUMP *(addr u —)*
Displays *u* bytes, starting at *addr*, as unsigned hex numbers.

References Number bases, Section 4.3
Dictionary entry format and parameter fields, Section 5.5
PAD, *Forth Programmer's Handbook*
Memory organization in SwiftForth, Section 5.2

2.4.6 Single-Step Debugger

SwiftForth's simple single-step debugger allows you to step through source compiled from a file. The sample program `Sstest.f` can be loaded with the phrase:

```
REQUIRES sstest
```

When the source has been compiled from the file `Sstest.f`, type the following to invoke the single-step interface:

```
4 DEBUG 5X
```

At each breakpoint between Forth words, the current data stack is displayed along with a prompt to select the next action:

Nest Execute the next word, nesting if it is a call.

Step Execute the next word, with no nesting.

Return Execute to the end of the current definition without stopping.

Finish Finish executing the DEBUG word without stopping.

To load the single-step debug support without the **5X** example above:

```
REQUIRES singlestep
```

SECTION 3: SOURCE MANAGEMENT

In early Forth systems, many of which used Forth as the only operating system, source and data were maintained in 1024-byte blocks on disk. In such an environment, a block number mapped directly to the physical head/track/sector location on disk, and provided a fast and reliable means of managing the disk. As more Forth systems were implemented on other host operating systems, blocks became a kind of *lingua franca*, or standardized source interface, presented to the programmer regardless of the host OS. On such systems, blocks were normally implemented within host OS files.

Today, native Forth systems are extremely rare, and text files are the most portable medium between systems. ANS Forth includes an optional wordset for managing text files, as well as a block-handling wordset. SwiftForth provides full support for both, although SwiftForth source resides in text files.

The primary vehicle for SwiftForth program source is text files, which may be edited using the editor of your choice as described in Section 1.3.2. This section describes tools for managing text files.

3.1 Interpreting Source Files

You may load source files using the INCLUDE command, either from the command line or from within a source file that uses INCLUDE to load other files.

```
INCLUDE <filename>
```

The **INCLUDE** command causes all of a file to be loaded, as in:

```
INCLUDE hexcalc
```

The file extension **.f** is assumed if no extension is given.

The standard OS environment rules for describing paths apply:

1. **Absolute path:** A name starting with a / indicates an absolute path to the file. This type of path is only appropriate in cases where the directory structure is unlikely to change.
2. **Relative path from current working directory:** The name does not contain a leading / or ../ designation, and the file is located in the current directory or a subdirectory below it.
3. **Relative path to parent directories:** The name begins with a series of ../ (two periods and a forward slash). Each series raises the location one level from the current working directory.
4. **Path from SwiftForth "root" directory:** SwiftForth can manage paths relative to the location of the directory that contains the SwiftForth tree. Such paths are indicated by the starting symbol %. The actual root is speficied by the **FORTHINC** environment variable or the default path **/usr/local**. Note that the % does not represent a variable or other text macro. It simply takes the place of the **FORTHINC** environment variable

(or **/usr/local).**

For example, the basic error display functions are loaded by:

```
INCLUDE %SwiftForth/src/ide/errmessages
```

If you have launched SwiftForth with the current working directory set to your project directory (the directory containing your project's source files or subdiretories), your default path is that directory, so you don't need to preface local files with any path information. Your "local" configuration file could be loaded like this:

```
INCLUDE config
```

The word **INCLUDING** returns the string address and length of the filename currently being interpreted by **INCLUDE**. This may be convenient for diagnostic or logging tools.

The **CD** (Change Directory) command works in the SwiftForth command window just as it does in a Linux or macOS command shell. The **CD** command followed by *no* string will change to your home directory. If there are spaces in your pathname, surround it with double quotes. **CD** followed by a single minus character (-) changes back to the directory that was current before the previous **CD** command. This only goes back one level, so repeated "**CD -**" sequences will have the effect of toggling between two directories. Use **PWD** to display the current path.

If you wish to change your directory path temporarily, you may take advantage of SwiftForth's "directory stack." The word **PUSHPATH** will push your current path information onto the stack, and **POPPATH** will pop the current top path to become the current one.

Files can load other files that load still others. It is the programmer's responsibility to develop manageable load sequences. We recommend that you cluster the **INCLUDE** commands for a logical section of a program into a single file, rather than scattering **INCLUDE** commands throughout your source files. Your program will be more manageable if you can look in a small number of files to find the **INCLUDE** sequences.

REQUIRES includes the first matching file it finds from among all the files in the following list of directories, in this order:

1. The current directory
2. **%SwiftForth/lib**
3. **%SwiftForth/lib/options**
4. **%SwiftForth/lib/options/<os>**
5. **%SwiftForth/lib/samples**
6. **%SwiftForth/lib/samples/<os>**

The syntax is simply:

REQUIRES <filename>

The default filename extension is .f.

The **OPTIONAL** command helps you to avoid loading the same file more than once. The usage is:

```
OPTIONAL <name> <description>
```

Name can be any text that doesn't contain a space character. *Description* must be on the same line and is treated as a comment.

OPTIONAL is valid only while including a file.

If *name* already exists in the **LOADED-OPTIONS** wordlist, the rest of the file will be ignored. If *name* does not exist in the **LOADED-OPTIONS** wordlist, a dictionary entry for *name* will be constructed and the rest of the line will be ignored.

Glossary

INCLUDE <filename>[<.ext>] *(—)*
Direct the text interpreter to process *filename*; the extension is required if it is not **.f**. Path information is optional; it will search only in the current path, unless you precede *filename* with path information. **INCLUDE** does not change the current working directory.

INCLUDING *(— c-addr u)*
Returns the string address and count of the filename currently being interpreted by **INCLUDE**.

OPTIONAL <name> <description> *(—)*
If *name* already exists in the **LOADED-OPTIONS** wordlist, the rest of the file will be ignored; otherwise, *name* will be added to the **LOADED-OPTIONS** wordlist and loading will continue. Valid only while including a file.

PUSHPATH *(—)*
Push the current directory path onto the directory stack.

POPPATH *(—)*
Pop the top path from the directory stack to become the new current path.

REQUIRES <filename>[<.ext>] *(—)*
INCLUDE the file *filename* from a pre-defined list of directories. The extension is required if it is not **.f**.

References File-based disk access, *Forth Programmer's Handbook*

3.2 Extended Comments

It is common in source files have blocks of comments extending over several lines. Forth provides for comments beginning with **(** (left parenthesis) and ending with **)** (right parenthesis) to extend over several lines. However, the most common use of multi-line comments is to describe a group of words about to follow, and such descriptions frequently need to include parentheses for stack comments or for actual parenthetical remarks.

To provide for this, SwiftForth defines brace characters **{ }** as functionally equivalent to parentheses except for taking a terminating brace. A multi-line comment can

begin with **{** and end with **}**, and can contain parenthetical remarks and stack comments. Note that the starting brace, like a left parenthesis, is a Forth word and therefore must be followed by a space. The closing brace is a delimiter, and does not need a space.

For extra visual highlighting of extended comments, SwiftForth uses a full line of dashes at the beginning and end of an extended comment:

```
{ ------------------------------------------------
Numeric conversion bases

Forth allows the user to deal with arbitrary numeric
conversion bases for input and output. These are
the most common.
------------------------------------------------ }
```

Glossary

{ *(—)*

Begin a comment that may extend over multiple lines, until a terminating right brace **}** is encountered.

**** *(—)*

During an **INCLUDE** operation, treat anything following this word as a comment; i.e., anything that follows **** in a source file will not be compiled.

3.3 File-related Debugging Aids

You can monitor the progress of an **INCLUDE** operation bFor example, you might place these commands in a file where there is a problem:

```
VERBOSE
< troublesome code >
SILENT
```

VERBOSE turns on the monitor, and **SILENT** turns off the monitor. While monitoring is active, any **INCLUDE** will also display the name of the file being processed.

These words are not immediate, which means they should be used outside definitions unless you specifically intend to define a word that incorporates this behavior.

The default mode for the system is **SILENT**.

Glossary

VERBOSE *(—)*

Enables the **INCLUDE** monitor, with a default behavior of “display the text of each line.”

SILENT *(—)*

Disables the **INCLUDE** monitor.

SECTION 4: PROGRAMMING IN SWIFTFORTH

SwiftForth is generally compatible with the basic principles of Forth programming described in *Forth Programmer's Handbook*. However, since that book is fairly general and documents a number of optional features, this section will discuss particular features in SwiftForth of interest to a Forth programmer.

4.1 Programming Procedures

The overall programming strategy in SwiftForth involves developing components of your application one at a time and testing each thoroughly using the tools described in Section 2.4, as well as the intrinsically interactive nature of Forth itself. This process is sometimes described as *incremental development*, and is known to be a good way to develop sound, reliable programs quickly.

This section describes tools and procedures for configuring SwiftForth to include the features you need, as well as for managing the incremental development process.

4.1.1 Dictionary Management

When you are working on a particular component of an application, it's convenient to be able to repeatedly load and test it. In order to avoid conflicts (and an ever-growing dictionary), it is desirable to be able to treat the portion of the program under test as an *overlay* that will automatically discard previous versions before bringing in a new one.

This section covers two techniques for managing such overlays. To replace the contents of your entire dictionary with a new overlay, we recommend use of the word **EMPTY**. To create additional levels of overlays within the task dictionary, such that when an overlay is loaded, it will replace its alternate overlay beginning at the appropriate level, we recommend use of dictionary markers **MARKER** or **REMEMBER**. This section also discusses the option of allowing an overlay to reset the boundary between system and private definitions.

4.1.1.1 Single-level Overlays

The command **EMPTY** empties the dictionary to a pre-defined point, sometimes called its *golden state*. Initially, this is the state of the dictionary following the start of SwiftForth. Any application definitions that you don't want discarded with the overlay should be loaded as system options (see ???).

EMPTY discards all definitions and releases all dictionary space. To maintain a one-level overlay structure, therefore, just place **EMPTY** at the beginning of the file (such as an **INCLUDE** file that manages the other files in your application) you are repeatedly loading during development.

If you have loaded a set of functions that you believe are sufficiently tested and stable to become a permanent part of your run-time environment, you may add them to the *golden dictionary* by using the word **GILD**. This resets the pointers used by **EMPTY** so that future uses of **EMPTY** will return to *this* golden state. To permanently save this reconfigured system as a turnkey image, use **PROGRAM** (see Section 4.1.2).

Glossary

EMPTY *(—)*

Reset the dictionary to a predefined *golden* state, discarding all definitions and releasing all allocated data space beyond that state. The initial golden state of the dictionary is that following the launch of SwiftForth; this may be modified using **GILD**.

GILD *(—)*

Records the current state of the dictionary as a golden state such that subsequent uses of **EMPTY** will restore the dictionary to this state.

4.1.1.2 Multi-level Overlays

As the kinds of objects that programs define and modify become increasingly complex, more care must be given to how such objects can be removed from the dictionary and replaced using program overlays. In earlier Forth systems, the simple words **EMPTY** and **FORGET** were used for this purpose; but in this system, it is not possible for these words to know about all the structures and interrelationships that have been created within the dictionary. Typically, problems are encountered if new structures are chained to some other structure, or if a structure is being used as an execution vector for a system definition. Thus, an extensible concept has been developed to satisfy these needs.

A *dictionary marker* is an extensible structure that contains all the information necessary to restore the system to the state it was in when the structure was added. The words **MARKER** and **REMEMBER** create such entries, giving each structure a name which, when executed, will restore the system state automatically. The difference between the words is simply that a **MARKER** word will remove itself from the dictionary when it is executed, while a **REMEMBER** word will preserve itself so that it can be used again. **MARKER** words are most useful for initialization code that will only be executed once, while **REMEMBER** words are most useful in creating program overlays.

Although these marker words have been written to handle most common application requirements, you may need to extend them to handle the unique needs of your application. For example, if one of your application overlays requires a modification to the behavior of **ACCEPT**, you will need to extend the dictionary markers so they can restore the original behavior of **ACCEPT** when that overlay is no longer needed. To do this, you must keep in mind both the compile-time (when the structure is created) and the run-time (when the structure is used) actions of dictionary markers.

At compile time, when **MARKER** or **REMEMBER** is executed, a linked list of routines is executed to create a structure in the dictionary containing the saved context information. The word **:REMEMBER** adds another routine to the end of the sequence. This

routine can use the compiling words **,** and **C,** to add context information. Then, when the words defined by **MARKER** or **REMEMBER** are executed, another linked list of routines is executed to restore the system pointers to their prior values (a process called *pruning*). The word **:PRUNE** adds another routine to the end of the sequence. It will be passed the address within the structure where the **:REMEMBER** word compiled its information, and it must return the address incremented past that same information.

Thus, **:REMEMBER** is paired with **:PRUNE**, and these words should never appear *except* in pairs. The only exception is when the information being restored by **:PRUNE** is constant and there is no need for **:REMEMBER** to add it to the structure.

The linked lists of **:REMEMBER** or **:PRUNE** words are executed strictly in the order in which they are defined, starting with the earliest (effectively at the **GILD** point).

Two additional words are useful when extending these markers. **?PRUNED** will return *true* if the address passed to it is no longer within the dictionary boundaries, and **?PRUNE** will return *true* if the definition in which it occurs is being discarded.

EMPTY and **GILD** use these dictionary markers. **GILD** adds a marker to the dictionary that **EMPTY** can execute to restore the system state; it is never possible to discard below the **GILD** point.

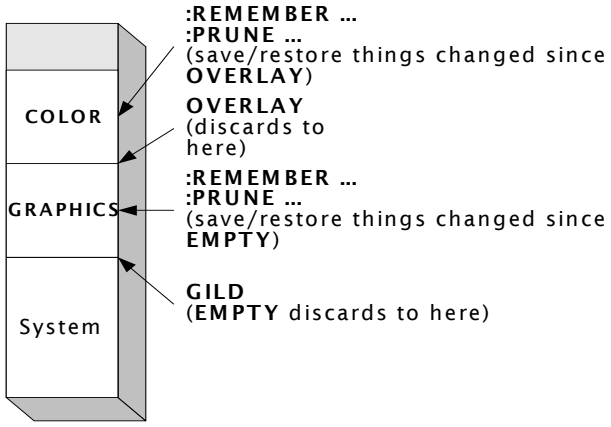

Figure 2. Multiple overlays

The following example (and Figure 2) illustrates how markers are used. Suppose your application includes an overlay called **GRAPHICS** whose load file begins with the command **EMPTY**. After **GRAPHICS** is loaded, you want to be able to load either 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 **REMEMBER** word as the *final definition* of the **GRAPHICS** overlay, using any name you want as a dictionary marker. For example:

   ```
   REMEMBER OVERLAY
   ```

   Place such a definition at the bottom of the **GRAPHICS** load file or block.

2. Place the appropriate reference to this marker definition on the *first line* of the load file or block of each level-two overlay. For instance,

```
( COLOR)   OVERLAY
```

Thus, when you execute the phrase:

```
INCLUDE COLOR
```

you "forget" any definitions which may have been compiled after **GRAPHICS**. The definition of **OVERLAY** is preserved to serve as a marker in the event you want to load an alternative, such as **B&W** (which would also have **OVERLAY** in its first line).

3. If the **COLOR** overlay changes a system pointer such as **'ACCEPT** to turn on text mode before accepting characters at the keyboard, then you must extend the memory pruning as follows:

```
:PRUNE   ?PRUNE IF  [ 'ACCEPT @ ] LITERAL
    'ACCEPT !  THEN ;
```

This will restore **'ACCEPT** to the value it had before the **COLOR** overlay changed it, if this extension to **PRUNE** is being removed from the dictionary (e.g., **?PRUNE** returns *true*).

4. If the **GRAPHICS** application is the one that changed **ACCEPT**, you must also extend the system's ability to remember what it was, both before and after **GRAPHICS** is loaded, as follows:

```
:PRUNE ( a - a')   ?PRUNE IF  [ 'ACCEPT @ ]
    LITERAL  ELSE  CELL SIZED @  THEN
     'ACCEPT ! ;

:REMEMBER   'ACCEPT @ , ;
```

This will remember what **'ACCEPT** was when a new marker word is created, and restore it to the original vector if the entire application is discarded. The extension must occur in the overlay level in which the necessity for it is created (i.e., in **GRAPHICS** itself).

By using different names for your marker definitions, you may create any number of overlay levels. However, no markers will work below the **GILD** point.

If you are working with multiple layers, you may encounter situations in which several layers have provided for the same data. In such cases, you need to take special care to avoid conflicting restorations. For example, a pointer named **'H** can be added like this:

```
:PRUNE ( a -- a' )
    ?PRUNE IF                     \ If pruning this extension
    [ 'H @ ] LITERAL DUP ?PRUNED IF\ If prior extension is
    DROP 'H @                     \  also pruned.
    THEN ELSE CELL SIZED @        \ Else get remembered
    THEN  'H ! ;                  \ Restore value

:REMEMBER   'H @ , ;
```

Note that this extension makes a distinction between a marker defined before this extension was defined and those that follow it.

If the marker was defined prior to this extension, the **:REMEMBER** data had not been

saved and **'H** needs to be restored to the value it had when this extension was compiled. It also accounts for some other extension touching the same location. If that other extension is also being removed, we assume it has already set the location to the proper value and we leave it alone this time.

:PRUNE and **:REMEMBER** are only necessary if the overlay changes a system pointer (or any value it did not itself instantiate).

If you are creating definitions or re-definitions at the keyboard, you can remove all of them either by typing **MARKER** <name> before you start and then executing *name* when you are done or, usually, just by typing **EMPTY**.

Glossary

:REMEMBER *(—)*

Add an un-named definition to the list of functions to be executed when **MARKER** or **REMEMBER** is invoked, to save system state data. The content of this definition must record the state information, normally using **,** or **C,**. When an associated **:PRUNE** definition is executed, it will be passed the address where the data was stored by **:REMEMBER**. Must be used with **:PRUNE** in the same overlay layer.

:PRUNE *($addr_1$ — $addr_2$)*

Add an un-named definition to the list of functions to be executed when **MARKER** or **REMEMBER** is invoked, to restore the system to a saved state. When the **:PRUNE** definition is executed, $addr_1$ provides the address where the data has been stored by a **:REMEMBER** in the same overlay layer.

?PRUNE *(— flag)*

Return *true* if the definition in which it is being invoked is being discarded (e.g., a **MARKER** is being executed).

?PRUNED *(addr — flag)*

Return *true* if *addr* is no longer within the active dictionary (e.g., it has been discarded via **MARKER** or **REMEMBER**).

REMEMBER <name> *(—)*

Create a dictionary entry for *name*, to be used as a deletion boundary. When *name* is executed, it will remove all subsequent definitions from the dictionary and execute all **:PRUNE** definitions (beginning with the earliest) to restore the system to the state it was in when *name* was defined. Note that *name* remains in the dictionary, so it can be used repeatedly.

4.1.2 Preparing a Turnkey Image

You may make a bootable binary image of SwiftForth, including additional code you have compiled, by typing:

```
PROGRAM <filename>
```

This records a “snapshot” of the current running system in the current path.

Simple use of **PROGRAM** is adequate to make a customized version of SwiftForth with added options of your choice, such as floating point support. To make a standalone application, however, requires storing the xt of the turnkey program's main entry point in the execution vector **'MAIN**.

For example, the following sequence sets the entry point to the word **GO** and saves the executable turnkey program **foo**:

```
' GO 'MAIN !
 PROGRAM foo
```

Glossary

PROGRAM <filename> *(—)*

Save an executable binary image of the current running system in the current path (or in the path specified with *filename*).

'MAIN *(— addr)*

Return the address of an execution vector containing the main program word to be launched at startup.

4.2 Compiler Control

This section describes how you can control the SwiftForth compiler using typed commands and (more commonly) program source. In most respects, the command-line user interface in the SwiftForth console is treated identically to program source. Anything you do in source may be done interactively in the command window.

SwiftForth may be programmed using either blocks or text files for source. Any source supplied with SwiftForth is provided in text files, the format that fits best with other programs and with the features of the programming environment; this section assumes you are working with text files.

References Using blocks for source, Section A.2

4.2.1 Case-Sensitivity

SwiftForth is normally case-insensitive, although you may set it to be case-sensitive. All standard Forth words are in upper case in SwiftForth, as are most SwiftForth words.

If you need to make SwiftForth case sensitive, you may do so with the command **CASE-SENSITIVE**; to return to case insensitivity, use **CASE-INSENSITIVE**. This may be useful for situations such as compiling legacy code from a case-sensitive system.

4.2.2 Detecting Name Conflicts

Because Forth is extremely modular, there are very many words and word names. As shipped, SwiftForth has over 3,500 named words. As a result, programmers are understandably concerned about inadvertently *burying* a name with a new one.

Insofar as possible, SwiftForth places words that are not intended for general use into separate vocabularies. This leaves about 1,600 words in the Forth vocabulary.

Re-defining a name is harmless, so long as you no longer need to refer to the buried word; it won't change references that were compiled earlier. Sometimes it is preferable to use a clear, meaningful name in a high level of your application—even if it buries a low-level function you no longer need—than to devise a more obscure or cumbersome name. And there are occasions when you specifically want to redefine a name, such as to add protection or additional features in ways that are transparent to calling programs.

However, it is often useful to have the compiler warn you of renamings. By default, SwiftForth will warn you if the name of a word being defined matches one in the same vocabulary. For example:

```
: DUP ( x -- x x )  DUP . DUP ;
DUP isn't unique.  ok
```

This makes it possible for you to look at the redefined word and make an informed judgement whether you really want to redefine it.

If you *are* redefining a number of words, you may find the messages annoying (after all, you already know about these instances). So SwiftForth has a flag called **WARNING** that you can set to control whether the compiler notifies you of redefinitions. You may set its state by using:

```
WARNING ON    or    WARNING OFF
```

You may also use **-?** just preceding a definition to suppress the warning that one time only, leaving **WARNING** enabled.

Glossary

WARNING *(— addr)*

Return the address of the flag that controls compiler redefinition warnings. If it is *true*, warnings will be issued.

ON *(addr —)*

Set the flag at *addr* to *true*.

OFF *(addr —)*

Set the flag at *addr* to *false*.

-? *(—)*

Suppress redefinition warning for the next definition only.

4.2.3 Conditional Compilation

[IF], **[ELSE]**, and **[THEN]** support conditional compilation by allowing the compiler to skip any text found in the unselected branch. These commands can be nested, although you should avoid very complex structures, as they impair the maintainability of the code.

Say, for example, you have defined a flag this way:

```
0 EQU MEM-MAP                   \ 1 Enables memory diagnostics
```

then in a load file you might find the statement:

```
MEM-MAP [IF] INCLUDE ../../memmap [THEN]
```

and later, this one:

```
MEM-MAP [IF] .ALLOCATED [THEN]        \ Display data sizes
```

Conditional compilation is also useful when providing a high-level definition that might be used if a code version of that word has not been defined. For example, in **Strings.f** we find:

```
[UNDEFINED] -ZEROS [IF]
: -ZEROS ( addr n -- addr n' )  \ Remove trailing 0s
    <high-level code> ;
[THEN]
```

[UNDEFINED] <word> will return *true* if *word* has not been defined. Thus, if a code or optimized version of **-ZEROS** was included in an earlier CPU-specific file (e.g., **Core.f**), it will not be replaced when this file is compiled later. Note that load order is extremely important!

In contrast, **[DEFINED]** <word> will return a *true* flag if *word* has been defined previously.

Conditional compilation is discussed more fully in *Forth Programmer's Handbook*.

4.3 Input-Number Conversions

When the SwiftForth text interpreter encounters numbers in the input stream, they are automatically converted to binary. If the system is in compile mode (i.e., between a **:** and **;**), it will compile a reference to the number as a literal; when the word being compiled is subsequently executed, that number will be pushed onto the stack. If the system is interpreting, the number will be pushed onto the host's stack directly.

All number conversions in Forth (input *and* output) are controlled by the user variable **BASE**. The system's **BASE** controls all input number conversions. Several words in this section may be used to control **BASE**. In each case, the requested base will remain in effect until explicitly changed. Punctuation in a number (decimal point, comma, colon, slash, or dash anywhere other than before the leftmost digit) will

cause the number to be converted as a double number.

Your current number base can be set by the commands **DECIMAL**, **HEX**, **OCTAL**, and **BINARY**. Each of these will stay in effect until you specify a different one. **DECIMAL** is the default.

In addition, input number conversion may be directed to convert an individual number using a particular base specified by a prefix character from Table 2. Following such a conversion, **BASE** remains unchanged from its prior value. If the number is to be negative, the minus sign must *follow* the prefix and *precede* the most-significant digit.

Table 2: Number-conversion prefixes

Prefix	Conversion base	Example
%	Binary	**%10101010**
&	Octal	**&177**
#	Decimal	**#-13579**
$	Hex	**$FE00**

SwiftForth also provides a more powerful set of words for handling number conversion. For example, you may need to accept numbers with:

- decimal places (dots or commas, depending on American or European conventions)
- angles or times with embedded colons
- dates with slashes
- part numbers, telephone numbers, or other numbers with embedded dashes

SwiftForth's number-conversion words are based on the low-level number conversion word from ANS Forth, **>NUMBER** (see "Number Conversions" in *Forth Programmer's Handbook*).

The word **NUMBER?** takes the address and length of a string, and attempts to convert it until either the length expires (in which case it is finished) or it encounters a character that is neither a digit (0 to **BASE**-1) nor valid punctuation.

NUMBER? interprets any number containing one or more valid embedded punctuation characters as a double-precision integer. Single-precision numbers are recognized by their *lack* of 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 task. A numeric string may be shorter than the length passed to **NUMBER?** if it is terminated with a blank. If another character is encountered (i.e., a character which is neither a digit according to the base nor punctuation), conversion will end. The leading minus sign, if present, must immediately precede the leftmost digit or punctuation character.

Any of the following punctuation characters may appear in a number (except in

floating-point numbers, as described in Section Section 10:):

```
,  .  +  -  /  :
```

All punctuation characters are functionally equivalent. A punctuation character causes the digits that follow to be counted. This count may be used later by certain of the conversion words. The punctuation character performs no other function than to set a flag that indicates its presence, and does not affect the resulting converted number. Multiple punctuation characters may be contained in a single number; the following two character strings would convert to the same number:

```
1234.56
1,23.456
```

NUMBER? will return one of three possible results:

- If number conversion failed (i.e., a character was encountered that was neither a digit nor a punctuation character), it returns the value zero.
- If the number is single precision (i.e., unpunctuated), it returns a 1 on top of the stack, with the converted value beneath.
- If the number is double-precision (i.e., contained at least one valid punctuation character), it returns a 2 on top of the stack, with the converted value beneath.

The floating-point option described in Section Section 10: extends the number conversion process to handle floating-point numbers.

The variable **DPL** is used during the number conversion process to track punctuation. **DPL** is initialized to a large negative value, and is incremented every time a digit is processed. Whenever a punctuation character is detected, it is set to zero. Thus, the value of **DPL** immediately following a number conversion contains potentially useful information:

- If it is negative, the number was unpunctuated and is single precision.
- Zero or a positive non-zero value indicates the presence of a double-precision number, and gives the number of digits to the right of the rightmost punctuation character.

This information may be used to scale a number with a variable number of decimal places. Since **DPL** doesn't care (or, indeed, know) what punctuation character was used, it works equally well with American decimal points and European commas to start the fractional part of a number.

The word **NUMBER** is the high-level input number-conversion routine used by SwiftForth. It performs number conversions explicitly from ASCII to binary, using the value in **BASE** to determine which radix should be used. This word is a superset of **NUMBER?**.

NUMBER will attempt to convert the string to binary and, if successful, will leave the result on the stack. Its rules for behavior in the conversion are similar to the rules for **NUMBER?** except that it always returns *just the value* (single or double). It is most useful in situations in which you know (because of information relating to the application) whether you will be expecting punctuated numbers. If the conversion fails due to illegal characters, a **THROW** will occur.

If **NUMBER**'s result is single precision (negative **DPL**), the high-order part of the working number (normally zero) is saved in the variable **NH**, and may be recovered to force the number to double precision.

Glossary

BASE *(— addr)*

Return the address of the user variable containing the current radix for number conversions.

DECIMAL *(—)*

Set **BASE** for decimal (base 10) number conversions on input and output. This is the default number base.

HEX *(—)*

Set **BASE** for hexadecimal (base 16) number conversions on input and output.

OCTAL *(—)*

Set **BASE** for octal (base 8) number conversions on input and output.

BINARY *(—)*

Set **BASE** for binary (base 2) number conversions on input and output.

>NUMBER *(ud_1 $addr_1$ u_1 — ud_2 $addr_2$ u_2)*

Convert the characters in the string at $addr_1$, 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. $addr_2$ 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.

NUMBER? *(addr u — 0 | n 1 | d 2)*

Attempt to convert the characters in the string at *addr*, whose length is *u*, into digits, using the radix in **BASE**, until the length *u* expires. If valid punctuation (**, . + - / :**) is found, returns *d* and 2; if there is no punctuation, returns *n* and 1; if conversion fails due to a character that is neither a digit nor punctuation, returns 0 (*false*).

NUMBER *(addr u — n | d)*

Attempt to convert the characters in the string at *addr*, whose length is *u*, into digits, using the radix in **BASE**, until the length *u* expires. If valid punctuation (**, . + - / :**) is found, returns *d*; if there is no punctuation, returns *n*; if conversion fails due to a character that is neither a digit nor punctuation, an **ABORT** will occur.

DPL *(— addr)*

Return the address of a variable containing the punctuation state of the number most recently converted by **NUMBER?** or **NUMBER**. If the value is negative, the number was unpunctuated. If it is non-negative, it represents the number of digits to the right of the rightmost punctuation character.

NH *(— addr)*

Return the address of a variable containing the high-order part of the number most

recently converted by **NUMBER?** or **NUMBER**.

References Numeric input, *Forth Programmer's Handbook*

4.4 Timing Functions

The words in this section support a time-of-day clock and calendar using the system clock/calendar functions.

4.4.1 Date and Time of Day Functions

SwiftForth supports a calendar using the <mm/dd/yyyy> format. Some of the words described below are intended primarily for internal use, whereas others provide convenient ways to enter and display date and time-of-day information.

SwiftForth's internal format for time information is an unsigned, double number representing seconds since midnight. There are 86,400 seconds in a day.

Dates are represented internally as a *modified Julian date* (MJD). This is a simple, compact representation that avoids the "Year 2000 problem," because you can easily do arithmetic on the integer value, while using the words described in this section for input and output in various formats.

The date is encoded as the number of days since 31 December 1899, which was a Sunday. The day of the week can be calculated from this with **7 MOD**.

The useful range of dates that can be converted by this algorithm is from 1 March 1900 thru 28 February 2100. Both of these are not leap years and are not handled by this algorithm which is good only for leap years which are divisible by 4 with no remainder.

A date presented in the form *mm/dd/yyyy* is converted to a double-precision integer on the stack by the standard input number conversion routines. A leading zero is not required on the month number, but is required on day numbers less than 10. Years must be entered with all four digits. A double-precision number entered in this form may be presented to the word **M/D/Y**, which will convert it to an MJD. For example:

```
8/03/1940 M/D/Y
```

will present the double-precision integer 8031940 to **M/D/Y**, which will convert it to the MJD for August 3, 1940. This takes advantage of the enhanced SwiftForth number conversion that automatically processes punctuated numbers (in this case, containing / characters) as double-precision (see Section 4.3).

Normally, **M/D/Y** is included in the application user interface command that accepts the date. For example:

```
: HIRED ( -- n )                    \ Gets date of hire
   CR ." Enter date of hire:"       \ User prompt
```

```
PAD 10 ACCEPT                   \ Await input to PAD
PAD SWAP NUMBER                 \ Text to number
M/D/Y                           \ Number to MJD
DATE-HIRED ! ;                  \ Store date
```

You can set the system date by typing:

```
<mm/dd/yyyy> NOW
```

To obtain the day of the week from an MJD, simply take the number modulo 7; a value of zero is Sunday. For example:

```
8/03/1940 M/D/Y 7 MOD .
```

gives 6 (Saturday).

An alternative form **D/M/Y** is also available. It takes the day, month, and year *as separate stack items*, and combines them to produce an MJD.

Output formatting is done by **(DATE)**, which takes an MJD as an unsigned number and returns the address and length of a string that represents this date. The word **.DATE** will take an MJD and display it in that format.

(DATE) is an execution vector. The following standard behaviors for this word are provided by SwiftForth:

- **(MM/DD/YYYY)** provides the SwiftForth default format, e.g., 12/30/2000.
- **(DD-MM-YYYY)** provides an alternative date format, e.g., 30-Dec-2000.

(MM/DD/YYYY) is the default. To change it, assign a new behavior to **(DATE)**:

```
<xt> IS (DATE)
```

...where *xt* is the execution token of the desired behavior.

Entry of times also takes advantage of SwiftForth's enhanced number conversion features. Just as you can use slashes in dates for readability, you can also use colons in times. For example, you could set your system's time-of-day clock by typing:

```
10:25:30 HOURS
```

Here, the double-precision integer 102530 is presented to **HOURS**, which converts it to internal units and stores it.

Glossary **Low-level time and date functions**

@NOW *(— ud u)*

Return the system time as an unsigned, double number *ud* representing seconds since midnight, and the system date as *u* days since 01/01/1900.

!NOW *(ud u —)*

Use parameters like those returned by **@NOW** to set the system time and date.

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

Return the system time and date as u_1 seconds (0–59), u_2 minutes (0–59), u_3 hours (0–23), u_4 day (1–31), u_5 month (1–12), *and* u_6 year (1900–2079)

Time functions

@TIME *(— ud)*

Return the system time as an unsigned, double number representing seconds since midnight.

(TIME) *(ud — addr u)*

Format the time *ud* as a string with the format `hh:mm:ss`, returning the address and length of the string.

.TIME *(ud —)*

Display the time *ud* in the format applied by **(TIME)** above.

TIME *(—)*

Display the current system time.

HOURS *(ud —)*

Set the current system time to the value represented by *ud*, which was entered as `hh:mm:ss`.

Date functions

D/M/Y *(u_1 u_2 u_3 — u_4)*

Convert day u_1, month u_2, and year u_3 into MJD u_4.

M/D/Y *(ud — u)*

Accept an unsigned, double-number date which was entered as `mm/dd/yyyy`, and convert it to MJD.

@DATE *(— u)*

Return the current system date as an MJD.

(DATE) *(u_1 — addr u_2)*

An execution vector that may be set to a suitable formatting behavior. The default **(MM/DD/YYYY)** formats the MJD u_1 as a string in the form `mm/dd/yyyy`, returning the address and length of the string.

(MM/DD/YYYY) *(u_1 — addr u_2)*

Format the MJD u_1 as a string with the format `mm/dd/yyyy`, returning the address and length of the string. This is the default behavior of **(DATE)**.

(DD-MM-YYYY) *(u_1 — addr u_2)*

Format the MJD u_1 as a string with the format `dd-mmm-yyyy` (where mmm is the text abbreviation for the month), returning the address and length of the string.

.DATE *(u —)*

Display the MJD *u* in the format applied by **(DATE)** above.

DATE *(—)*

Display the current system date.

NOW *(ud —)*

Set the current system date to the value represented by the unsigned, double number which was entered as `mm/dd/yyyy`.

References Number conversion, Section 4.3
Execution vectors, including **IS**, Section 4.5.5

4.4.2 Interval Timing

SwiftForth includes facilities to time events, both in the sense of specifying when something will be done, and of measuring how long something takes. These words are described in the glossary below.

The word **MS** causes a task to suspend its operations for a specified number of milliseconds, during which time other tasks can run. For example, if an application word **SAMPLE** records a sample, and you want it to record a specified number of samples, one every 100 ms., you could write a loop like this:

```
: SAMPLES ( n -- )
   ( n ) 0 DO                    \ Record n samples
      SAMPLE  100 MS             \ Take one sample, wait 100 ms
   LOOP ;
```

Because **MS** relies on a system timer, the accuracy of the measured interval depends on the underlying system. In general, the error on an interval will be on the order of the duration of a clock tick; however, it can be longer if a privileged system operation preempts the process running **MS**.

The words **COUNTER** and **TIMER** can be used together to measure the elapsed time between two events. For example, if you wanted to measure the overhead caused by other tasks in the system, you could do it this way:

```
: MEASURE ( -- )              \ Time the measurement overhead
   COUNTER                    \ Initial value
   100000 0 DO                \ Total time for 100,000 trials
      PAUSE                   \ One lap around the multitasker
   LOOP
   TIMER ;                    \ Display results.
```

Following a run of **MEASURE**, you can divide the time by 100,000 to get the time for an average **PAUSE**. For maximum accuracy, you can run an empty loop (without **PAUSE**) and measure the measurement overhead itself.

A formula you can use in Forth for computing the time of a single execution is:

```
<t> 100 <n> */ .
```

where *t* is the time given by a word such as **MEASURE**, above, and *n* is the number of iterations. This yields the number of 1/100ths of a millisecond per iteration (the

extra 100 is used to obtain greater precision).

The pair of words **uCOUNTER** and **uTIMER** are analogous to **COUNTER** and **TIMER**, but use the high-performance clock that runs at about 1 MHz, and manage 64-bit counts of microseconds.

Glossary

MS *(n —)*
PAUSE the current task for *n* milliseconds. The accuracy of this interval is always about one clock tick.

COUNTER *(— u)*
Return the current value of the millisecond timer.

TIMER *(u —)*
Repeat **COUNTER**, then subtract the two values and display the interval between the two in milliseconds.

EXPIRED *(u — flag)*
Return *true* if the current millisecond timer reading has passed *u*. For example, the following word will execute the hypothetical word **TEST** for *u* milliseconds:

```
: TRY ( u -- )                  \ Run TEST repeatedly for u ms.
   COUNTER +  BEGIN             \ Add interval to curr. value.
      TEST                      \ Perform test.
   DUP EXPIRED UNTIL ;          \ Stop when time expires.
```

uCOUNTER *(— d)*
Return the current value of the microsecond timer.

uTIMER *(d —)*
Repeat **uCOUNTER**, then subtract the two values and display the interval between the two in microseconds.

References Time and timing, *Forth Programmer's Handbook*
Timer support, Section 5.7

4.5 Specialized Program and Data Structures

The typical interface to system and library functions has led to the inclusion of the specialized structures described in this section.

4.5.1 String Buffers

ANS Forth provides the word **PAD** to return the address of a buffer that may be used for temporary data storage. In SwiftForth, **PAD** starts at 512 bytes above **HERE**, and its actual size can extend for the balance of unused memory (typically several MB). You can determine the actual size by the phrase:

```
UNUSED 512 - .
```

This will change, as will the address of **PAD**, any time you add definitions, perform **ALLOT**, or otherwise change the size of the dictionary.

SwiftForth itself uses **PAD** only for various high-level functions, such as the **WORDS** browser, cross-reference, etc. It is highly recommended as a place to put strings for temporary processing, such as building a message or command string that will be used immediately, or to keep a search key during a search. It is *not* appropriate for long-term storage of data; for this purpose you should define a buffer using **BUFFER:** or one of the words from Section 4.5.2.

BUFFER: is a simple way to define a buffer or array whose size is specified in characters. For example:

```
100 BUFFER: MYSTRING
```

...defines a 100-character region whose address will be returned by **MYSTRING**. If you prefer to specify the length in cells, you may use:

```
50 CELLS BUFFER: MYARRAY
```

...which defines a buffer 50 cells (200 bytes) long whose address is returned by **MYARRAY**.

There is an important distinction between **PAD** and buffers defined by **BUFFER:**. **PAD** is in a task's "user area," which means that if you have multiple tasks, each may have its own **PAD**. In contrast, a **BUFFER:** (like all other Forth data objects) is static and global.

Glossary

PAD *(— addr)*

Returns the address of a region in the user area suitable for temporary storage of string data. The size of **PAD** is indefinite, comprising all unused memory in a task's dictionary space.

BUFFER: <name> *(n —)*

Defines a buffer *n* characters in size. Use of *name* will return the address of the start of the buffer.

References **WORDS** browser, Section 2.4.2
Cross-reference, Section 2.4.3

4.5.2 String Data Structures

SwiftForth includes the standard Forth words **S"** and **C"**, and in addition provides several string-defining words that are used similarly. Most system and library funcstion use a standard string format that is terminated by a binary zero, referred to as an ASCIIZ string. Unicode strings might also required in a few places. In SwiftForth, these are ASCII characters in the low byte of a 16-bit character, with the high-order

byte zero.

All of the string-defining words are intended to be used inside definitions or structures such as switches (Section 4.5.4). If you use one of them interpretively, it will return an address in a temporary buffer. This is useful for interactive debugging, but you should not attempt to record such an address, as there is no guarantee how long the string will remain there!

SwiftForth includes a special set of string words with **** in their names, listed in the glossary below, that provide for the inclusion of control characters, quote marks, and other special characters in the string.

Within the string, a backslash indicates that the character following it is to be translated or treated specially, following the conventions used in the C printf() function. The transformations listed in Table 3 are supported. Characters following the \ are case-sensitive.

Table 3: Character sequence transformations

Character sequence	Compiled byte(s), hex	Description
****	5C	backslash
\"	22	double quote
\q	22	double quote
\a	07	bell
\b	08	backspace
\e	1B	escape
\l	0A	line feed
\f	0C	form-feed
\n	0A	Platform-specific end-of-line
\r	0D	carriage return
\t	09	horizontal tab
\v	0B	vertical tab
\xcc	*cc*	General constant, expressed ashex *cc*
\z	00	null

The string-management extensions in SwiftForth are documented below.

Glossary

Z" <string> **"** *(— addr)*
Compile a zero-terminated string, returning its address.

Z\" <string> **"** *(— addr)*
Compile a zero-terminated string, returning its address. The string may contain a \ followed by one or more characters which will be converted into a control or other special character according to the transformations listed in Table 3.

,Z" <string> **"** *(—)*

Compile a zero-terminated string with no leading count.

,Z\" <string> **"** *(—)*

Compile a zero-terminated string. The string may contain a \ followed by one or more characters which will be converted into a control or other special character according to the transformations listed in Table 3. (Differs from **Z\"** in that it is used interpretively to compile a string, whereas **Z\"** belongs inside a definition).

,U" <string> **"** *(—)*

Compile a Unicode string. Analogous to **U"** but used interpretively, whereas **U"** belongs inside a definition.

,U\" <string> **"** *(—)*

Compile a Unicode string. The string may contain a \ followed by one or more characters which will be converted into a control or other special character according to the transformations listed in Table 3.

S\" <string> **"** *(— addr n)*

Compile a string, returning its address and length. The string may contain a \ followed by one or more characters which will be converted into a control or other special character according to the transformations listed in Table 3.

.\" (—)

Compile a counted string and print it at run-time. The string may contain a \ followed by one or more characters which will be converted into a control or other special character according to the transformations listed in Table 3.

C\" <string> **"** *(— addr)*

Compile a counted string, returning its address. The string may contain a \ followed by one or more characters which will be converted into a control or other special character according to the transformations listed in Table 3.

," <string> **"** *(—)*

Compile the following string in the dictionary starting at **HERE**, and allocate space for it.

,\" <string> **"** *(—)*

Similar to **,"** but the string may contain a \ followed by one or more characters which will be converted into a control or other special character according to the transformations listed in Table 3. For example, **,\" Nice\nCode!\n"**.

STRING, *(addr u —)*

Compile the string at *addr*, whose length is *u,* in the dictionary starting at **HERE**, and allocate space for it.

PLACE *($addr_1$ u $addr_2$ —)*

Put the string at $addr_1$, whose length is *u*, at $addr_2$, formatting it as a counted string (count in the first byte). Does not check to see if space is allocated for the final string, whose length is *n*+1.

APPEND *($addr_1$ u $addr_2$ —)*

Append the string at $addr_1$, whose length is *u*, to the counted string already existing at $addr_2$. Does not check to see if space is allocated for the final string.

ZPLACE *($addr_1$ u $addr_2$ —)*

Put the string at $addr_1$, whose length is *u*, at $addr_2$ as a zero-terminated string. Does not check to see if space is allocated for the final string.

ZAPPEND *($addr_1$ u $addr_2$ —)*

Append the string at $addr_1$, whose length is *u*, to the zero-terminated string already existing at $addr_2$. Does not check to see if space is allocated for the final string.

4.5.3 Linked Lists

SwiftForth provides for linked lists of items that are of different lengths or that may be separated by intervening objects or code. This is an important underlying implementation technology used by switches (Section 4.5.4) as well as other structures. A linked list is controlled by a **VARIABLE** that contains a relative pointer to the head of the chain. Each link contains a relative pointer to the next, and a zero link marks the end.

In SwiftForth, references to data objects return full, absolute addresses (as described in Section 5.1.4). To convert these to and from relative addresses (the only form that is portable in a relocatable exectuable object file), SwiftForth provides the words **@REL**, **!REL**, and **,REL**. Respectively, these fetch a relative address (converting it to absolute), store an address converted from absolute to relative, and compile a converted relative address. These words are used when constructing linked lists or when referring to the links in them.

Linked lists are built at compile time. The word **>LINK** inserts a new entry at the top of the chain, updating the controlling variable and compiling a relative pointer to the next link at **HERE**. Similarly, **<LINK** inserts a link at the bottom of the chain, replacing the 0 in the previous bottom entry with a pointer to this link, whose link contains zero. Here is a simple example:

```
VARIABLE MY-LIST

MY-LIST >LINK 123 ,
MY-LIST >LINK 456 ,
MY-LIST >LINK 789 ,

: TRAVERSE ( -- )\ Display all values in MY-LIST
   MY-LIST BEGIN
      @REL ?DUP WHILE          \ While there are more links...
      DUP CELL+ @ .            \ Display cell following link
   REPEAT ;
```

Glossary

@REL *($addr_1$ — $addr_2$)*

Fetch a relative address from $addr_1$ to the stack, converting it to the absolute

address *addr*$_2$.

!REL *(addr*$_1$ *addr*$_2$ *—)*

Store absolute address *addr*$_1$ in *addr*$_2$, after converting it to a relative address.

,REL *(addr —)*

Compile absolute address *addr* at **HERE**, after converting it to a relative address.

>LINK *(addr —)*

Add a link starting at **HERE** to the top of the linked list whose head is at *addr* (normally a variable). The head is set to point to the new link, which, in turn, is set to point to the previous top link.

<LINK *(addr —)*

Add a link starting at **HERE** to the bottom of the linked list whose head is at *addr*. The new link is given a value of zero (indicating the bottom of the list), and the previous bottom link is set to point to this one.

CALLS *(addr —)*

Run down a linked list starting at *addr*, executing the high-level code that follows each entry in the list.

References Memory model and address management, Section 5.1.4

4.5.4 Switches

Switches can be used to process system event messages, Forth **THROW** codes, and other encoded information for which specific responses are required.

Switches are defined using this syntax:

```
[SWITCH <name> <default-word>
   <val1> RUNS <wordname>
   <val2> RUN: <words> ;
   ...
SWITCH]
```

Two words assign the runtime action:

- **RUNS** followed by a previously defined word will set that switch entry to execute the word.
- **RUN:** starts compiling a nameless colon definition (terminated by **;**) that will be executed for that value. This form is appropriate when the response code is used only in this one case.

This will build a structure whose individual entries have the form:

```
|link|value|xt|
```

...where each link points to the next entry. This is necessary because the entries can be anywhere in memory, and each individual list is subject to extension by **[+SWITCH**. The last link in a list contains zero. The *xt* will point either to a specified

word (if **RUNS** is used) or to the code fragment following **RUN:**. Note that the values do not need to be in any particular order, although performance will be improved if the most common values appear early.

When the switch is invoked with a value on the stack, the execution behavior of the switch is to run down the list searching for a matching value. If a match is found, the routine identified by *xt* will execute. If no match is found, the default word will execute. The data value is not passed to the *xt*s, but is passed to the default word if no match is found.

For example:

```
[SWITCH TESTING DROP
   1 RUNS WORDS
   2 RUN: HERE 100 DUMP ;
   3 RUNS ABOUT
SWITCH]
```

You may add cases to a previously defined switch using a similar structure called **[+SWITCH**, whose syntax is:

```
[+SWITCH <name>
   <valn+1> RUNx <wordname>
   <valn+2> RUNx <words> ;
   ...
SWITCH]
```

...where *name* must refer to a previously defined switch. No new default may be given. The cases *valn+1*, etc., will be added to the list generated by the original **[SWITCH** definition for *name*, plus any previous **[+SWITCH** additions to it.

Glossary

[SWITCH <name> *(— switch-sys addr)*

Start the definition of a switch structure consisting of a linked list of single-precision numbers and associated behaviors. The switch definition will be terminated by **SWITCH]**, and can be extended by **[+SWITCH**. See the discussion above for syntax.

switch-sys and *addr* are used while building the structure; they are discarded by **SWITCH]**.

The behavior of *name* when invoked is to take one number on the stack, and search the list for a matching value. If a match is found, the corresponding behavior will be executed; if not, the switch's default behavior will be executed with the value on the stack.

[+SWITCH <name> *(— switch-sys addr)*

Open the switch structure *name* to include additional list entries. The default behavior remains unchanged. The additions, like the original entries, are terminated by **SWITCH]**.

switch-sys and *addr* are used while building the structure; they are discarded by **SWITCH]**.

SWITCH] *(switch-sys addr —)*

Terminate a switch structure (or the latest additions to it) by marking the end of its linked list.

switch-sys and *addr* are used while building the structure; they are discarded by **SWITCH]**.

RUNS <word> *(switch-sys addr n — switch-sys addr)*

Add an entry to a switch structure whose key value is *n* and whose associated behavior is the previously defined *word*. The parameters *switch-sys* and *addr* are used internally during construction of the switch.

RUN: <words> **;** *(switch-sys addr n — switch-sys addr)*

Add an entry to a switch structure whose key value is *n* and whose associated behavior is one or more previously defined *word*s, ending with **;**. The parameters *switch-sys* and *addr* are used internally during construction of the switch.

4.5.5 Execution Vectors

An execution vector is a location in which an execution token (or *xt*) may be stored for later execution. SwiftForth supports several common mechanisms for managing execution vectors.

The word **DEFER** makes a definition (called a *deferred word*) which, when invoked, will attempt to execute the execution token stored in its data space. If none has been set, it will abort. To store an *xt* in such a vector, use the form:

```
<xt> IS <vector-name>
```

Note that **DEFER** is described in *Forth Programmer's Handbook* as being set by **TO**; that is not a valid usage in SwiftForth.

You may also construct execution vectors as tables you can index into. Such a table is described in *Forth Programmer's Handbook*. SwiftForth provides a word for executing words from such a vector, called **@EXECUTE**. This word performs, in effect, **@** followed by **EXECUTE**. However, it also performs the valuable function of checking whether the cell contained a zero; if so, it automatically does nothing. So it is not only safe to initialize an array intended as an execution vector to zeros, it is a useful practice.

DEFER <name> *(—)*

Define *name* as an execution vector. When *name* is executed, the execution token stored in *name*'s data area will be retrieved and its behavior performed. An abort will occur if *name* is executed before it has been initialized.

IS <name> *(xt —)*

Store *xt* in *name*, where *name* is normally a word defined by **DEFER**.

@EXECUTE *(addr —)*

Execute the *xt* stored in *addr*. If the contents of *addr* is zero, do nothing.

4.5.6 Local Variables

SwiftForth provides a mechanism for local variables that is compatible with ANS Forth. Local variables are defined and used as follows:

```
: <name>   ( xn ... x2 x1 -- )
   LOCALS| name1 name2 ... namen |
      < content of definition > ;
```

When *name* executes, its local variables are initialized with values taken from the stack. Note that the order of the local names is the inverse of the order of the stack arguments as shown in the stack comment; in other words, the first local name (e.g., $name_1$) will contain the top stack item (e.g., x_1).

The behavior of a local variable, when invoked by name, is to return its value. You may store a value into a local using the form:

```
<value> TO <name>
```

or increment it by a value using the form:

```
<value> +TO <name>
```

Local variable names are instantiated only within the definition in which they occur. Following the locals declaration, locals are not accessible except by name. During compilation of a definition in which locals have been declared, they will be found *first* during a dictionary search. Local variable names may be up to 254 characters long, and follow the same case-sensitivity rules as the rest of the system. SwiftForth supports up to 16 local variables in a definition.

Local variables in SwiftForth are instantiated on the return stack. Therefore, although you may perform some operations in a definition before you declare locals, you must not place anything on the return stack (e.g., using **>R** or **DO**) before your locals declarations. Return stack usage after the declaration of locals is governed by the rules in Section 5.1.5.

Note that, since local variables are not available outside the definition in which they are instantiated, use of local variables precludes interpretive execution of phrases in which they appear. In other words, when you decide to use local variables you may simplify stack handling inside the definition, but at some cost in options for testing. For this reason, we recommend using them sparingly.

Glossary

LOCALS| <$name_1$> <$name_2$> ... <$name_n$> **|** *(x_n ... x_2 x_1 —)*
Create up to 16 local variables, giving each an initial value taken from the stack such that $name_1$ has the value x_1, etc. Must be used inside a colon definition.

TO <name> *(x —)*
Store *x* in *name*, where *name* must be a local variable or defined by **VALUE**.

+TO <name> *(n —)*
Add *n* to the contents of *name*, where *name* must be a local variable or defined by

VALUE.

&OF <name> *(— addr)*
Return address of *name*, where *name* must be a local variable or defined by **VALUE**.

4.6 Convenient Extensions

This section presents a collection of words that have been found generally useful in SwiftForth programming.

Glossary

++ *(addr —)*
Increment the value at *addr*.

@+ *(addr — addr+4 x)*
Fetch the value *x* from *addr*, and increment the address by one cell.

!+ *(addr x — addr+4)*
Write the value *x* to *addr*, and increment the address by one cell.

~!+ *(x addr — addr+4)*
Write the value *x* to *addr*, and increment the address by one cell (accepts the parameters in reverse order compared to **!+**).

3DUP *(x_1 x_2 x_3 — x_1 x_2 x_3 x_1 x_2 x_3)*
Place a copy of the top three stack items onto the stack.

3DROP *(x_1 x_2 x_3 —)*
Drop the top three items from the stack.

ZERO *(x — 0)*
Replace the top stack item with the value 0 (zero).

ENUM <name> *(n_1 — n_2)*
Define *name* as a constant with value n_1, then increment n_1 to return n_2. Useful for defining a sequential list of constants (e.g., **THROW** codes; see Section 4.7).

ENUM4 <name> *(n_1 — n_2)*
Define *name* as a constant with value n_1, then increment n_1 by four to return n_2. Useful for defining a sequential list of constants that reference cells or cell offsets.

NEXT-WORD *(— addr u)*
Get the next word in the input stream—extending the search across line breaks as necessary, until the end-of-file is reached—and return its address and length. Returns a string length of 0 at the end of the file.

GET-XY *(— nx ny)*
Return the current screen cursor position. The converse of the ANS Forth word **AT-XY**.

/ALLOT *(n —)*

Allocate *n* bytes of space in the dictionary and initialize it to zeros (nulls).

4.7 Exceptions and Error Handling

Program exceptions in SwiftForth are handled using the **CATCH**/**THROW** mechanism of Standard Forth. This provides a flexible way of managing exceptions at the appropriate level in your program. This approach to error handling is discussed in *Forth Programmer's Handbook*.

SwiftForth includes a scheme to provide optional warnings of potentially serious errors. These include redefinitions of Forth words and possible attempts to compile or store absolute addresses, which is often inappropriate; see Section 5.1.4 for a discussion of address management. You may configure SwiftForth to report only certain classes of warnings, or disable warnings altogether.

At the top level of SwiftForth, the text interpreter provides a **CATCH** around the interpretation of each line processed. You may also place a **CATCH** around any application word that may generate an exception whose management you wish to control. **CATCH** is often followed by a **CASE** statement to process possible **THROW** codes that may be returned; if you are only interested in one or two possible **THROW** codes at this level, an **IF** … **THEN** structure may be more appropriate.

As shipped, SwiftForth handles errors detected by the text interpreter's **CATCH** by displaying an error message in the console window and aborting interpretation. You may add application-specific error messages if you wish, using the word **>THROW**. This word associates a string with a **THROW** code such that SwiftForth's standard error handler will display that string if it **CATCH**es its error code. The code is returned to facilitate naming it as a constant. For example:

```
12345 S" You've been bad! "  >THROW  CONSTANT BADNESS
```

Usage would be **BADNESS THROW**. In combination with **ENUM** (an incrementing constant, described in Section 4.6), **>THROW** can be used to define a group of **THROW** codes in your application. SwiftForth has predefined many internally used **THROW** codes, using the naming convention **IOR_**<name>. You may see which are available using **WORDS** specifying words that start with **IOR_**:

```
WORDS IOR_
```

To avoid re-naming a particular constant, let SwiftForth continue to manage your throw code assignments. The **VALUE THROW#** records the next available assigned **THROW** code. So, you could define your codes this way:

```
THROW#
    S" Unexpected Error"          >THROW ENUM IOR_UNEXPECTED
    S" Selection unknown"         >THROW ENUM IOR_UNKNOWN
    S" Selection not available"   >THROW ENUM IOR_NOTAVAIL
    S" Value too high"            >THROW ENUM IOR_TOOHIGH
    S" Value too low"             >THROW ENUM IOR_TOOLOW
TO THROW#
```

This defines codes to issue the messages shown. You can use the names defined with **ENUM** to issue the errors; for example:

```
GET-SELECTION  0 10 WITHIN NOT IF IOR_UNKNOWN THROW THEN
```

SwiftForth's error-handling facility includes an interface to the operating system's *signal* handler. When a signal is asserted, SwiftForth will generate a **THROW**. If your application has a **CATCH** for such an exception, you can handle it like any other SwiftForth exception.

SwiftForth's interface to signal handling is structured such that each individual thread can have its own exception (**CATCH**) frame, operating independently of others and nested as desired.

For example, if you type **0 @** (an illegal attempt to read memory location zero) you'll get a signal #11 (**SIGSEGV**). The resulting error message would look like this:

```
Signal #11 at 08048F3F in @
```

Glossary

>THROW *(n addr u — n)*

Associate **THROW** code *n* with the string *addr u* such that SwiftForth's standard error handler will display the string if it **CATCH**es its error code. The code is returned to facilitate naming it as a constant.

4.8 Standard Forth Compatibility

In implementing SwiftForth, we have made every attempt to deliver a standard Forth system in compliance with ANSI X3.215-1994 and ISO/IEC 15145:1997 (hereafter referred to as *Standard Forth*). The package includes the Hayes Standard Forth compliance test suite, in **SwiftForth/unsupported/anstest**.f. To run it, type:

```
CD %SwiftForth/unsupported/anstest
INCLUDE anstest
```

Sample output is shown in **anstest.log**.

This section provides a summary of SwiftForth's compliance to Standard Forth. Further details are given in Appendix B. We welcome any comments or questions in this regard.

Table 4 summarizes SwiftForth support for the wordsets defined in ANS Forth.

Table 4: SwiftForth support for Standard Forth wordsets

Wordset	Support provided
CORE	All words provided.
CORE EXT	All words provided except **EXPECT** and **SPAN**, which are marked "obsolescent."
BLOCK	All words provided.

Table 4: SwiftForth support for Standard Forth wordsets *(continued)*

Wordset	Support provided
BLOCK EXT	All words provided as an option. Enhanced editor and block management and editing support is also provided; see Sections A.2 and A.1.
DOUBLE	All words provided.
DOUBLE EXT	All words provided.
EXCEPTION	All words provided.
EXCEPTION EXT	All words provided.
FACILITY	All words provided.
FACILITY EXT	All words provided.
FILE	All words provided.
FILE EXT	All words provided.
FLOATING	All words provided.
FLOATING EXT	All words provided.
LOCALS	All words provided.
MEMORY	All words provided.
TOOLS	All words provided.
TOOLS EXT	All words provided except **FORGET**, which is obsolescent.
SEARCH	All words provided.
SEARCH EXT	All words provided.
STRING	All words provided.
STRING EXT	All words provided.

SECTION 5: SWIFTFORTH IMPLEMENTATION

SwiftForth is designed to produce optimal performance in the 32-bit Linux, macOS, and Windows environments. This section describes the implementation of the Forth *virtual machine* in this context.

5.1 Implementation Overview

This section provides a summary of the important implementation features of SwiftForth. More detail on critical issues is provided in later sections.

5.1.1 Execution model

SwiftForth is a 32-bit, direct code and subroutine-threaded Forth implementation.

Subroutine threading is an implementation strategy in which references in a colon definition are compiled as subroutine calls, rather than as addresses that must be processed by an *address interpreter*.

Colon and code definitions do not have a *code field* distinct from the content of the definition itself; data structures typically have a code field consisting of a call to the code for that data type. At the end of a colon definition, the **EXIT** used in other Forth implementation strategies is replaced by a subroutine return.

A subroutine-threaded implementation lends itself to inline code expansion. SwiftForth takes advantage of this via a header flag indicating that a word is to be compiled inline or called. Many kernel-level primitives are designated for inline expansion by being defined with **ICODE** rather than by **CODE** (see Section 6.2).

The compiler will automatically inline a definition whose **INLINE** field is set.

References Forth implementation strategies, *Forth Programmer's Handbook*

5.1.2 Code Optimization

More extensive optimization is provided by a powerful rule-based optimizer that can optimize many common high-level phrases. This optimizer is normally on, but can be turned off for debugging or comparison purposes. Consider the definition of **DIGIT**, which converts a small binary number to a digit:

```
: DIGIT ( u -- char)  DUP 9 > IF  7 +  THEN  [CHAR] 0 + ;
```

With the optimizer turned off, you would get:

```
SEE DIGIT
4078BF   4 # EBP SUB                    83ED04
```

```
4078C2   EBX 0 [EBP] MOV              895D00
4078C5   4 # EBP SUB                  83ED04
4078C8   EBX 0 [EBP] MOV              895D00
4078CB   9 # EBX MOV                  BB09000000
4078D0   403263 ( > ) CALL            E88EB9FFFF
4078D5   EBX EBX OR                   09DB
4078D7   0 [EBP] EBX MOV              8B5D00
4078DA   4 [EBP] EBP LEA              8D6D04
4078DD   4078F4 JZ                    0F8411000000
4078E3   4 # EBP SUB                  83ED04
4078E6   EBX 0 [EBP] MOV              895D00
4078E9   7 # EBX MOV                  BB07000000
4078EE   0 [EBP] EBX ADD              035D00
4078F1   4 # EBP ADD                  83C504
4078F4   4 # EBP SUB                  83ED04
4078F7   EBX 0 [EBP] MOV              895D00
4078FA   30 # EBX MOV                 BB30000000
4078FF   0 [EBP] EBX ADD              035D00
407902   4 # EBP ADD                  83C504
407905   RET                         C3 ok
```

But with it turned on, you would get:

```
SEE DIGIT
45A2D3   9 # EBX CMP                  83FB09
45A2D6   45A2DF JLE                   0F8E03000000
45A2DC   7 # EBX ADD                  83C307
45A2DF   30 # EBX ADD                 83C330
45A2E2   RET                          C3 ok
```

Another example shows the compiler's ability to "fold" literals, and operations on literals, into shorter sequences. The definition…

```
: TEST   DUP 6 CELLS + CELL+ $FFFF AND @ ;
```

…optimizes nicely to:

```
SEE TEST
45A2F3   4 # EBP SUB                  83ED04
45A2F6   EBX 0 [EBP] MOV              895D00
45A2F9   18 # EBX ADD                 83C318
45A2FC   4 # EBX ADD                  83C304
45A2FF   FFFF # EBX AND               81E3FFFF0000
45A305   0 [EBX] EBX MOV              8B1B
45A307   RET                          C3 ok
```

To experiment with this further, follow this procedure:

1. Turn off the optimizer, by typing **-OPTIMIZER**
2. Type in a definition.
3. Decompile it, using **SEE** <name>.
4. Turn the optimizer back on with **+OPTIMIZER**
5. Re-enter your definition.

Tip: You can re-enter the previous definition by pressing your up-arrow key until

you see the desired line, then press Enter to re-enter it.

6. Decompile it using **SEE** (Tip: use that up-arrow again here!) and compare.

   At the end of each colon definition, the optimizer attempts to convert a **CALL** followed by a **RET** into a single **JMP** instruction. This process is known as *tail recursion* and is a common compiler technique. To prevent tail recursion on a definition (e.g. a word that performs implementation-specific manipulation of the return stack), follow the end of the definition with the directive **NO-TAIL-RECURSION**.

5.1.3 Register usage

Following are the register assignments:

- **EBX** is the top of stack
- **ESI** is the user area pointer
- **EDI** contains the base address of SwiftForth's memory
- **EBP** is the data stack pointer
- **ESP** is the return stack pointer

All other registers are available for use without saving and restoring.

Note that the processor stack pointer is now for the return stack; this is a consequence of the subroutine-threaded model. This model also does not require registers for the address interpreter (**I** and **W** in some implementations). With more registers free, this system uses **EBX** to cache the top stack item, as noted above, which further improves performance.

References Assembly language programming in SwiftForth, Section 6

5.1.4 Memory Model and Address Management

SwiftForth's dictionary occupies a single, contiguous, flat 32-bit address space. SwiftForth is position-independent, which means it can run wherever the OS loads it without having to keep track of where that is. This means that compiled address references are relative; however, when words that reference data objects (e.g., things constructed with **CREATE**, **VARIABLE**, etc.) are executed, they return an absolute address that can be passed to external library or system calls, if desired, without change. All references in this book to "addresses" as stack arguments refer to these full, absolute addresses.

By being position independent, SwiftForth simplifies and speeds up all external library and system calls. Forth *execution tokens* are relative to the start of the runtime memory space, but they are used only internally and, thus, do not need conversion.

Although it is efficient for data objects to return absolute addresses, these must never be compiled into definitions in code that may be used in a turnkey program.

Instead, you should always *execute the object name at run time to get its address.* Special rules apply for references to data space in assembler code; see Section 6.5.5.

5.1.5 Stack Implementation and Rules of Use

The Forth virtual machine has two stacks with 32-bit items, which in SwiftForth are located in the stack frame assigned by the OS. Stacks grow downward in address space. The return stack is the CPU's subroutine stack, and it functions analogously to the traditional Forth return stack (i.e., carries return addresses for nested calls). A program may use the return stack for temporary storage during the execution of a definition, subject to the following restrictions:

- A program shall 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 within a **DO** loop, a program shall 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 shall be removed before **I**, **J**, **LOOP**, **+LOOP**, or **LEAVE** is executed.
- In a definition in which local variables will be used, values may not be placed on the return stack *before* the local variables declaration.
- All values placed on the return stack within a definition shall be removed before the definition is terminated or before **EXIT** is executed.

References Return stack use by local variables, Section 4.5.6

5.1.6 Dictionary Features

The dictionary can accommodate word names up to 254 characters in length. There is no address alignment requirement in SwiftForth.

References Dictionary structure, Section 5.5.1
Wordlists in SwiftForth, Section 5.5.2

5.2 Memory Organization

SwiftForth's dictionary resides in a virtual memory space whose size is fixed when SwiftForth is loaded. The default size is 8 MB. This size can be changed using the word **SET-MEMSIZE**, described below, although the change will not take effect until SwiftForth is re-launched.

The SwiftForth dictionary begins at **ORIGIN**. The dictionary grows toward high memory while the stacks grow toward low memory.

The Standard Forth command **UNUSED** returns the current amount of free space remaining in the dictionary; the command **FYI** displays the origin, current value of

HERE (top of dictionary), and remaining free space. If you wish to change the size of your dictionary, you may do so using:

```
<n> SET-MEMSIZE
```

…which will set the requested memory size to the greater of *n* or the current dictionary size plus 32K, and record this size for use by **PROGRAM** (which will generate a saved image as described in Section 4.1.2). You must close SwiftForth and launch the new image for the new size to take effect.

stack space is allocated by the OS when SwiftForth is loaded. SwiftForth reserves the top portion of this stack space for its data stack, and the balance is left for the return stack.

SwiftForth is an inherently multitasked system. When booted, it has a single task, whose name is **OPERATOR**. **OPERATOR**'s user area is allocated above the kernel. You may define and manage additional tasks, as described in Section 7.

Glossary

MEMTOP *(— addr)*
Return the address of the top of SwiftForth's committed memory.

ORIGIN *(— addr)*
Return the address of the origin of SwiftForth's committed memory.

+ORIGIN *(xt — addr)*
Convert an execution token to an address by adding **ORIGIN**.

-ORIGIN *(addr — xt)*
Convert an absolute address to a relative address by subtracting **ORIGIN**.

UNUSED *(— n)*
Return the amount of available memory remaining in the dictionary. This value must not fall below 32K.

SET-MEMSIZE *(n —)*
Set the requested dictionary size to the greater of *n* or the current size plus 32K bytes. The size will be recorded such that a new program image generated by **PROGRAM** will boot with the new size.

FYI *(—)*
Display current memory statistics: origin address, next available address, total committed memory, and amount of free memory.

5.3 Control Structure Balance Checking

The SwiftForth colon definition compiler performs an optional control structure balance check at the end of each definition. If any control structure is left unbalanced, SwiftForth will abort with this error message:

```
Unbalanced control structure
```

This feature can be turned on and off with the words **+BALANCE** and **-BALANCE**. The default is on.

Here is an advanced use of control structures that passes control between two high-level definitions, but leaves the first definition out of balance until the **IF** and **BEGIN** are resolved in the second definition[1]:

```
-BALANCE
: DUMIN ( d1 d2 -- d3 ) 2OVER 2OVER DU< IF BEGIN 2DROP ;
: DUMAX ( d1 d2 -- d3 ) 2OVER 2OVER DU< UNTIL THEN 2SWAP 2DROP ;
+BALANCE
```

Glossary

-BALANCE (—)

Turn off control structure balance checking.

+BALANCE (—)

Turn on control structure balance checking.

5.4 Dynamic Memory Allocation

SwiftForth supports the ANS Forth dynamic memory allocation wordset, described in *Forth Programmer's Handbook*. **ALLOCATE** gets memory from the system's virtual memory heap (which is outside SwiftForth's data space) using the standard **malloc** library call.

Virtual memory acquired thtough **ALLOCATE** does not have to be explicitly released (using **FREE**), as it will be automatically released when the program terminates.

5.5 Dictionary Management

This section describes the layout and management of the SwiftForth dictionary.

5.5.1 Dictionary Structure

The SwiftForth dictionary includes code, headers, and data. There is no separation of code and data in this system.

1.Thanks to Bill Muench of Intellasys for this elegant example.

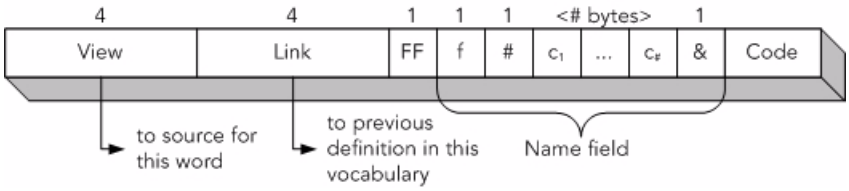

Figure 3. Dictionary header fields

Headers are laid out as shown in Figure 3. Within the header, the following fields are defined:

- **FF** is always set to FF_H; this is a *marker byte*, used to identify the start of the name field. No other byte in the name field of the header may be FF.
- **f** is the flags byte (depicted in Figure 4).
- # is a count byte, valid for 0–254 characters; it is followed by the given number of characters.
- **&** is the *inline byte*. If it is zero, references to this word must compile a **CALL** to this word; otherwise, it specifies the inline expansion size, 1–254 bytes.

  Expansion of an inline definition does not include a trailing **RET** if one is present.

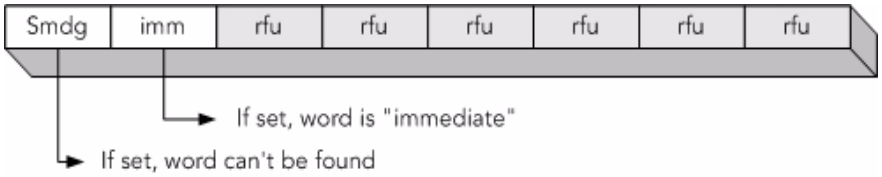

Figure 4. Structure of the "flags" byte

SwiftForth also provides a number of words for managing and navigating to various parts of a definition header. These are summarized in the glossary.

Glossary

IMMEDIATE *(—)*

Set the *immediate* bit in the most recently created header.

+SMUDGE *(—)*

Set the *smudge* bit in the flags byte, thus rendering the name invisible to the dictionary search. This bit is set for a colon definition while it is being constructed, to avoid inadvertent recursive references.

-SMUDGE *(—)*

Clear the smudge bit.

>BODY *(xt — addr)*

Return the parameter field address for the definition *xt*.

BODY> *(addr — xt)*
Return the *xt* corresponding to the parameter field address *addr*.

>CODE *(xt — addr)*
Return the code address *addr* corresponding to *xt*.

CODE> *(addr — xt)*
Return the *xt* corresponding to the code address *addr*.

>NAME *(xt — addr)*
Return the address of the name field for the definition *xt*.

NAME> *(addr — xt)*
Return the *xt* corresponding to the name at *addr*.

References
Wordlists in Forth, *Forth Programmer's Handbook*
Forth dictionaries, *Forth Programmer's Handbook*

5.5.2 Wordlists and Vocabularies

Definitions in the SwiftForth dictionary are organized in multiple linked lists. This serves several purposes:

- shortens dictionary searches
- allows names to be used in different contexts with different meanings (as often occurs in human languages)
- protects "internal" words from inappropriate or unintentional use or redefinition
- enables the programmer to control the order in which various categories of words are searched

Such lists are called *wordlists*. ANS Forth provides a number of system-level words for managing wordlists, discussed in *Forth Programmer's Handbook*. These facilities are fully implemented in SwiftForth.

In addition, SwiftForth defines a number of *vocabularies*. A vocabulary is a named wordlist. When the name of a vocabulary is invoked, its associated wordlist will be added to the beginning of the search order. A vocabulary is defined using the word **VOCABULARY** (also discussed in *Forth Programmer's Handbook*).

New definitions are linked into the current wordlist, which is normally a vocabulary. However, there may be times when you want to manage a special wordlist outside the normal system of Forth defining words and vocabularies.

The word **WORDLIST** creates a new, unnamed wordlist, and returns a unique single-cell numeric identifier for it called a *wid* (wordlist identifier). This may be given a name using **CONSTANT**.

The word **(WID-CREATE)** will create a definition from a string in a specified wordlist identifier, given its *wid*.

For example:

```
S" FOO" FORTH-WORDLIST (WID-CREATE)
```

In this example, the string parameters for **FOO** and the *wid* returned by **FORTH-WORDLIST** are passed to **(WID-CREATE)**.

To search a wordlist of this type, you may use **SEARCH-WORDLIST**. It takes string parameters for the string you're searching for and a *wid*. It will return a zero if the word is not found; if the word is found, it will return its *xt* and a 1 if the definition is immediate, and a -1 otherwise.

Wordlists are linked in multiple *strands*, selected by a hashing mechanism, to speed up the dictionary search process. Wordlists in SwiftForth may have any number of strands, and the user can set the system default for this. The default number of strands in a wordlist is 31. To change it, store the desired value in the variable **#STRANDS** and save a new executable as described in Section 4.1.2.

Glossary

WORDLIST *(— wid)*
Create a new empty wordlist, returning its wordlist identifier.

(WID-CREATE) *(addr u wid —)*
Create a definition for the counted string at *addr*, in the wordlist *wid*.

SEARCH-WORDLIST *(addr u wid — 0 | xt 1 | xt -1)*
Find the definition identified by the string *addr u* in the wordlist identified by *wid*. If the definition is not found, return zero. If the definition is found, return its execution token *xt* and 1 if the definition is immediate, -1 otherwise.

#STRANDS *(— addr)*
Variable containing the number of strands in a wordlist. Its default value is 31.

References Wordlists in Forth, *Forth Programmer's Handbook*

5.5.3 Packages

Encapsulation is the process of containing a set of entities such that the members are only visible thru a user-defined window. Object-oriented programming is one kind of encapsulation. Another is when a word or routine requires supporting words for its definition, but which are not inended for use by the "outside" world.

SwiftForth *packages* can be used to encapsulate groups of words. This is useful when you are writing special-purpose code that includes a number of support words plus a specific API. The words that constitute the API need to be *public* (globally available), whereas the support words are best kept *private* as they are only intended for internal use by the package itself. Packages in SwiftForth are implemented using wordlists.

The simplest way to show how packages work is with an example.

```
PACKAGE MYAPPLICATION
```

PACKAGE defines a named wordlist, and places a set of marker values (referred to as a *tag* in the glossary below) on the data stack. These marker values will be used by the words **PRIVATE** and **PUBLIC** to specify the scope of access for groups of words in the package, and must remain on the stack throughout compilation of the package.

Initially words defined in a package are **PRIVATE**. This means that the system variable **CURRENT**, which indicates the wordlist into which to place new definitions, is set to the newly created wordlist **MYAPPLICATION**.

Now we define a few private words. These words are the building blocks for the package, but are not intended to be used outside the package (indeed, they may be changed to suit modifications to the package without "breaking" any global dependencies). They are available for use while the package is being compiled, and are available at any time by explicit wordlist manipulation.

```
: WORD1 ( -- ) ...  ;
: WORD2 ( -- ) ...  ;
: WORD3 ( -- ) ...  ;
```

PUBLIC words are the words that are available to the user as the API, or to make the package accessible from other code. These words are placed into whatever wordlist was **CURRENT** when the package was opened. **PUBLIC** words may reference any words in the **PRIVATE** section, as well as any words normally available in the current search order.

```
PUBLIC
: WORD4 ( -- )   WORD1 WORD2 DUP + ;
: WORD5 ( -- )   WORD1 WORD3 OVER SWAP ;
```

We can switch back to **PRIVATE** words anytime.

```
PRIVATE
: WORD6 ( -- )   WORD1 WORD5 DUP + OVER ;
: WORD7 ( -- )   WORD1 WORD4 WORD6 SWAP ROT DROP ;
```

We can switch between **PUBLIC** and **PRIVATE** as many times as we wish. When we are finished, we close the package with the command:

```
END-PACKAGE
```

With the package closed, only the **PUBLIC** words are still "visible" as the worlist containing the **PRIVATE** words is no longer part of the search order.

If you need to add words to a previously-constructed package, you may re-open it by re-asserting its defining phrase:

```
PACKAGE MYAPPLICATION
```

An imporant feature of the word **PACKAGE** is that it will create a new package only if it doesn't already exist. Using **PACKAGE** with an already-created package name re-opens it. After re-opening, the normal **PUBLIC**, **PRIVATE**, and **END-PACKAGE** definitions apply.

Glossary

PACKAGE <name> *(— tag)*
If the package *name* has been previously defined, open it and return its *tag*. Otherwise, create it and return a *tag*.

PRIVATE *(tag — tag)*
Mark subsequent definitions invisible outside the package. This is the default condition following the use of **PACKAGE**.

PUBLIC *(tag — tag)*
Mark subsequent definitions available outside the package.

END-PACKAGE *(tag —)*
Close the package.

5.5.4 Automatic Resolution of References to Header Constants

Linux and macOS programming uses a large number of named "header" constants, such as **O_RDWR** (flags passed to the **open** syscall). The system would be burdened both in dictionary size and search speend if all of these were included as Forth definitions. The SwiftForth compiler uses an extension to load and access an external list of these header constants and the dictionary search mechanism is extended to include it if the parsed text is not a known word or a valid number. References to system constants act as numeric literals.

5.5.5 Dictionary Search Extensions

Because numerous entities can be referenced in SwiftForth beyond just defined words and numbers, the dictionary search mechanism has been extended. The full search includes the following steps (listed in the order performed):

1. Search the list of currently active local variables (if any).
2. Search the dictionary, according to the current search order.
3. Try to convert the word as a number (including floating point, if the floating-point option is loaded).
4. Check the system constants list (see Section 5.5.4)
5. Abort with an error message

References Local variables, Section 4.5.6
Search orders, Section 5.5.2
Number conversion, Section 4.3

5.6 Terminal-type Devices

Forth provides a standard API for terminal-type devices (those that handle character

I/O) that is described in the Terminal Input and Terminal Ouput topics in *Forth Programmer's Handbook*. Most implementations handle the existence of varied actual character-oriented devices by vectoring the standard words **KEY**, **EMIT**, **TYPE**, etc., to perform appropriate behaviors for each device supported, along with a mechanism for selecting devices.

5.6.1 Device Personalities

In SwiftForth, the collection of device-specific functions underlying the terminal API is called a *personality*. Personalities are useful for making specialized terminal-type devices; all the terminal I/O words in SwiftForth (e.g., **TYPE**, **EMIT**, **KEY**, etc.) are implemented as vectored functions whose actual behavior depends upon the current personality. SwiftForth's console window has a specific personality, for example, which is different from the one supported for buffered I/O.

A personality is a data structure composed of data and execution vectors; the first two cells contain the number of bytes of data and the number of vectors present. Any SwiftForth task may have its own current personality; a pointer to the current personality is kept in the user variable **'PERSONALITY**.

A description of a personality data structure is given in Table 5. When defining a personality for a device that supports only a subset of these, the unsupported ones must be given appropriate null behaviors. In most cases, this can be provided with **' NOOP** (address of a word that does nothing); or you could use **DROP** or **2DROP** to discard parameters.

Table 5: Terminal personality elements

Item	Stack	Description
Data section (values assumed to be single-cell integers)		
datasize		Size of the data section, in bytes.
maxvector		Number of vectors (whole cells).
handle		Handle for input/output.
previous		Address of saved previous personality.
Vector section (*xt*s of actual words to be executed by "Item" word)		
INVOKE	*(—)*	Open the personality (performing necessary initialization, if any).
REVOKE	*(—)*	Close the personality (performing necessary cleanup, if any).
/INPUT	*(—)*	Reset the input stream.
EMIT	*(char —)*	Output *char*.
TYPE	*(addr len —)*	Output the string *addr len*.
?TYPE	*(addr len —)*	Output the string *addr len*, respecting margins and performing necessary additional formatting.
CR	*(—)*	Go to the next line.
PAGE	*(—)*	Go to the next page (or clear screen).

Table 5: Terminal personality elements *(continued)*

Item	Stack	Description
ATTRIBUTE	*(n —)*	Set the attribute *n* for output strings from **TYPE** and **EMIT**.
KEY	*(— char)*	Low-level function to wait for a character.
KEY?	*(— flag)*	Low-level function to return *true* if a character is waiting.
EKEY	*(— char)*	Low-level function to wait for an extended character.
EKEY?	*(— flag)*	Low-level function to return *true* if an extended character is waiting.
AKEY	*(— char)*	Specialized version of **KEY** used by **ACCEPT**, which processes Enter, Backspace, etc., if necessary.
PUSHTEXT	*(addr len —)*	Push the string *addr len* into the input stream for interpretation.
AT-XY	*(nx ny—)*	Position the cursor at row *nx*, column *ny*.
GET-XY	*(— nx ny)*	Return the current cursor position.
GET-SIZE	*(— nx ny)*	Return the size, in characters, of the current display device.
ACCEPT	*(addr1 u1 -- u2)*	Input a line to buffer *addr1 u1*, returning the actual received length *u2*.

The handle and "previous" locations are used during execution; they are normally initialized to zero by using **0 ,** to allocate their space.

A personality does not have to implement a full set of vectors, but the correct order and structure must be maintained, with no gaps. In other words, you may leave off unused items at the end, if desired, making a shorter structure. Any vectors that are not implemented but are present must be assigned a default behavior and the stack effect must be correct.

For example, the following code defines a personality that does nothing:

```
: NULL 0 ;
: 2NULL 0 0 ;
: NOBUF 2DROP 0 ;

CREATE MUTE             \ data offset (bytes)
    4 CELLS ,           \ datasize0
    18 ,                \ maxvector4
    0 ,                 \ PHANDLE8
    0 ,                 \ PREVIOUS12

                        \ Vector
    ' NOOP ,            \ INVOKE( -- )
    ' NOOP ,            \ REVOKE( -- )
    ' NOOP ,            \ /INPUT( -- )
    ' DROP ,            \ EMIT  ( char -- )
    ' 2DROP ,           \ TYPE  ( addr len -- )
    ' 2DROP ,           \ ?TYPE ( addr len -- )
```

```
' NOOP ,             \ CR    ( -- )
' NOOP ,             \ PAGE  ( -- )
' DROP ,             \ ATTRIBUTE( n -- )
' NULL ,             \ KEY   ( -- char )
' NULL ,             \ KEY?  ( -- flag )
' NULL ,             \ EKEY  ( -- echar )
' NULL ,             \ EKEY? ( -- flag )
' NULL ,             \ AKEY  ( -- char )
' 2DROP ,            \ PUSHTEXT( addr len -- )
' 2DROP ,            \ AT-XY ( x y -- )
' 2NULL ,            \ GET-XY( -- x y )
' 2NULL ,            \ GET-SIZE( -- x y )
‘ NOBUF ,            \ ACCEPT( addr1 u1 -- u2)
```

An example is in **SwiftForth/src/ide/buffio.f**. This personality provides output to a temporary buffer between the words **[BUF** and **BUF]**. The phrase **@BUF** returns the address and length of the string in the buffer. Its main use is to create the output of a word in full before saving or printing it.

To activate a personality, pass the address of its data structure to **OPEN-PERSONALITY**; it will automatically save the present personality and perform the **INVOKE** behavior of the new one. Conversely, to close a personality you would use **CLOSE-PERSONALITY**, which closes the current one and re-asserts the saved one. For example, from **buffio.f**, we have:

```
: [BUF ( -- )   SIMPLE-BUFFERING OPEN-PERSONALITY ;
: BUF] ( -- )   CLOSE-PERSONALITY ;
```

…where **SIMPLE-BUFFERING** is the data structure for the buffered I/O personality.

Glossary

OPEN-PERSONALITY *(addr —)*

Makes the personality at *addr* the current one, saving the previous personality in its data structure.

CLOSE-PERSONALITY *(—)*

Close the current personality, restoring the previous saved one.

5.6.2 Keyboard Events

The words **EKEY** and **EKEY?** return *keyboard events*, beyond simple characters. Function key sequences passed in by the terminal emulator are converted to *extended key codes* which are defined as constants listed in Table 6 and are returned by **EKEY**.

Table 6: Extended key codes

Extended key code	Description
K-UP	Up arrow
K-DOWN	Down arrow
K-LEFT	Left arrow

Table 6: Extended key codes

Extended key code	Description
K-RIGHT	Right arrow
K-INSERT	Insert
K-DELETE	Delete
K-HOME	Home
K-END	End
K-PRIOR	Page up
K-NEXT	Page down
K-F1	F1
K-F2	F2
K-F3	F3
K-F4	F4
K-F5	F5
K-F6	F6
K-F7	F7
K-F8	F8
K-F9	F9
K-F10	F10
K-F11	F11
K-F12	F12
K-CTRL_UP	Ctrl key plus up arrow
K-CTRL_DOWN	Ctrl key plus down arrow
K-CTRL_LEFT	Ctrl key plus left arrow
K-CTRL_RIGHT	Ctrl key plus right arrow
K-CTRL-MASK	Mask for ctrl key

Note that some key presses (often referred to as “hot keys”) are completely processed by the host terminal emulator or desktop environment, and will never be passed to a program.

5.7 Timer Support

The OS supplies the following timers:

- **System clock**, which is returned by the **gettimeofday** system call. This clock is used by the SwiftForth words **COUNTER** and **TIMER**, and has a granularity as fine as one microsecond, depending on hardware and system configuration.
- **Sleep timer**, which controls the interval that a task is inactive when it executes the **nanosleep** system call. Its granularity could be as fine as one nanosecond, but this is also dependent on the hardware and system configuration. The SwiftForth **MS** function is implemented using a call to **nanosleep**.

It is important to remember, however, that timing is never completely accurate or predictable, because it is subject to system configuration differences, as well as to the demands of whatever other applications or hardware are operating at a given time.

References Interval timing, Section 4.4.2

SECTION 6: I386 ASSEMBLER

SwiftForth operates in 32-bit protected mode on the IA-32 (Intel Architecture, 32-bit) processors. This class of processors is referred to here as "i386", which encompasses Intel and AMD derivatives.

Throughout this book, we assume you understand the hardware and functional characteristics of the i386 as described in Intel IA-32 documentation (available for download at www.intel.com and from the *SwiftForth* page on www.forth.com). We also assume you are familiar with the basic principles of Forth.

This section supplements, but does not replace, Intel's manuals. Departures from the manufacturer's usage are noted here; nonetheless, you should use the Intel's manuals for a detailed description of the instruction set and addressing modes.

Where **boldface** type is used here, it distinguishes Forth words (such as register names) from Intel names. Usually, these are the same; for example, **MOV** can be used as a Forth word and as an Intel mnemonic. Where boldface is *not* used, the name refers to the manufacturer's usage or to hardware issues that are not particular to SwiftForth or Forth.

References
IA-32 (Intel Architecture, 32-bit), Wikipedia, wikipedia.org/wiki/IA-32
SwiftForth Programming References, www.forth.com

6.1 SwiftForth Assembler Principles

Assembly routines are used to implement the Forth kernel, to perform low-level CPU-specific operations, and to optimize time-critical functions.

SwiftForth provides an assembler for the i386. The mnemonics for the 386 opcodes have been defined as Forth words which, when executed, assemble the corresponding opcode at the next location in code space. The SwiftForth kernel is itself implmented with this assembler, so there are plenty of examples available in the kernel source.

Most mnemonics, addressing modes, and other mode specifiers use MASM names, but postfix notation and Forth's data stack are used to specify operands. Words that specify registers, addressing modes, memory references, literal values, etc. *precede* the mnemonic.

SwiftForth constructs conditional jumps by using a condition code specifier followed by **IF**, **UNTIL**, or **WHILE**, as described in Section 6.8. Table 10 summarizes the relationship between SwiftForth condition codes used with one of these words and the corresponding Intel mnemonic.

References
Assemblers in Forth, *Forth Programmer's Handbook*.

6.2 Code Definitions

Code definitions normally have the following syntax:

```
CODE <name>   <assembler instructions>  RET  END-CODE
```

or:

```
ICODE <name>   <assembler instructions>  RET  END-CODE
```

For example:

```
ICODE DEPTH ( -- +n )           \ Return current stack depth
    PUSH(EBX)                    \ save current tos
    16 [ESI] EBX MOV             \ get SP0 from user area
    EBP EBX SUB                  \ calculate stack size in bytes
    EBX SAR                     \ convert to cells
    EBX SAR
    RET END-CODE
```

All code definitions must be terminated by the command **END-CODE**.

The differences between words defined by **ICODE** and **CODE** are small, but crucial.

- **ICODE** begins a word which, when referenced inside a colon definition, will expand in-line code at that point in the definition. It may be called from another assembler routine, but it may not call any external function, nor may it exit any way except via the **RET** at the end.
- **CODE** begins a word which can only be called as an external function. When a **CODE** word is referenced inside a colon definition, a **CALL** to it will be assembled. A **CODE** word may call other words, and may have multiple exits.

You may name a code fragment or subroutine using the form:

```
LABEL <name>   <assembler instructions>  END-CODE
```

This creates a definition that returns the address of the next code space location, in effect naming it. You may use such a location as the destination of a branch or call, for example.

The critical distinction between **LABEL** and **CODE** is:

- If you reference a **CODE** definition inside a colon definition, SwiftForth will assemble a call to it; if you invoke it interpretively, it will be executed.
- Reference to a **LABEL** under any circumstance will return the *address* of the labelled code.

The defining words **LABEL**, **CODE**, and **ICODE** may not be used to define entry points to any part of a code routine except the beginning.

Within code definitions, the words defined in the following sections may be used to construct machine instructions.

CODE <name> *(—)*

Start a new assembler definition, *name*. If the definition is referenced inside a colon definition, it will be called; if the definition is referenced interpretively, it will be executed.

ICODE <name> *(—)*

Start a new assembler definition, *name*. If the definition is referenced inside a colon definition, its code will be expanded in-line; if the definition is referenced interpretively, it will be executed. A word defined by **ICODE** may not contain any external references (calls or branches).

LABEL <name> *(—)*

Start an assembler code fragment, *name*. If the definition is referenced, either inside a definition or interpretively, the address of its code will be returned on the stack.

END-CODE *(—)*

Terminate an assembler sequence started by **CODE** or **LABEL**.

6.3 Registers

The processor contains sixteen registers, of which eight 32-bit general registers can be used by an application programmer. Some have 8-bit and 16-bit, as well as 32-bit forms, as shown in Figure 5

.

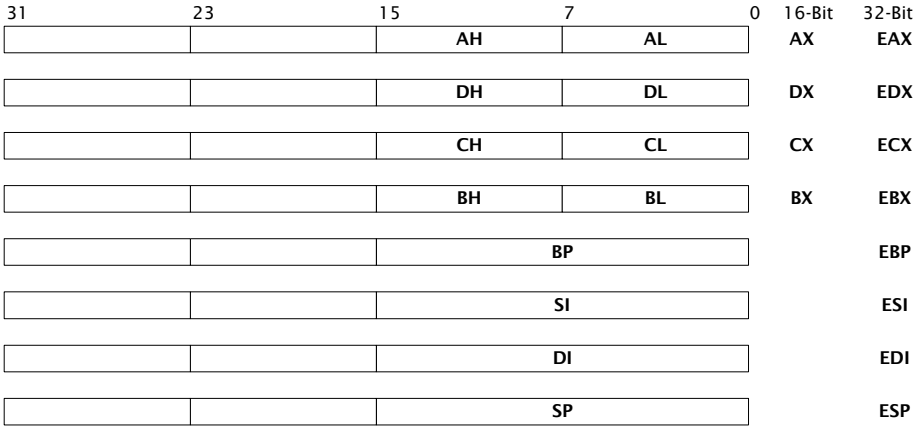

Figure 5. General registers

The general registers **EAX**, **EBX**, etc., hold operands for logical and arithmetic operations. They also can hold operands for address calculations (except the **ESP** register cannot be used as an index operand). As Figure 5 shows, the low 16 bits of the general registers can be referenced using these names.

Each byte of the 16-bit registers **AX**, **BX**, **CX**, and **DX** also has another name. The byte registers are named **AH**, **BH**, **CH**, and **DH** (high bytes) and **AL**, **BL**, **CL**, and **DL** (low bytes).

All of the general-purpose registers are available for address calculations and for the results of most arithmetic and logical operations; however, a few instructions assign specific registers to hold operands. For example, string instructions use the contents of the **ECX**, **ESI**, and **EDI** registers as operands. By assigning specific registers for these functions, the instruction set can be encoded more compactly. The instructions that use specific registers include: double-precision multiply and divide, I/O, strings, move, loop, variable shift and rotate, and stack operations.

SwiftForth has assigned certain registers special functions for the Forth virtual machine (see Section 5.1.3), as follows:

- **EBX** is the top of stack
- **ESI** is the user area pointer
- **EDI** contains the origin of SwiftForth's memory window
- **EBP** is the data stack pointer
- **ESP** is the return stack pointer

These registers may not be used for any other purpose unless you save and restore them.

References The Forth virtual machine, *Forth Programmer's Handbook*

6.4 Instruction Components

An instruction may have a number of components, which are listed below. The only required component is the opcode itself. In SwiftForth, as in many Forth assemblers, an opcode is represented by a Forth word which takes arguments specifying the other components, as desired, and assembles the completed instruction. For this reason, the opcode generally comes last, preceded by other notation (i.e., the syntax is `<operand(s)> <opcode>`). Except for this ordering, SwiftForth assembler notation follows Intel notation.

The components of an instruction may include:

1. **Prefixes:** one or more bytes that modify the operation of the instruction.
2. **Register specifier:** an instruction can specify one or two register operands. Register specifiers occur either in the same byte as the opcode or in the same byte as the addressing-mode specifier.
3. **Addressing-mode specifier:** when present, specifies whether an operand is a register or memory location; if in memory, specifies whether a displacement, a base register, an index register, and scaling are to be used.
4. **SIB (scale, index, base) byte:** when the addressing-mode specifier indicates the use of an index register to calculate the address of an operand, an SIB byte is included in the instruction to encode the base register, the index register, and a scaling factor.
5. **Displacement:** when the addressing-mode specifier indicates a displacement will be

used to compute the address of an operand, the displacement is encoded in the instruction. A displacement is a signed integer of 32, 16, or 8 bits. The 8-bit form is used in the common case when the displacement is sufficiently small. The processor extends an 8-bit displacement to 16 or 32 bits, taking into account the sign.

6. **Opcode:** specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different form of the operation.

These components make up the instruction operands discussed in the next section.

References Index registers and displacement address mode specifiers, Section

6.5 Instruction Operands

Everything except the opcode may be considered an operand. SwiftForth notation for various kinds of operands is similar to Intel's, except for order. In general, where there are two operands, the order is `<source> <destination>`.

6.5.1 Implicit Operands

An operand is *implicit* if the instruction itself specifies it. In effect, this means the operand is absent! Some examples include:

- **`CLD`** Clear direction flag to zero
- **`RET`** Return from subroutine

6.5.2 Register Operands

A register operand specifies one or two registers, by giving their names. Some examples include:

- **`ESI INC`** Increment **`ESI`**.
- **`EBX ECX MOV`** Move top-of-stack to **`ECX`**

6.5.3 Immediate Operands

An *immediate* operand specifies a number as part of the instruction. SwiftForth indicates immediate operands by following the number with a **`#`** character. Numbers may be specified in any base; if your base is decimal and you want to specify a single number in hex, you can use the **`$`** prefix (see Section 4.3). Some examples of immediate operands include:

- **`8 # EBP ADD`** Add 8 to the data stack pointer
- **`2 # EBX SUB`** Subtract 2 from the top stack item
- **`32 # AL XOR`** Exclusive-or the character in AL by 32

6.5.4 I/O Operands

I/O operands specify a port address as an argument to an **IN** or **OUT** instruction. A port address may be specified as an immediate value, provided it fits in eight bits (values to 256); otherwise you must put it in a register. Some examples:

- **4 # AL IN** Read a byte from the port at address 4 into register **AL**
- **DX EAX IN** Read from the port whose address is given by **DX** into **EAX**.

However, these instructions are not generally useful in Linux and macOS programming because direct port access is not normally allowed by user-mode programs.

6.5.5 Memory Reference Operands

This is a large class of operands, because of the many ways to address memory.

6.5.5.1 Direct Addressing

This is the simplest form of memory addressing, in which you simply specify an address and SwiftForth assembles a reference to it.

Addresses used as destinations for a **JMP** or **CALL** are assembled as offsets from the location of the instruction, within the memory space allocated to SwiftForth when it is loaded. To get an address of a defined word to be used as a destination for a **JMP** or **CALL** use the form:

```
' <name> >CODE
```

The word **>CODE** converts the execution token supplied by **'** (“tick”) to a suitable address.

Variables and other data structures present some particular problems as a consequence of the position-independent implementation strategy used in SwiftForth. The address returned by such words (to be precise, any word defined using **CREATE** that returns a data space address) is the absolute address in the machine. SwiftForth and executable programs generated from SwiftForth are always instantiated in the same virtual address space, but libraries may be instantiated in different places at different times. This means that code that might *ever* be used in a shared object must avoid compiling references to absolute addresses.

In order to obtain a generic address, special actions are required. SwiftForth provides an assembler macro **ADDR** that will place a suitable data address in a designated register, used in the form:

```
<addr> <reg> ADDR
```

where *addr* is the data address returned by a **VARIABLE** or any word defined using **CREATE**, and *reg* is the desired register. This macro is optimized for **EAX**, the general-purpose accumulator, but you may specify any other register. The following example shows how to add a value to the contents of a **VARIABLE**:

```
VARIABLE DATA

CODE +DATA   ( n -- )          \ Add n to the contents of DATA
   DATA EAX ADDR               \ Address of DATA to EAX
   EBX 0 [EAX] ADD             \ Add top-of-stack to DATA
   POP(EBX)                    \ POP stack
RET   END-CODE
```

Another example is:

```
' THROW >CODE JMP
```

Jump to **THROW** (on an error condition). When using this method to abort from code, you must provide the throw code value in **EBX** (top-of-stack).

Glossary

ADDR *(addr reg —)*

Convert the data address *addr* from an absolute address to an address relative to the start of SwiftForth's memory, and place it in the register *reg*.

References Addressing in SwiftForth, Section 5.1.4
>CODE, Section 5.5.1
Addressing with an Offset

Offset addressing combines parameters that are added to a base or index register, with other parameters, as shown in Figure 6.

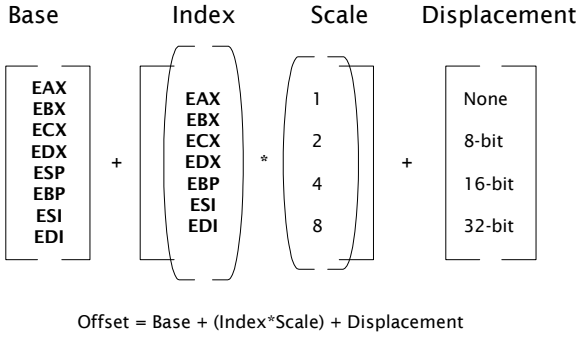

Figure 6. Offset (or effective address) computation

The SwiftForth assembler syntax is:

```
disp [base] [index*scale] opcode
```

The displacement must be given, even if it is zero. The assembler will use the correct instruction form, depending upon the size of the displacement (omitting it if it is zero).

You may specify a base register, an index register, or both. The default scale of an index register is 1; the form of reference is the same as for a base register, e.g.,

[EBX]. For other scale factors, the form of reference is given in Table 7.

Table 7: Forms for scaled indexing

Scale (from Figure 6)	Form	Example
2	[reg*2]	[EAX*2]
4	[reg*4]	[EAX*4]
8	[reg*8]	[EAX*8]

For example:

- **0 [EBP] EAX MOV**
  Move the second stack item to **EAX**
- **4 [EBP] EAX MOV**
  Move the third stack item to **EAX**
- **-1 [ECX] [EDI] EDI LEA**
  Load the effective address (**ECX**+**EDI**) -1 into **EDI**.
- **0 [EDI] [EDX*4] EAX CMP**
  Add the number of cells (4 bytes each) in **EDX** to the base address in **EDI**, and compare the referenced item to **EAX**.

6.5.5.2 Stack-based Addressing

The hardware stack, controlled by **ESP**, is used for subroutine calls, which makes it a natural choice for the Forth return stack pointer. The stack is affected implicitly by **CALL** and **RET** instructions, but is directly manipulated using **PUSH** and **POP**. These are single-operand instructions, where the operand may be an immediate value, register, or memory location.

The hardware stack is a convenient place for temporarily saving a register's contents while you use that register for something else.

SwiftForth uses **EBP** as its data stack pointer. However, this stack must be managed more directly, since **PUSH** and **POP** are specifically tied to **ESP**. Both stacks are considered to grow towards low memory. That is, if you do a **PUSH**, **ESP** will be decremented; if you do a **POP**, it will be incremented. Correspondingly, to push something on the data stack, decrement **EBP**; to pop something, increment **EBP**. Data stack management is slightly complicated (but its performance is improved) by SwiftForth's practice of keeping the top-of-stack item in **EBX**.

To facilitate use of the data stack pointer, SwiftForth provides the assembler macros **PUSH(EBX)** and **POP(EBX)** that push and pop the top stack item in **EBX**.

Here are some examples of data stack management:

- **0 [EBX] EBX MOV**
  Replace the top item with the contents of its address (the action of **@**).
- **4 # EBP SUB EBX 0 [EBP] MOV**
  Push the contents of **EBX** onto the data stack (the action of **PUSH(EBX)**).
- **PUSH(EBX) 4 [EBP] EBX**

Push a copy of the second stack item on top of the stack (the action of **OVER**).

- **4 [EBP] EBX MOV 8 # EBP ADD**
  Drop the top two stack items (the action of **2DROP**).

6.5.5.3 User Variable Addressing

ESI contains the address of the start of the user area. User variables are defined in terms of an offset from this area. When a user variable is executed, an absolute address is returned which is the address of the start of the current task's user area plus the offset to this user variable.

The assembler line:

```
BASE [ESI] EBX MOV
```

would load **EBX** from the address *userarea* + *userarea* + *offset*, which is wrong. But the phrase

```
BASE STATUS - [ESI] EBX MOV
```

would load the real contents of **BASE** into **EBX**, subtracting off the extraneous *user-area*.

Since this is rather cumbersome, SwiftForth provides the following shorthand notation:

```
: [U] ( useraddr -- offset ) STATUS - [ESI] ;
```

which is available as an assembler addressing mode. So, the appropriate form of an instruction to load **EBX** from **BASE** becomes:

```
BASE [U] EBX MOV
```

Glossary

[U] *(addr — +addr)*
Provides the appropriate addressing mode for accessing a user variable at *addr* by converting *addr* to an offset *+addr* in the user area, which may be applied as an index to register **ESI**.

References User variables, Section 7.2.1

6.6 Instruction Mode Specifiers

Mode specifiers, including instruction prefixes, modify the action of instructions. These include:

- **Size specifiers** control whether the instruction affects data elements whose size is eight bits, 16 bits, etc.
- **Repeat prefixes** control string instructions, causing them to act on whole strings instead of on single data elements.

- **Segment register overrides** control what memory space is affected (not allowed in user-mode programs).

6.6.1 Size Specifiers

The size specifiers defined in SwiftForth are listed in Table 8. These words must be used *after* a memory reference operand such as those discussed in Section 6.5.5, and will modify its size attribute.

Table 8: Size specifiers

Specifier	Description
BYTE	8-bit
WORD	16-bit integer
DWORD	32-bit integer and real
QWORD	64-bit integer and real
TBYTE	BCD or 80-bit internal real (floating point)

For example:

- **`$40 # 5 [EBX] BYTE TEST`**
  Test the bit masked by $40 in the byte at EBX+5 (the immediate bit in a name field).
- **`0 [EAX] TBYTE FLD`**
  Push the floating-point value pointed to by **EAX** onto the numeric stack.

6.6.2 Repeat Prefixes

The 386 family provides a group of string operators that use **ESI** as a pointer to a *source string*, **EDI** as a pointer to a *destination string*, and **ECX** as a *count register*. These instructions can do a variety of things, including move, compare, and search the strings. The string instructions can operate on single data elements or can process the entire string. In the default mode, they operate on a single item, and automatically adjust the registers to prepare for the next item. If, however, a string instruction is preceded by one of the prefixes in Table 9, it will repeat until one of the terminating conditions is encountered.

Table 9: Repeat prefixes

Prefix	Description
REP	Repeat until **ECX** = 0
REPE, **REPZ**	Repeat until **ECX** = 0 or ZF = 0
REPNE, **REPNZ**	Repeat until **ECX** ≠ 0 or ZF = 1

6.7 Direct Branches

In Forth, most direct branches are performed using structures (such as those described above) and code endings (described below). Good Forth programming style involves many short, self-contained definitions (either code or high level), without the unstructured branching and long code sequences that are characteristic of conventional assembly language. The Forth approach is also consistent with principles of structured programming, which favor small, simple modules with one entry point, one exit point, and simple internal structures.

However, direct transfers are useful at times, particularly when compactness of the compiled code or extreme performance requirements override other criteria. The SwiftForth assembler supports **JMP**, **CALL**, and all forms of conditional branches, although most conditional branching is done using the structure words described in Section 6.8.

A **CALL** is automatically generated when a word defined by **CODE** is referenced inside a colon definition, but you may also use **CALL** in assembly code. The argument to **CALL** must be the address of the code field of a subroutine that ends with an **RET**.

The normal way to define a piece of code intended to be referenced from other code routines is to use **LABEL** (described below). To get a suitable address for a word defined by **CODE** or **ICODE**, you must use the form:

```
' <name> >CODE CALL
```

The word **>CODE** transforms the execution token returned by **'** to a suitable code field address.

LABEL is used in the form described in Section 6.2. Invoking *name* returns the address identified by the label, which may be used as a destination for either a **JMP** or a **CALL**.

For example, this code fragment is used by code that constructs a temporary data buffer to provide a substitute return address:

```
LABEL R-GO-ON
   EAX JMP                  \ jump to address in EAX
   END-CODE
```

It is also possible to define local branch destinations within a single code routine, using the form <n> **L:** where *n* is 0–19. To reference a local label of this kind, use <n> **L#** where *n* is the number of the desired destination. You may reference such local labels either in forward or backward branches within the routine in which they were defined. For example:

```
CODE UBETWEEN ( u ulo uhi -- flag )
    0 [EBP] EAX MOV             \ get ulo
    4 [EBP] EDX MOV             \ get u
    EAX EDX CMP                 \ compare ulo with u
    8 [EBP] EBP LEA             \ discard unused space, preserve flags
    1 L# JB                     \ u is below ulo, return zero
    EBX EDX CMP                 \ compare uhi with u
    1 L# JA                     \ u is above uhi, return zero
```

```
    -1 # EBX MOV                \ return true
    RET
1 L:                            \ here if returning false
    EBX EBX SUB                 \ zero
    RET END-CODE
```

In this example, the label **1 L:** identifies the code for the false case, which is branched to from two locations above.

Although labels such as this are standard practice in assembly language programming, they tend to encourage unstructured and unmaintainable code. We strongly recommend that you keep your code routines short and use the structure words in Section 6.8 in preference to local labels.

Glossary

L: *(n —)*

Define local label *n* and resolve any outstanding forward branches to it. Local labels can only be referenced within the routine in which they are defined.

L# *(n — addr)*

Reference local label *n*, and leave its address to be used by a subsequent branch instruction.

6.8 Assembler Structures

In conventional assembly language programming, control structures (loops and conditionals) are handled with explicit branches to labeled locations. This is contrary to principles of structured programming, and is less readable and maintainable than high-level language structures.

Forth assemblers in general, and SwiftForth in particular, address this problem by providing a set of program-flow macros, listed in the glossary at the end of this section. These macros provide for loops and conditional transfers in a structured manner, and work like their high-level counterparts. However, whereas high-level Forth structure words such as **IF**, **WHILE**, and **UNTIL** test the top of the stack, their assembler counterparts test the processor condition codes.

The structures supported in this assembler (and others from FORTH, Inc.) are:

```
BEGIN  <code to be repeated> AGAIN
BEGIN  <code to be repeated> <cc> UNTIL
BEGIN  <code>  <cc> WHILE  <more code>  REPEAT
<cc> IF  <true case code>  ELSE  <false case code>  THEN
```

In the sequences above, *cc* represents condition codes, which are listed in a glossary beginning on page 91. The combination of a condition code and a structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional branch instruction *Bcc*, where *cc* is the condition code. The other components of the structures — **BEGIN**, **REPEAT**, **ELSE**, and **THEN** — enable the assembler to provide an appropriate destination address for the branch.

In addition, the Intel instructions **LOOP**, **LOOPE**, and **LOOPNE** may be used with **BEGIN** to make a loop. For example:

```
BEGIN
LODSB                       \ read a char
BL AL CMP                   \ check for control
U< IF                       \ if control
CHAR ^ # AL MOV             \ replace with caret
THEN
STOSB                       \ write the char
LOOP                        \ and repeat
```

All conditional branches use the results of the previous operation which affected the necessary condition bits. Thus:

```
EAX EBX CMP  < IF
```

executes the true branch of the **IF** structure if the contents of **EAX** is less than the contents of **EBX**.

The word **NOT** following a condition code inverts its sense. Since the name **NOT** is also the name of an opcode mnemonic, the SwiftForth assembler will examine the stack, and if a valid condition code is present, it will invert it; otherwise, it will assemble a **NOT** instruction.

In high-level Forth words, control structures must be complete within a single definition. In assembler, this is relaxed; the assembler will automatically assemble a short or long form for all relative jump, call, and loop instructions. Control structures that span routines are not recommended, however—they make the source code harder to understand and harder to modify.

Table 10 shows the instructions generated by SwiftForth condition codes in combination with words such as **IF** or **UNTIL**. See the glossary below for details. Refer to your processor manual for details about the condition bits.

Note that the standard Forth syntax for sequences such as **0< IF** implies *no branch in the true case*. Therefore, the combination of the condition code and branch instruction assembled by **IF**, etc., branch on the *opposite* condition (i.e., ≥ 0 in this case).

These constructs provide a level of logical control that is unusual in assembler-level code. Although they may be intermeshed, care is necessary in stack management, because **REPEAT**, **UNTIL**, **AGAIN**, **ELSE**, and **THEN** use the addresses on the stack.

Table 10: SwiftForth condition codes and conditional jumps

Intel Instruction	SwiftForth Condition (with IF or UNTIL)
JA	**U> NOT**
JAE	**U<**
JAE	**CY**
JB	**U< NOT**
JECXZ	**1NZ**

Intel Instruction	SwiftForth Condition (with IF or UNTIL)
JBE	**U>**
JC	**U< NOT**
JC	**CY NOT**
JCXZ	**1NZ**
JNE	**0=**

Table 10: SwiftForth condition codes and conditional jumps *(continued)*

Intel Instruction	SwiftForth Condition (with IF or UNTIL)
JE	**0= NOT**
JZ	**0= NOT**
JG	**S> NOT**
JGE	**0> NOT**
JL	**S< NOT**
JLE	**S>**
JLE	**0>**
JNA	**U>**
JNAE	**U< NOT**
JNAE	**CY NOT**
JNB	**U<**
JNB	**CY**
JNBE	**U> NOT**
JNC	**U<**

Intel Instruction	SwiftForth Condition (with IF or UNTIL)
JNG	**S>**
JNG	**0>**
JNGE	**S< NOT**
JNL	**S<**
JNLE	**S> NOT**
JNO	**0V**
JNP	**PE**
JNS	**0<**
JNZ	**0=**
JO	**OV**
JP	**PE NOT**
JPE	**PE NOT**
JPO	**PE**
JS	**0< NOT**

In the glossary entries below, the stack notation *cc* refers to a condition code. Available condition codes are listed beginning on page 91.

Glossary

Branch Macros

BEGIN *(— addr)*

Leave the current address *addr* on the stack, to serve as a destination for a branch. Doesn't assemble anything.

AGAIN *(addr —)*

Assemble an unconditional branch to *addr*.

UNTIL *(addr cc —)*

Assemble a conditional branch to *addr*. **UNTIL** must be preceded by one of the condition codes (see below).

WHILE *(addr$_1$ cc — addr$_2$ addr$_1$)*

Assemble a conditional branch whose destination address is left empty, and leave the address of the branch *addr* on the stack. A condition code (see below) must precede **WHILE**.

REPEAT *(addr$_2$ addr$_1$ —)*

Set the destination address of the branch that is at *addr$_1$* (presumably having been left by **WHILE**) to point to the next location in code space, which is outside the loop. Assemble an unconditional branch to the location *addr$_2$* (presumably left by a preceding **BEGIN**).

IF *(cc — addr)*

Assemble a conditional branch whose destination address is not given, and leave the address of the branch on the stack. A condition code (see below) must precede

IF.

ELSE *($addr_1$ — $addr_2$)*

Set the destination address $addr_1$ of the preceding **IF** to the next word, and assemble an unconditional branch (with unspecified destination) whose address $addr_2$ is left on the stack.

THEN *(addr —)*

Set the destination address of a branch at *addr* (presumably left by **IF** or **ELSE**) to point to the next location in code space. Doesn't assemble anything.

Condition Codes

0< *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on positive.

0= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on non-zero.

0> *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on zero or negative.

0<> *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on zero.

0>= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on negative.

U< *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on unsigned greater-than-or-equal.

U> *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on unsigned less-than-or-equal.

U>= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on unsigned less-than.

U<= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on unsigned greater-than.

< *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on greater-than-or-equal.

> *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on less-than-or-equal.

>= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on less-than.

<= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on greater-than.

CS *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on carry clear. This condition is the same as **U<**.

CC *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on carry set.

PE *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on parity odd.

PO *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on parity even.

OV *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on overflow not set.

ECXNZ *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on **CX** equal to zero.

NEVER *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate an unconditional branch.

NOT *(cc_1 — cc_2)*

Invert the condition code cc_1 to give cc_2.

Note that the sense of phrases such as **< IF** and **> IF** is parallel to that in high-level Forth; that is, since **< IF** will generate a branch on greater-than-or-equal, the code following **IF** will be executed if the test fails. For example:

```
CODE Test  ( n1 -- n2 )
   10 #  EBX CMP < IF 0 # EBX MOV THEN
   RET END-CODE
```

When this is executed, one gets the result:

```
12 test . 12 ok
```

```
8 test . 0 ok
```

6.9 Writing Assembly Language Macros

The important thing to remember when considering assembler macros is that the various elements in SwiftForth assembler instructions (register names, addressing mode specifiers, mnemonics, etc.) are Forth words that are executed to create machine language instructions. Given that this is the case, if you include such words in a colon definition, they will be executed when that definition is executed, and will construct machine language instructions at that time, i.e., expanding the macro. Therefore:

An assembly language macro in SwiftForth is a colon definition whose contents include assembler commands.

The only complication lies in the fact that SwiftForth assembler commands are not normally "visible" in the search order (see the discussion of search orders in Section 5.5.2). This is necessary because there are assembler versions **IF**, **WHILE**, and other words that have very different meanings in high-level Forth. When you use **CODE**, **ICODE**, or **LABEL** to start a code definition, those words automatically select the assembler search order, and **END-CODE** restores the previous search order. However, to make macros, you will need to manipulate search orders more directly.

Relevant commands for manipulating vocabularies for assembler macros are given in the glossary at the end of this section. Here are a few simple examples.

Example 1: POP(EBX)

```
: POP(EBX)   ( -- )              \ Pop data stack
   [+ASSEMBLER]
   0 [EBP] EBX MOV               \ Move top-of-stack to EBX
   4 # EBP ADD                   \ Pop stack pointer
   [PREVIOUS] ;
```

This is a macro included with SwiftForth (described in Section 6.5.5.2). Its purpose is to **POP** the data stack, in a fashion analogous to **POP** on the hardware stack (SwiftForth's return stack). Since SwiftForth keeps its top stack item in the register **EBX**, the operation consists of moving what was the second stack item (pointed to by **EBP**) into **EBX**, and then incrementing **EBP** by one cell. These two machine instructions **MOV** and **ADD** will be assembled in place whenever **POP(EBX)** is used in code.

Example 2: High-level comparisons

```
ASSEMBLER
: BINARY-TEST ( cc -- )
   [+ASSEMBLER]\
   EBX 0 [EBP] CMP             \ compare top with second
   ( cc) IF                    \ if the condition is true
   -1 # EBX MOV                \ return TRUE
   ELSE                        \ otherwise
   EBX EBX SUB                 \ return false
   THEN
```

```
   4 # EBP ADD                \ discard unused stack item
   RTS  END-CODE ;            \ end routine
PREVIOUS
ICODE <  ( u u -- flag )     < BINARY-TEST
ICODE >  ( u u -- flag )     > BINARY-TEST
ICODE =  ( n n -- flag )    0= BINARY-TEST
```

This example shows how the high-level Forth two-item (binary) comparison operators are defined (only the first few are actually given here; all are defined similarly). All share a common behavior: they compare two items on the stack, removing both and leaving the results of the comparison as a well-formed flag (i.e., *true* = -1). They *differ* only in the actual comparison to be performed.

As noted in Section 6.8 above, the assembler's **IF** takes a condition code on the stack. So the macro **BINARY-TEST** also takes a condition code on the stack, which is the argument to **IF**. All the rest of the logic in performing the comparison is the same.

Glossary

ASSEMBLER *(—)*
Add the assembler vocabulary to the top of the current search order.

[+ASSEMBLER] *(—)*
Add the assembler vocabulary to the top of the current search order. An immediate word (will be executed immediately when used inside a colon definition).

[FORTH] *(—)*
Add the main Forth vocabulary to the top of the current search order. An immediate word (will be executed immediately when used inside a colon definition).

[PREVIOUS] *(—)*
Remove the top of the current search order. Often used to "undo" **[FORTH]** or **[+ASSEMBLER]**. An immediate word (will be executed immediately when used inside a colon definition).

References Search orders in Forth, *Forth Programmer's Handbook*
Wordlists and vocabularies in SwiftForth, Section 5.5.2

Section 7: Multitasking

Linux and macOS are inherently multitasked environments. The actual task management algorithms may vary between implementations, but the interface from the perspective of a program such as SwiftForth and its applications is the same.

7.1 Basic Concepts

This section discusses the underlying principles behind both Linux and macOS multitasking and SwiftForth's facilities for defining and controlling multiple tasks or threads.

7.1.1 Definitions

Multitasking in the Linux and macOS environments works on different levels:

- **Multiple processes** (e.g., user logins, service daemons, application programs) can be started and running concurrently. Each has its own resources (memory, CPU registers and stacks, etc.). Each is a completely different program; they are not sharing any code, although they might call some of the same libraries. They might also communicate with one another, but they are still separate programs.
- **Multiple threads** can be launched from within a process. A thread is a piece of code (often a loop) that the OS executes separately and concurrently. The thread may use some of the resources of the process that launched it (the "parent" process), meaning it is executing code that resides in the parent's memory, using its registers, etc. However, the code is running asynchronously to the parent application, with the OS controlling the scheduling according to its priority with respect to others. The parent process launches the thread and sets its priority; the parent can also start, stop, or kill it.

  A SwiftForth program can support multiple threads. A thread is in some respects similar to a *background task* in other FORTH, Inc. products, in that it is given its own stacks and user variables, and is executing code from within the SwiftForth process's dictionary. However, because the OS controls execution and task switching, there is no equivalent of the round-robin loop found in pF/x or SwiftOS.

  Just as you can launch multiple instances of application programs, you can also launch multiple instances of SwiftForth. These function as completely independent programs, each of which might have its own multiple threads.

7.1.2 Forth Re-entrancy and Multitasking

When more than one task can share a piece of code, that code can be called *re-entrant*. Re-entrancy is valuable in a multitasking system, because it facilitates

shared code and simplifies inter-task communication.

Routines that are not re-entrant are those containing elements subject to change while the program runs. For example, routines that use *global variables* are not re-entrant.

Forth routines can be made completely re-entrant with very little effort. Most keep their intermediate results on the data stack or the return stack. Programs to handle strings or arrays can be designed to keep their data in the section of memory allotted to each task. It is possible to define public routines to access variables, and still retain re-entrancy, by providing private versions of these variables to each task; such variables are called *user variables*.

Because re-entrancy is easily achieved, tasks may share routines in a single program space. In SwiftForth, the entire dictionary is shared among tasks.

References User variables, Section 7.2.1

7.2 SwiftForth Tasks

A *task* is a system thread with additional facilities assigned by SwiftForth. It may be thought of as an entity capable of independently executing Forth definitions. It may be given permanent or temporary job assignments. If it is given a permanent job assignment, the recommended naming convention is a *job title*. For example, a task that will acquire data from a remote device attached to a serial port might be called **MONITOR**.

A task has a separate stack frame assigned to it by the OS, containing its data and return stacks and user variables. SwiftForth may read and write a task's user variables, but must not modify its stacks.

7.2.1 User Variables

In SwiftForth, tasks can share code for system and application functions, but each task may have different data for certain facilities. The fact that all tasks have private copies of variable data for such shared functions enables them to run concurrently without conflicts. For example, number conversion in one task needs to control its **BASE** value without affecting that of other tasks.

Such private variables are referred to as *user variables*. User variables are not shared by tasks; each task has its own set, kept in the task's *user area*.

- Executing the name of a user variable returns the address of that variable within the task that executes it.
- Invoking `<taskname> @` returns the address of the first user variable in *task-name*'s user area. (That variable is generally named **STATUS**.)

Some user variables are defined by the system for its use; they may be found in the file **SwiftForth/src/kernel/data.f**. You may add more in your application, if you need

to in order to preserve re-entrancy, to provide private copies of the application-specific data a task might need.

User variables are defined by the defining word **+USER**, which expects on the stack an offset into the user area plus a size (in bytes) of the new user variable being defined. A copy of the offset will be compiled into the definition of the new word, and the size will be added to it and left on the stack for the next use of **+USER**. Thus, when specifying a series of user variables, all you have to do is start with an initial offset (**#USER**) and then specify the sizes. When you are finished defining **+USER** variables, you may save the current offset to facilitate adding more later, i.e., **<n> TO #USER**.

It is good practice to group user variables as much as possible, because they are difficult to keep track of if they are scattered throughout your source.

When a task executes a user variable, its offset is added to the register containing the address of the currently executing task's user area. Therefore, all defined user variables are available to all tasks that have a user area large enough to contain them (see task definition, Section 7.2.3).

A task may need to initialize another task's user variables, or to read or modify them. The word **HIS** allows a task to access another task's user variables. **HIS** takes two arguments: the address of the task of interest, and the address of the user variable of interest. For example:

```
2 MONITOR BASE HIS !
```

will set the user variable **BASE** of the task named **MONITOR** to 2 (binary). In this example, **HIS** takes the address of the executing task's **BASE** and subtracts the address of the start of the executing task's user area from it to get the offset, then adds the offset to the user area address of the desired task, supplied by **MONIITOR**.

The glossaries below list words used to manage the user variables. The actual user variables are listed in **SwiftForth/src/kernel/data.f**.

Glossary

+USER (n_1 n_2 — n_3)

Define a user variable at offset n_1 in the user area, and increment the offset by the size n_2 to give a new offset n_3.

#USER (— n)

A **VALUE** that returns the number of bytes currently allocated in a user area. This is the offset for the next user variable when this word is used to start a sequence of **+USER** definitions intended to add to previously defined user variables.

HIS ($addr_1$ n — $addr_2$)

Given a task address $addr_1$ and user variable offset n, returns the address of the referenced user variable in that task's user area. Usage:

```
<task-name> <user-variable-name> HIS
```

7.2.2 Sharing Resources

Some system resources must be shared by tasks without giving any single task permanent control of them. Devices, non-reentrant routines, and shared data areas are all examples of resources available to any task but limited to use by only one task at a time. SwiftForth provides two levels of control: control of an individual resource, or control of the CPU itself within the SwiftForth process.

SwiftForth controls access to these resources with two words that resemble Dijkstra's semaphore operations. (Dijkstra, E.W., *Comm. ACM*, 18, 9, 569.) These words are **GET** and **RELEASE**.

As an example of their use, consider an A/D multiplexer. Various tasks in the system are monitoring certain channels. But it is important that while a conversion is in process, no other task issue a conflicting request. So you might define:

```
VARIABLE MUX
: A/D ( ch# -- n )   \ Read a value from channel ch#
   MUX GET  (A/D)  MUX RELEASE ;
```

In the example above, the word **A/D** requires private use of the multiplexer while it obtains a value using the lower-level word **(A/D)**. The phrase **MUX GET** obtains private access to this resource. The phrase **MUX RELEASE** releases it.

In the example above, **MUX** is an example of a *facility variable*. A facility variable behaves like a normal **VARIABLE**. When it contains zero, no task is using the facility it represents. When a facility is in use, its facility variable contains the address of the **STATUS** of the task that owns the facility. The word **GET** waits, executing **PAUSE** repeatedly, until the facility is free or is owned by the task that is running **GET**.

RELEASE checks to see whether a facility is free or is already owned by the task that is executing **RELEASE**. If it is owned by the current task, **RELEASE** stores a zero into the facility variable. If it is owned by another task, **RELEASE** does nothing.

Note that **GET** and **RELEASE** can be used safely by any task at any time, as they don't let any task take a facility from another.

SwiftForth does not have any safeguards against *deadlocks*, in which two (or more) tasks conflict because each wants a resource the other has. For example:

```
: 1HANG   MUX GET  TAPE GET ... ;
: 2HANG   TAPE GET  MUX GET ... ;
```

If **1HANG** and **2HANG** are run by different tasks, the tasks could eventually deadlock.

The best way to avoid deadlocks is to get facilities one at a time, if possible! If you have to get two resources at the same time, it is safest to always request them in the same order. In the multiplexer/tape case, the programmer could use **A/D** to obtain one or more values stored in a buffer, then move them to tape. In almost all cases, there is a simple way to avoid concurrent **GET**s. However, in a poorly written application the conflicting requests might occur on different nesting levels, hiding the problem until a conflict occurs.

It is better to design an application to **GET** only one resource at a time—deadlocks

are impossible in such a system.

Glossary

GET *(addr —)*

Obtain control of the facility variable at *addr*, having first executed **PAUSE** (to allow other tasks to run). If the facility is owned by another task, the task executing **GET** will wait until the facility is available.

RELEASE *(addr —)*

Relinquish the facility variable at *addr*. If the task executing **RELEASE** did not previously own the facility, this operation is a no-op.

7.2.3 Task Definition and Control

There are two phases to task management: *definition* and *instantiation*.

When a task is defined, it gets a dictionary entry containing a *Task Control Block*, or TCB, which is the table containing its size and other parameters. This happens when a SwiftForth program is compiled, and the task's definition and TCB are permanent parts of the SwiftForth dictionary.

When a task is instantiated, the OS is requested to allocate a private stack frame to it, within which SwiftForth sets up its data and return stacks and user variables. At this time, the task is also assigned its *behavior*, or words to execute.

After SwiftForth has instantiated a task, it may communicate with it via the shared memory that is visible to both SwiftForth and the task, or via the task's user variables.

A task is defined using the sequence:

```
<size> TASK <taskname>
```

where *size* is the requested size of its user area and data stack, combined. When invoked, *taskname* returns the address of the task's TCB.

Task instantiation must be done inside a colon definition, using the form:

```
: <name>   <taskname> ACTIVATE <words to execute> ;
```

When *name* is executed, the task *taskname* will be instantiated and will begin executing the words that follow **ACTIVATE**.

The task's assigned behavior, represented by *words to execute* above, may be one of two types:

- **Transitory behavior**, which the task simply executes and then terminates.
- **Persistent behavior**, represented by an infinite (e.g., **BEGIN … AGAIN**) loop which the task will perform forever.

Transitory behavior simply ends in EXIT or ; (semicolon). A task that has terminated

in this fashion may be instantiated again, to perform the same or a different transitory behavior.

Persistent behavior must include the infinite loop and, within that loop, provision should be made for the task to relinquish the CPU using **PAUSE** or a word that calls a system function that blocks the thread. If this is not done, the task will consume all available CPU time (subject to the OS time slicing) and performance of other tasks and programs will degrade.

A task that activates another task is that task's *owner*. A task may re-activate another task, or **HALT** it, which causes it to cease operation permanently.

Glossary

TASK <taskname> *(u —)*
Define a task,whose dictionary size will be *u* bytes in size. Invoking *taskname* returns the address of the task's Task Control Block (TCB).

CONSTRUCT *(addr —)*
Instantiate the task whose TCB is at *addr* (if not already done by **CONSTRUCT**), and leave a pointer to the task's memory in the first cell of the TDB. If the task has already been instantiated (i.e. the first cell of the TDB is not zero), **CONSTRUCT** does nothing. The use of **CONSTRUCT** is optional; **ACTIVATE** will do a **CONSTRUCT** automatically if needed.

ACTIVATE *(addr —)*
Instantiate the task whose TCB is at *addr*, and start it executing the words following **ACTIVATE**. Must be used inside a definition. If the rest of the definition after **ACTIVATE** is structured as an infinite loop , it must call **PAUSE** within the loop so task control can function properly.

HALT *(addr —)*
Cause the task whose TCB is at *addr* to cease operation permanently at the next **PAUSE**, but to remain instantiated. The task may be reactivated.

KILL *(addr —)*
Cause the task whose TCB is at *addr* to cease operation and release all its memory. The task may be reactivated.

PAUSE *(—)*
Causes the thread to relinquish the CPU.

References **MS**, used for interval timing, Section 4.4.2.

SECTION 8: SYSTEM FUNCTIONS

In this section, we describe how to use SwiftForth to access Linux and macOS dynamically loaded libraries, export Forth words as “callback” functions to external programs and libraries, and access command-line arguments and environment variables.

8.1 Dynamically Loaded (DL) Libraries

Dynamically loaded (DL) libraries are libraries that are loaded at times other than during the startup of a program. They're particularly useful for implementing plugins or modules, because they permit waiting to load the plugin until it's needed. They're also useful for implementing interpreters that wish to occasionally compile their code into machine code and use the compiled version for efficiency purposes, all without stopping. For example, this approach can be useful in implementing a just-in-time compiler like SwiftForth.

Dynamically loaded libraries are built as standard object files or standard shared libraries. However, these libraries aren't automatically loaded at program link time or startup; instead, there is an API for opening a library, looking up symbols, handling errors, and closing the library. SwiftForth uses this API to provide a simple DL library function import mechanism that resembles the C function prototype.

8.1.1 Importing Library Functions and Data

There are two simple steps to importing a library function:

1. Declare which library functions are to be imported from.
2. Define the function interface.

The word **LIBRARY** accomplishes the first step:

```
LIBRARY <filename>
```

Thereafter, functions in *filename* will be available for import, using the following procedure.

To make a Forth word that references a function in a library, use the form:

```
FUNCTION: <name> ( params -- return )
```

The left side of the Forth stack notation used mirrors the parameters in the C function prototype (usually listed on the function’s man page). The right side should be a single item (the return value) or it may be empty (nothing returned).

A special case of the above stack notation is used for functions that take a variable number of args (sometimes called “varargs” functions):

```
FUNCTION: <name> ( ... -- return )
```

At run-time, the number of arguments to be passed to the function call must be on top of the stack.

Note that the order of the parameters is the same left-to-right order specified by the C function prototype. Each item is one cell in size, and strings are normally null-terminated.

Note that although the **FUNCTION:** syntax is the same as in SwiftForth for Windows, there's one important difference in the implmementation. The SwiftForth version searches all open libraries for the named function. The Linux and macOS versions only search the most recent library declared by **LIBRARY**.

A library or imported function may be marked as optional. For example:

```
[OPTIONAL] LIBRARY libfoo.so
[OPTIONAL] FUNCTION: fooproc ( n -- ior )
```

If the declared library or imported function is not present at compile-time (or when a turnkey program is launched), no error message is generated. However, a run-time call to an unresolved imported function results in an abort with a message sent to STDERR.

To import global data from a library, use this form instead:

```
GLOBAL: <name>
```

A word defined by **GLOBAL:** differs from a **FUNCTION:** word in three ways:

1. Does not require a stack picture to follow it.
2. Does not take any parameters from the data stack.
3. Returns the address of the global data; does not call any imported procedure.

Glossary

LIBRARY <filename> *(—)*
Uses library *filename* for subsequent imports by **FUNCTION:**.

FUNCTION: <name> <parameter list> *(—)*
Defines a named call to a library function, which takes the number of parameters shown in the parameter list, in the form of a stack comment: **(** <input parameters> **--** <result> **)**. Imported functions return at most one value on the stack.

GLOBAL: <name> *(—)*
Defines a named global data location in the current library. When executed, *name* returns the global data address.

.LIBS *(—)*
Displays a list of all available libraries previously opened with **LIBRARY**).

.IMPORTS *(—)*
Displays a list of all currently available functions imported by **FUNCTION:** For each entry, shows the resolved address, the library from which the function was

imported, and the function name..

[OPTIONAL] (—)
Specifies that the next library or function declaration is optional.

8.2 System Callbacks

A *callback* is an entry point in your program. It is provided for the OS to call in certain circumstances. For example, in order to handle signals from the OS, the system function **sigaction** is called and is passed the address of a callback in one of its parameters.

A callback is in many respects analogous to an interrupt in other programming environments, in that the code that responds to it is not executing as a sequential part of your application. In SwiftForth, callbacks are handled by a synthetic, transient task with its own stacks and user area, separate from any tasks your application may be running and existing only for the duration of the callback function. Callbacks may execute any re-entrant SwiftForth words, but may not directly communicate with the running program other than by storing data where the program may find it.

You may define a callback with any number of parameters, using the form:

```
<xt> <n> CB: <name>
```

where *xt* identifies the routine to respond to the callback, and *n* is the number of parameters passed to the callback. Note that n is used to generate the stack "clean up" code associated with the callback, so it must be correct!

A callback must always return a single integer result, which is used as the return value from the callback function. The defining word **CB:** "wraps" the execution of the *xt* with the necessary code to construct and discard the transient task environment in which the callback will run.

The Forth data stack on entry to the callback is empty. The parameters passed to the callback function may be accessed by the **_PARAM** words listed in the glossary below.

Beacuse SwiftForth has constructed a temporary task area in which the callback executes, a callback may use stacks, **HERE**, **PAD**, and the output number conversion buffer. Both the data and return stacks are empty on entry to the callback, and its **BASE** is decimal. However, the callback has no dictionary or private data space.

The only user variables initialized on entry to the callback are **S0**, **R0**, **H**, and **BASE**. You may use other user variables for temporary storage, but remember that the entire task area will be discarded when the callback exits. You may also use global data objects (e.g., **VARIABLE**s), but you must do so with great care, because they will be shared.

Glossary

CB: *(xt n —)*
Defines a callback, which will execute the code associated with *xt* and take *n* parameters.

_PARAM_0, **_PARAM_1**, **_PARAM_2**, **_PARAM_3** *(— x)*
_PARAM_4, **_PARAM_5**, **_PARAM_6**, **_PARAM_7**
Return the first through the eighth parameters passed in on the callback's stack frame, respectively.

NTH_PARAM <name> *(n —)*
Defines a callback stack frame parameter, appearing in the *n*th position from the top of the stack frame (0 = top). **_PARAM_0** through **_PARAM_7** above are defined by **NTH_PARAM**.

8.3 Command-line Arguments

The main entry point of a program is called by the operating system using this function prototype:

```
int main(int argc, char *argv[])
```

Although any name could be given to these parameters, they are usually referred to as **argc** and **argv**. The first parameter, **argc** (argument count) is an integer that indicates how many arguments were entered on the command line when the program was started. The second parameter, **argv** (argument vector), is an array of pointers to arrays of character objects. The array objects are null-terminated strings, representing the arguments that were entered on the command line when the program was started.

The first element of the array, **argv[0]**, is a pointer to the character array that contains the program name or invocation name of the program that is being run from the command line. The second element, **argv[1]** points to the first argument passed to the program, **argv[2]** points to the second argument, and so on.

SwiftForth provides the following words to access the command-line arguments:

Glossary

ARGC *(— n)*
Returns the command-line argument count. This is always at least one (for argument zero, the name of the program itself).

ARGV *(i — addr len)*
Returns argument *i* as a string. If *i* is an invalid argument number (i.e., not less than **ARGC**), a zero-length string is returned.

CMDLINE *(— addr len)*
Concatenates the command-line **ARGV** strings into a single buffer and returns its address and count. Note: Arguments are concatenated to a counted string buffer, so

the max length is 255.

8.4 Environment Queries

The operating system environment variables can be queried with **FIND-ENV**. For example, if the environment contains the entry "SHELL=/bin/ksh" then the query

```
S" SHELL" FIND-ENV
```

returns the address and length of the string "/bin/ksh". Do not include the "=" in the search string; **FIND-ENV** appends it to its search buffer.

The command **.ENV** (or its synonym **PRINTENV**) displays the entire list of environment variables for diagnostic purposes.

Glossary

FIND-ENV *(addr1 len1 — addr2 len2 flag)*
Searches for the string *addr1 len1* in the environment and returns its value as string *addr2 len2* and true if found. Returns the original string parameters and false if not found.

.ENV
PRINTENV (—)
Displays the environment for diagnostic purposes.

ATOI *(addr len — n)*
Converts a counted string to a single integer. Returns 0 if it fails. "0x" preceeds hex numbers; all others are decimal.

GETCH *(addr1 len1 — addr2 len2 char)*
Removes the first character from the string *addr1 len1* and returns the remaining string *addr2 len2* and the character.

SECTION 9: SWIFTFORTH OBJECT-ORIENTED PROGRAMMING (SWOOP)

SwiftForth includes an object-oriented programming package called SWOOP[1], SwiftForth Object-Oriented Programming. It includes all the essential features of object-oriented programming, including:

- **Encapsulation:** combining data and methods into a single package that responds to messages.
- **Information hiding:** the ability of an object to possess data and methods that are not accessible outside its class.
- **Inheritance:** the ability to define a new class based on a previously defined ("parent") class. The new class automatically possesses all members of the parent; it may add to or replace these members, or define behaviors for deferred members.
- **Polymorphism:** the ability of different sub-classes of a class to respond to the same message in different ways. For example, all vehicles can steer, but bicycles do it differently from automobiles.

This section describes the essential features of SWOOP.

9.1 Basic Components

This section will present a simple example of a class for the purpose of discussing its members, an instantiation of the class, and its use. This example will be extended in various ways in subsequent sections.

9.1.1 A Simple Example

POINT (defined below) is a simple class we shall use as a primary building-block example for SWOOP. It demonstrates two of the four basic class member types: *data* and *colon*.

The word following **CLASS** is the name of the class; all definitions between **CLASS** and **END-CLASS** belong to it. These definitions are referred to as the *members* of the class. When a class name is executed, it leaves its handle (*hclass*) on the stack. The constructor words are the primary consumers of this handle.

```
CLASS POINT
    VARIABLE X
    VARIABLE Y
    : SHOW ( -- )   X @ . Y @ . ;
    : DOT ( -- )   ." Point at " SHOW ;
END-CLASS
```

1.Portions of this chapter adapted, with permission, from material that originally appeared in *Forth Dimensions*, a publication of the not-for-profit Forth Interest Group (www.forth.org).

The class definition itself does not allocate any instance storage; it only records how much storage is required for each instance of the class. **VARIABLE** reserves a cell of space and associates it with a member name.

The *colon members* **SHOW** and **DOT** are like normal Forth colon definitions, but are only valid in the execution context of an object of type **POINT**. **X** and **Y** also behave exactly like normal Forth **VARIABLE**s.

There are five kinds of members:

1. **Data members** include all data definitions. Available data-member-defining words include **CREATE** (normally followed by data compiled with **,** or **C,**), **BUFFER:** (an array whose length is specified in address units), **VARIABLE**, **CVARIABLE** (single *char*), or **CONSTANT**s.
2. **Colon members** are definitions that may act on or use data members.
3. **Other previously defined objects, available within this object.** These are discussed in Section 9.1.4**.**
4. **Deferred members** are colon-like definitions with default behaviors that can be referenced while defining the class, but which may have substitute behaviors defined by sub-classes defined later. These allow for polymorphism and late binding, and will be discussed in Section 9.1.6.
5. **Messages** are un-named responses to arbitrary numeric message values and are discussed in Section 9.1.7.

Glossary

CLASS <name> *(—)*

Begin a class definition that will be terminated by **END-CLASS**. All definitions between **CLASS** and **END-CLASS** are members of the class *name*. Invoking *name* returns the handle (*hclass*) of this class.

END-CLASS *(—)*

End a class definition.

9.1.2 Static Instances of a Class

Having defined a class, we can create an instance of it. **BUILDS** is the *static instance* constructor in SWOOP; it is a Forth defining word and requires the handle of a class on the stack when executed. For example:

```
POINT BUILDS ORIGIN
```

…defines an object named **ORIGIN** that is a static instance of the **POINT** class. Now, any members of **POINT** (e.g., the **X** and **Y** variables defined in the earlier example) may be referenced in the context of **ORIGIN**. For example:

```
5 ORIGIN X !
8 ORIGIN Y !
ORIGIN DOT
```

X and **Y** are data members of **ORIGIN** that were inherited from **POINT** and the values stored into them here are completely independent of the **X** and **Y** members of any other instances of the **POINT** class. **X** and **Y** have no existence independent of an instance of **POINT**.

DOT is a colon member of the **POINT** class. When it executes in the context of a specific instance of **POINT** such as **ORIGIN**, it will use the addresses for the versions of **X** and **Y** belonging to **ORIGIN**.

When the name of an object is executed, two things happen: first, the Forth interpreter's context is modified to include the name space of the class that created it. Second, the address of the object is placed on the stack.

Each of the members of the class act on this address: members that represent data simply add an offset to it; members that are defer or colon definitions push the address into **'SELF** (see Section 9.3.1)—which holds the current object address—before executing, and restore it afterwards.

You can build an array of objects of the same class using **BUILDS[]**. Individual instances of such an array must be provided with an index. For example:

```
20 POINT BUILDS[] ICOSOHEDRON[]
0 ICOSOHEDRON[] DOT
1 ICOSOHEDRON[] DOT
...
19 ICOSOHEDRON[] DOT
```

(The **[]** in the name is used as a reminder that this is an array.)

Glossary

BUILDS <name> *(hclass —)*

Constructs a static instance of the class identified by *hclass*. Use of *name* returns the address of the instance and changes the search order contehclass to reflect the members and methods of the class.

Usage: <classname> **BUILDS** <instance-name>

BUILDS[] <name> *(n hclass —)*

Constructs an array of *n* static instances of the class identified by *hclass*. When invoked, *name* expects an index into the array, and will return the address of the indexed instance.

Usage: <size> <classname> **BUILDS[]** <instance-name>

9.1.3 Dynamic Objects

We can also create a temporary context in which to reference the members of a class. **NEW** is a *dynamic constructor* that will build a temporary instance of a class; it is not a defining word, but is a memory management word similar to **ALLOCATE**. It requires a class handle on the stack, and returns an address. When the object is no longer needed, it can be disposed of with **DESTROY**.

USING parses the word following it, and (assuming that it is the name of a class) makes its members available for use on data at a specified address. For example:

```
0 VALUE FOO                          \ Contains pointer to instance
POINT NEW TO FOO                     \ Construct instance of class POINT
8 FOO USING POINT X !                \ Store data in X
99 FOO USING POINT Y !               \ Store data in Y
FOO USING POINT DOT                  \ Display X and Y
FOO USING POINT DESTROY  0 TO FOO    \ Release space
```

This example uses **FOO** to hold the address of an instance of **POINT**. After the instance is created, it may be manipulated (with a slight change in syntax) in the same way that a static instance of **POINT** is. When it's no longer needed, the instance is destroyed, and the address kept in **FOO** invalidated.

Objects constructed by **NEW** do not exist in the Forth dictionary, and must be explicitly destroyed when no longer used.

Another form of dynamic object instantiation is *local objects*. These, like local variables, are available only inside a single colon definition, and are instantiated only while the definition is being executed. Here's an example:

```
: TEST ( -- )
   [OBJECTS  POINT MAKES JOE  OBJECTS]
   JOE DOT ;
```

You can define as many local objects as you need between **[OBJECTS** and **OBJECTS]**. They will all be instantiated when **TEST** is executed, and destroyed when it is completed.

In the example of **TEST**, the address of **POINT**'s data space is valid while **TEST** is executing, but its namespace is only available within the definition of **TEST** itself. In order to make it possible for a word such as **TEST** to pass addresses within **POINT** to words it may call, there is a second form of local object called **NAMES** that names an arbitrary address and makes members of a specified class available for it. Like **MAKES**, it's used between **[OBJECTS** and **OBJECTS]** inside a definition, and its scope is local to the definition in which it's defined. For example:

```
: TRY ( addr -- )
   [OBJECTS   POINT NAMES SAM  OBJECTS]
   SAM DOT ;

: TEST ( -- )
   [OBJECTS  POINT MAKES JOE  OBJECTS]
   JOE ADDR TRY ;
```

Data space was allocated only once, for **JOE** in **TEST**. Its address was passed to **TRY**, which applied **POINT**'s member **DOT** to the data structure at the address passed to it from **TEST**.

Some examples of this strategy may be found in the file **\Lib\Samples\Win32\ListView.f.**

MAKES and **NAMES** may both be used within the scope of a single **[OBJECTS** ... **OBJECTS]** pair.

Glossary

NEW *(hclass — addr)*
Constructs a temporary (dynamic) instance of the class identified by *hclass* and allocates memory for it, returning the address of the start of the data.

DESTROY *(addr —)*
Releases the dynamically allocated memory at *addr*.

[OBJECTS *(—)*
Begins instantiation of local objects, which will be terminated by **OBJECTS]**. It must be used inside a definition. There may be multiple objects instantiated or named (using **MAKES** and/or **NAMES**) between **[OBJECTS** and **OBJECTS]**, but there may be only one such region in a definition.

OBJECTS] *(—)*
Terminates the instantiation of local objects inside a definition.

MAKES <name> *(hclass — addr)*
Constructs a local (dynamic) instance of the class identified by *hclass*. Use of *name* returns the handle of the instance name.

MAKES must be used inside a definition, between **[OBJECTS** and **OBJECTS]**, and *name* cannot be used outside the definition in which it is instantiated. Memory is allocated for *name* only while the definition in which it is instantiated is being executed. In all other respects, local objects follow the same rules of usage as static objects.

NAMES <name> *(addr hclass —)*
Provides local access to members of the class identified by *hclass* applicable to the data space at *addr*, assuming *addr* is an instance of the class *hclass*. Use of *name* returns the address of this data space.

NAMES must be used inside a definition, between **[OBJECTS** and **OBJECTS]**, and *name* cannot be used outside the definition in which it is defined. No memory is allocated for *name*, and no assumption is made about the persistence of this data space except while this definition is executing.

USING <classname> *(addr — addr)*
Make the members of *classname* available to operate on the dynamic data structure at *addr,* as though *addr* represents an instance of *classname*.

References Local variables, Section 4.5.6
Dynamic memory allocation, *Forth Programmer's Handbook*

9.1.4 Embedded Objects

Previously defined classes may be used as members of other classes. The syntax for using one is the same as for defining static objects. These objects are not static; they will be constructed only when their container is instantiated.

```
CLASS RECTANGLE
    POINT BUILDS UL
    POINT BUILDS LR
    : SHOW ( -- )   UL DOT  LR DOT ;
    : DOT ( -- )   ." Rectangle, " SHOW ;
END-CLASS
```

In this example, the points giving the upper-left and lower-right corners of the rectangle are instantiated as **POINT** objects. The members of **RECTANGLE** may reference them by name, and may use any of the members of **POINT** to manipulate them. In this example, **SHOW** references the **DOT** member of **POINT** to print **UL** and **LR**; this member is *not* the same as the **DOT** member of **RECTANGLE**.

These embedded objects are exactly like data allocations in the class: they simply add their data space to the object's data, and the enclosing object has all of the public members of its embedded objects available in addition to its own.

9.1.5 Information Hiding

Thus far, all named members a class have been visible in any reference to that class or to an object of that class. Even though member names are hidden from casual reference by the user (i.e., to follow from the earlier example in Section 9.1.1, attempting to invoke **X** or **Y** outside the context of an instance of the **POINT** class), the information-hiding requirements of object-oriented programming are more stringent.

In true object-oriented programming, classes must have the ability to hide members from external access. SWOOP accomplishes this using three key words:

- **PUBLIC** identifies members that can be accessed globally.
- **PROTECTED** identifies members available only within the class in which they are defined and in its sub-classes.
- **PRIVATE** identifies members available only within the defining class.

When a class definition begins, all member names default to being **PUBLIC** (i.e., visible outside the class definition). **PRIVATE** or **PROTECTED** changes the level of visibility of the members.

```
CLASS POINT
PRIVATE
    VARIABLE X
    VARIABLE Y
    : SHOW ( -- )   X @ .  Y @ . ;
PUBLIC
    : GET ( -- x y )   X @ Y @ ;
    : PUT ( x y -- )   Y ! X ! ;
    : DOT ( -- )   ." Point at " SHOW ;
END-CLASS
```

In this definition of **POINT**, the members **X**, **Y**, and **SHOW** are now private, available to local use while defining **POINT** but hidden from view afterwards. Because a point is relatively useless unless its location can be set and read, members that can do this

are provided in the public section. However, these definitions achieve the desired information hiding: the actual data storage is unavailable to the user and may only be accessed through the members provided for that purpose.

Glossary

PUBLIC *(—)*

Asserts that following class members will be globally available. This is the default mode. Must be used inside a class definition.

PROTECTED *(—)*

Asserts that following class members will be available only in sub-classes of the current class. Must be used inside a class definition.

PRIVATE *(—)*

Asserts that following class members will be available only within the current class. Must be used inside a class definition.

9.1.6 Inheritance and Polymorphism

Inheritance means the ability to define a new class based on an existing class. The new *subclass* initially has exactly the same members as its parent, but can replace some inherited members or add new ones. If the subclass redefines an existing member, all further use within that subclass will reference the new member; however, all prior references were already bound and continue to reference the previous member.

Polymorphism goes a step further than inheritance. In it, a new class inherits all the members of its parents, but may also redefine any deferred members of its parents. A deferred member is defined like a normal colon method, except the defining word used is **DEFER:** and it is followed by a default behavior. The default behavior will be used whenever no overriding behavior has been defined by a subclass.

For example, our previous example could be written this way:

```
CLASS POINT
    VARIABLE X
    VARIABLE Y
    DEFER: SHOW ( -- )   X @ . Y @ . ;
    : DOT ( -- )   ." Point at " SHOW ;
END-CLASS
```

Then you could make a subclass like this:

```
POINT SUBCLASS LABEL-POINT
    : SHOW ( -- )   ." X" X @ .  ." Y" Y @ . ;
END-CLASS

LABEL-POINT BUILDS POO

POO DOT
```

The original definition **DOT** in the parent class **POINT** will still reference **SHOW**, but

when it is executed for an instance of **LABEL-POINT**, the new behavior will automatically be substituted, so **POO DOT** will print the labeled coordinates.

Glossary

DEFER: <name> *(—)*

Begin defining a deferred member. The content of this definition, following the name, is the default behavior of the member. Subclasses of the class in which *name* is defined may make overriding colon definitions for *name*, which will automatically be substituted for the default behavior for any reference to *name* in either the subclass or members of the parent class that are called in the context of the subclass.

SUBCLASS <name> *(hclass —)*

Begin defining a class which will inherit all members of the superclass identified by *hclass*. In all other respects, its use is the same as that of **CLASS** (Section 9.1.1).

SUPREME *(— hclass)*

Return the handle of the ultimate superclass in SWOOP. All classes defined by **CLASS** are subclasses of **SUPREME**.

9.1.7 Numeric Messages

All the members we've discussed so far are named definitions. Use of a member name returns a member ID which can be matched against members in one of the member chains associated with a class. It is also possible, however, to make members that don't have names and which, instead, are identified by an arbitrary numeric value that in SwiftForth terms is called a *message*. The form of a numeric message definition is:

```
<value> MESSAGE: <words to be executed> ;
```

Whereas named members are invoked by name, numeric messages are dispatched to an object using the word **SENDMSG**, which takes as arguments the handle to an object and the message value.

Glossary

MESSAGE: *(n —)*

Define an unnamed member whose behavior (words following **MESSAGE:**) will be executed when an object containing it is sent the message *n* (which is equivalent to a message ID of *n*).

SENDMSG *(addr n —)*

Execute the member represented by *n*, in the context of the object represented by *addr*. This is equivalent to sending a message to the object. In this case, *n* is equivalent to a member ID, and will be treated as such by the object.

9.1.8 Early and Late Binding

Binding refers to the way a member behaves when referenced; this may be decided at compile time or at run time.

If the decision is made at compile time, it is known as *early binding* and assumes that a specific, known member is being referenced. This provides for simple compilation and for the best performance when executed.

If the decision is made at run time, it is known as *late binding*, which assumes that the member to be referenced is *not* known at compile time and must therefore be looked up at run time. This is slower than early binding because of the run-time lookup, but it is more general. Because of its interactive nature, this behavior parallels the use of the Forth interpreter to reference members.

SWOOP is primarily an early-binding system, but makes several provisions for late binding. The first is deferred members, a technique that parallels the Forth concept of a deferred word. This implements the facet of late binding in which the member name to be referenced is known, but the behavior is not yet determined when the reference is made. The second is the word **SENDMSG** (described in Section 9.1.7), which sends an arbitrary message ID to an arbitrary object.

Of the two strategies for using dynamic objects discussed in Section 9.1.3, **USING** is an example of early binding, whereas **CALLING** and **->** are late binding. The distinction is based on the fact that the class is known at compile time with **USING**, but only at run time with **CALLING**.

9.2 Data Structures

This section will describe the basic data structures involved in classes and members, as a foundation for discussing the more-detailed implementation strategies underlying SWOOP.

9.2.1 Classes

The data representation of a class is shown in Figure 7. Each class is composed of a ten-cell structure. All classes are linked in a single list that originates in the list head **CLASSES**.

Typing **CLASSES** on the command line allows the user to display the hierarchy of all created classes.

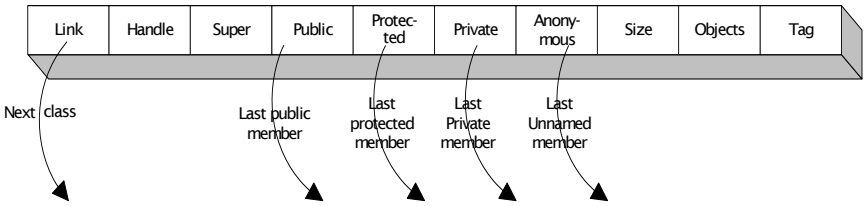

Figure 7. Structure of a class

Each class has a unique handle. When executed, a class name will return this handle. The handle also happens to be the *xt* that is returned by ticking the class name. For example, if **POINT** is a class, then:

```
' POINT .
```

prints the same value as:

```
POINT .
```

Each class (except **SUPREME**) has a superclass. By default, it is **SUPREME**, but a class can be a child of any pre-existing class. The value in the Super field is the handle (*hclass*) of the superclass.

Classes are composed of members, divided into four lists, or *chains*—public, protected, private, and anonymous. The lists are identical in structure and treatment, but differ in their level of information hiding (discussed in Section 9.1.5). Each list has a head in the class data structure. With inheritance, these lists may chain back into the superclass, and into *its* superclass, etc., all the way back to **SUPREME**.

The public, protected, and private lists, in conjunction with the class handle and the wordlist **MEMBERS**, define the class *namespace*.

MEMBERS is a wordlist that contains one entry for every name used in every class. If **X** is defined in multiple classes, there is only one entry for **X** in **MEMBERS**. If you say **ORIGIN X**, then **MEMBERS** becomes part of the search order and **X** is found; its *xt* is then used to search the member lists belonging to the class of which **ORIGIN** is a member, to see if **ORIGIN** has an **X**. If it does, its offset (since it's a data object) is added to the base address of the instance **ORIGIN**. If not, SwiftForth treats it as an ordinary Forth word and searches for it in the remaining available wordlists.)

The anonymous field is used to organize unnamed members (discussed in Section 9.1.7), which make up another list like the three just discussed but which are of primary use for handling Windows message constants and other messages which consist solely of a numerical ID and parameters. Note that there are actually three anonymous chains but they are shown here as a single item for simplicity. Consult the source code for details about these three chains.

The size field represents the size in bytes required by a single instance of the class. This value is the sum of all explicitly referenced data in the class itself, plus the size of its superclass.

When objects with embedded objects (i.e., objects that contain other objects) are **CONSTRUCT**ed, the embedded objects must also be constructed. The list represented

by the object field links all of the objects embedded in a class, so that they can be **CONSTRUCT**ed also.

The class tag is just a constant used to identify the data structure as a valid class.

A class definition is begun by **CLASS** or **SUBCLASS** and is ended by **END-CLASS**. While a class is being defined, the normal Forth interpreter/compiler is used; its behavior is modified by changing the search order to include the class namespace and the wordlist **CC-WORDS** (described in Section 9.3.1).

All links in this system are relative, and all handles are execution tokens. This means that objects created in the interactive system at a given address will work when loaded at a base address different from where it was originally compiled.

9.2.2 Members

Members are defined between **CLASS** and **END-CLASS**. They parallel the basic Forth constructs of variables, colon definitions, and deferred words. The definition of a member has two parts. First is the member's name, which exists in the wordlist **MEMBERS**. The *xt* of this name is used as the member ID when it is referenced. Second is the member's data structure. This contains information about how to compile and execute the member. Each member is of the general format shown in Figure 8; the specific format of some member types is shown in Figure 9.

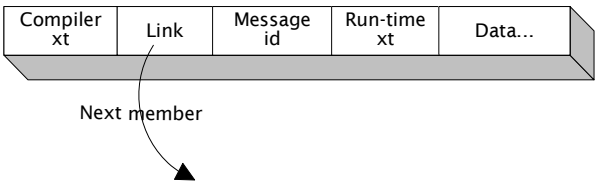

Figure 8. Basic structure of a member

The data structure associated with a member has five fields: member compiler, link, message ID, member run time, and data. The data field is not of fixed length; its content depends on the programmatic expectations of the compiler and run-time routines.

The *compiler-xt* is the early-binding behavior for members, and the *runtime-xt* is the late-binding behavior. Each variety of member has a unique *compiler-xt* and *run-time-xt*; both expect the address of the member's data field on the stack when executed. The *message ID* in each entry is the *xt* given by the member's name in the

MEMBERS wordlist.

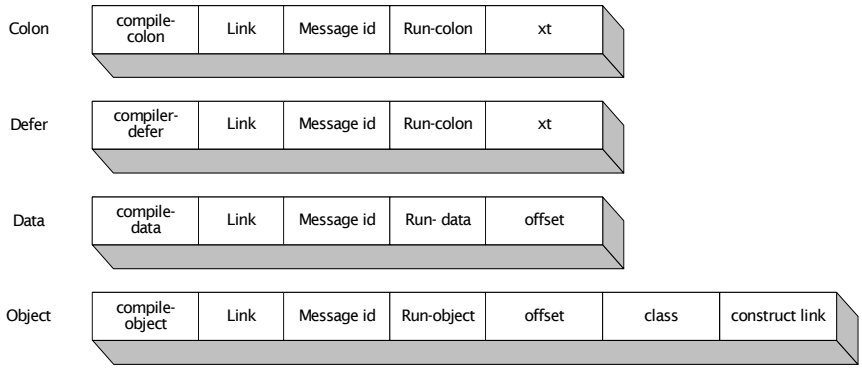

Figure 9. Data structures for various member types

The data field contents vary depending on what type of member the structure represents. For data members, the data field contains the offset in the current object. For colon members, it contains the *xt* that will be executed to perform the actions defined for the member. In deferred members, the data field also contains an *xt*, but it is only used if the defer is not extended beyond its default behavior. In object members, the data field contains both the offset in the current object of the member and the class handle of the member.

9.2.3 Instance Structures

The structure of an instantiated object is largely dependent on the members defined in the object. To a great extent, members can be applied to arbitrary memory addresses. However, formal instantiation does add useful information.

Static objects, instantiated by **BUILDS**, have the handle (*xt*) of the object and handle (*hclass*) of the class in the first two cells. Dynamic objects, instantiated by **NEW** or **MAKES**, have only a class handle in the first cell. The object handle of a static object is primarily used at compile time. The class handle, however, is used at run time to ensure the validity of a member being applied to an object (i.e., the member's *xt* must appear in one of the visible chains, e.g., **PUBLIC**, of the class of which it is an instance). For this reason, it is better to use formal instantiation and apply members with **CALLING** or **->**, even though you can apply any class's members to any arbitrary address (e.g., **PAD**, **HERE**) with **USING**.

9.3 Implementation Strategies

Having discussed the basic syntax and data structures involved in SWOOP, we can now consider the underlying mechanisms in the system.

9.3.1 Global State Information

In some OOP implementations, classes are composed of instance data, methods that can act on the data, and messages corresponding to these methods that can be sent to objects derived from the class.

In SWOOP, instance data and methods are combined into a single orthogonal concept: members. Each member has a unique identifier which can be used as a message. Members exist as created *names* in a special wordlist called **MEMBERS**; each member's *xt* is its identifier. A given name will exist only once in **MEMBERS**; a member name always corresponds to the same identifier (i.e., *xt*) regardless of the class or context in which it is referenced. (See also Section 9.2.1.)

A second special wordlist called **CC-WORDS** contains the compiler words used to construct the definitions of the members of classes.

Classes are composed of members organized in the public, protected, and private lists. The structure of a class is shown in Figure 7. The member lists of a class are based on switches (see Section 4.5.4) and use a member identifier as a key. A class doesn't know the names of its members, only their identifiers.

SWOOP depends on two variables for its behavior during compilation and execution. **'THIS** contains the handle of the active class, and **'SELF** has the active object's data address. These are *user variables*, so object code is re-entrant.

Glossary

MEMBERS *(—)*

Select the wordlist containing all members of all classes. A defined member name has a unique entry in this list, even though it may have different definitions in different classes. The entry in the **MEMBERS** wordlist returns an identifier that can be sought in the list of defined members in the current class; if a match is found, it will link to the appropriate definition for that class.

This wordlist is automatically added to the search order whenever a class name is invoked.

CC-WORDS *(—)*

Select the wordlist containing compiling words used to construct member definitions. This wordlist is automatically added to the search order between **CLASS** or **SUBCLASS** and **END-CLASS**, to define members of the class.

'THIS *(— addr)*

Return the address containing the handle of the current class. This provides access to its members.

'SELF *(— addr)*

Return the address containing a pointer to the data space of the current class.

CSTATE *(— addr)*

Return the address of this variable. During definition of class members, it contains the handle of the class being defined. At all other times, it is zero.

References Search orders, Section 5.5.2
User variables, ???

9.3.2 Compilation Strategy

A class's *namespace* is defined by all words in the **MEMBERS** wordlist whose handles match keys in the class's lists of members.

The executable definitions associated with entries in **MEMBERS** are immediate. When **MEMBERS** is part of the search order, a reference to a member may be found there, and it will be executed. When executed, it will search for a match on its handle in the list of keys in the member lists for the current class (identified by **'THIS**). If a match is found, the compilation or execution *xt* associated with the matching member will be executed, depending on **STATE**. If there is no match in the current class, the name will be re-asserted in the input stream and the Forth interpreter will be invoked to search for it in other wordlists, handling it subsequently in normal fashion.

During compilation of a class, certain of the normal Forth defining words are superseded by SWOOP-specific versions in a wordlist called **CC-WORDS**. These member-defining words are only present in the search order while compiling a class—that is, between **CLASS** or **SUBCLASS** and the terminating **END-CLASS**.

The simplest way to discuss the compiler is to walk through its operation as a class is built. So, we define a simple class:

```
CLASS POINT
    VARIABLE X
    VARIABLE Y
    : DOT ( -- )   X @ .  Y @ . ;
END-CLASS
```

The phrase **CLASS POINT** creates a class data structure named **POINT**, links it into the **CLASSES** list, adds **CC-WORDS** and **MEMBERS** to the search order, and sets **'THIS** and **CSTATE** to the handle of **POINT**. The variable **CSTATE** contains the handle of the current class being defined, and remains non-zero until **END-CLASS** is encountered. This is used by the various member compilers to decide what member references mean, and how to compile them.

VARIABLE X (and, likewise, **Y**) executes the member-defining word **VARIABLE** in **CC-WORDS**, which adds a member name to **MEMBERS** and to the chain of public members for **POINT**.

Although the colon definition **DOT** looks like a normal Forth definition, its critical components **:** and **;** are highly specialized in the **CC-WORDS** wordlist. This version of **:** searches for the name **DOT** in the **MEMBERS** wordlist; if there is already one, it uses its handle as the message ID for the member being defined. Otherwise, it constructs a name in **MEMBERS** (rather than with the class definitions being built), keeping its handle. Then it begins a **:NONAME** definition, which is terminated by the **;**. This version of **;** not only completes the definition, it uses its *xt* along with the message ID to construct the entry in the appropriate members chain for **DOT**.

When a class member is referenced (such as in the reference to **X** in **DOT**), its compiler method is executed. The compiler method constructs the appropriate kind of reference to the member (e.g., via **COLON-METHOD** or **DATA-METHOD**).

9.3.3 Self

Notice that we have seemingly inconsistent use of our members. While defining **POINT**, we can simply reference **X**; while not defining **POINT**, we must reference an object prior to **X**. This problem is resolved in some systems by requiring the word **SELF** to appear as an object proxy during the definition of the class.

```
: DOT ( -- )   SELF X @ .  SELF Y @ . ;
```

This results in a more consistent syntax, but is wordy and repetitive. However, to the compiler, the reference to **X** is *not* ambiguous, so the explicit reference to **SELF** is unnecessary. While a class is being defined, SWOOP notices that **X** (or any other member) is indeed a reference to a member of the class being defined and *automatically* inserts **SELF** before the reference is compiled. This results in a simpler presentation of the routine, and makes the code inside a class look like it would if were not part of a class definition at all.

9.4 Tools

DUMP-OBJECT is provided as a method of the **SUPREME** class (the super class from which all other classes inherit the original methods and data). You can invoke this dump method to show the entire data area of an object like this:

```
<object> DUMP-OBJECT
```

SUPREME also supplies a zero-length data object named **ADDR**, which has the effect of returning the start address of the data area of an object.

If you wish to dump memory in only part of an object's data area, use **ADDR** to get the address followed by one of the standard SwiftForth memory dump words:

```
<object> ADDR <n> DUMP
```

Reference Static memory dumps, Section 2.4.5.1

SECTION 10: FLOATING-POINT MATH LIBRARY

SwiftForth's floating-point math library is a system option providing support for the Intel Architecture Floating-Point Unit (FPU).

10.1 The Intel FPU

The Intel Architecture Floating-Point Unit (FPU) provides high-performance floating-point processing capabilities. It supports the real, integer, and BCD-integer data types and the floating-point processing algorithms and exception-handling architecture defined in the IEEE 754 and 854 Standards for Floating-Point Arithmetic. The FPU executes instructions from the processor's normal instruction stream and greatly improves the efficiency of Intel architecture processors in handling the types of high-precision, floating-point operations commonly found in scientific, engineering, and business applications.

Support for the FPU adds an important computational dimension to SwiftForth. The SwiftForth implementation makes use of the FPU's 80-bit wide by eight deep hardware stack only during primitive operations. Each task has its own floating-point stack, which is **#NS** bytes long and is located at the bottom of a task's data space (below **HERE**). **#NS** is sized for 32 80-bit stack elements. A task's numeric stack is referred to as *the numeric stack* or *N-stack*; the numeric stack internal to the FPU is referred to as the *hardware stack*.

Floating-point primitives transfer the number of stack items they require to the hardware stack and return the result to the numeric stack when finished. Floating-point numbers are converted directly on the hardware stack, extending input conversion to a full 80 bits. Integers and double-precision integers are converted, as always, on the data stack. The primary goals in the development of the FPU co-processor support package were high throughput, precision, and code compatibility with standard integer SwiftForth.

10.2 Use of the Math Co-processor Option

Operation of the FPU requires only that its file **fpmath.f** be loaded. You may **INCLUDE** it directly, or load it using **REQUIRES fpmath**. If this file is included, the FPU will be initialized as part of the system initialization.

The set of user-level words in this option is fully compliant with ANS Forth, and all the documentation on floating point in *Forth Programmer's Handbook* applies; material discussed in that section is not repeated here. The default length of floating-point memory operations (e.g., **F@**, **F!**, etc.) is 64 bits, the same as the IEEE long floating-point format. This option includes an extended set of commands not discussed in *Forth Programmer's Handbook*, as well as features particular to the FPU/80486 processor; these additional commands and features are described in the following sections.

10.2.1 Configuring the Floating-Point Options

TBD

10.2.2 Input Number Conversion

The text interpreter in SwiftForth handles floating-point numbers as specified by ANS Forth. They 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.*

SwiftForth will only attempt to convert a number in floating point if **BASE** is decimal. If it is not, or if the number is not a valid floating-point number, then the default number-conversion rules described in Section 4.3 apply.

The conversion of floating number strings is performed by the word **(REAL)**. **(REAL)** performs the conversion directly on the hardware stack of the co-processor to simplify the task and to extend the accuracy to the full 80 bits.

This system also supports the ANS extended conversion word **>FLOAT**. **>FLOAT** is more general than the text interpreter and will convert nearly any reasonably constructed number. See the *Handbook* for details.

References Input number conversions in SwiftForth, Section 4.3

10.2.3 Output Formats

Some additional output words are provided for the display of floating-point numbers beyond those described in *Forth Programmer's Handbook*. The phrases:

```
<n> F.R
<n> FS.R
```

display the top item on the numeric stack right-justified field of *n* characters in **F.** and **FS.** formats.

If all the significant digits are too far behind the decimal point, the displayed number will be all zeroes. If the digits to the left of the decimal won't fit in the space, they will be printed regardless.

The programmable display word **N.** allows you to select the output format and number of significant digits at run time. **N.** is useful as a general-purpose numeric stack output word, and finds its greatest utility in compiled definitions in which the output format can be set at run time before executing the definition.

Glossary

F.R *(n —); (F: r —)*
Display *r* in the **F.** output format, right-justified in a field *n* characters wide.

FS.R *(n —); (F: r —)*
Display *r* in the **FS.** output format, right-justified in a field *n* characters wide.

N. *(—); (F: r —)*
Display *r* in a format selected by **FIX**, **SCI**, or **ENG**.

FIX *(n —)*
Configure **N.** to use the **F.** output format with *n* significant digits.

SCI *(n —)*
Configure **N.** to use the **FS.** output format with *n* significant digits.

ENG *(n —)*
Configure **N.** to use the **FE.** output format with *n* significant digits.

SET-PRECISION *(n —)*
Set the default output precision for the above formatting words to *n*.

10.2.4 Real Literals

Floating-point literals may be compiled in one of four data types: integer, double integer, short, and long. Words for managing them are described in the glossary below.

Glossary

DFLITERAL *(—); (F: r —)*
Compile the number on the floating-point stack as a 64-bit floating-point literal.

FLITERAL *(—); (F: r —)*
Same as **DFLITERAL**.

SFLITERAL *(—); (F: r —)*
Compile the number on the floating-point stack as a 32-bit floating-point literal.

FILITERAL *(—); (F: r —)*
Compile the number on the floating-point stack as a 32-bit or 64-bit two's complement rounded integer. A double-length (64-bit) integer is compiled if the number exceeds 32 bits.

10.2.5 Floating-Point Constants and Variables

In addition to the defining words **FCONSTANT** and **FVARIABLE** in *Forth Programmer's Handbook*, which are implemented here as 64-bit quantities, this system provides

the corresponding IEEE standard format words **SFCONSTANT** (32 bits), **DFCONSTANT** (64 bits), **SFVARIABLE** (32 bits), and **DFVARIABLE** (64 bits). In this system, **FCONSTANT** and **DFCONSTANT** are identical, as are **FVARIABLE** and **DFVARIABLE**.

Although the FPU co-processor also supports 80-bit floating and 80-bit packed BCD data types, these are usually found in specific applications and are not supported in this option. The programmer may easily incorporate these functions through simple **CODE** definitions.

Glossary

SFCONSTANT <name> *(—); (F: r —)*
Define a floating-point constant with the given *name* whose value is *r*, compiled in short (32-bit) format. When *name* is executed, *r* will be returned on the floating-point stack.

DFCONSTANT <name> *(—); (F: r —)*
Define a floating-point constant with the given *name* whose value is *r*, compiled in double (64-bit) format. When *name* is executed, *r* will be returned on the floating-point stack.

SFVARIABLE <name> *(—)*
Define a floating-point variable with the given *name*, allocating space to store values in short (32-bit) format. When *name* is executed, the address of its data space will be returned on the data stack.

DFVARIABLE <name> *(—)*
Define a floating-point variable with the given *name*, allocating space to store values in double (64-bit) format. When *name* is executed, the address of its data space will be returned on the data stack.

10.2.6 Memory Access

Memory access words similar to those in standard SwiftForth are provided for floating-point data types. These obtain addresses from the data stack and transfer data to and from the numeric stack.

In addition to the words documented in *Forth Programmer's Handbook,* the following are provided:

Glossary

F+! *(addr —) (F: r —)*
Add the top floating-point stack value to the 64-bit contents of the address on the data stack. "Floating plus store"

DF+! *(addr —) (F: r —)*
Same as **F+!**.

SF+! *(addr —) (F: r —)*
The 32-bit equivalent of **F+!**. "Short floating plus store"

F, *(—) (F: r —)*

Compile the top 64-bit floating-point stack value into the dictionary. "Floating comma"

FL, *(—) (F: r —)*

Same as **F,**.

FS, *(—) (F: r —)*

The 32-bit equivalent of **F,**. "Floating short comma"

Integer Transfers To and From the Numeric Stack

FI@ *(addr —) (F: — r)*

Push on the floating-point stack the 64-bit data specified by the address on the data stack. "Integer fetch"

SFI@ *(addr —) (F: — r)*

Push on the floating-point stack the 32-bit data specified by the address on the data stack. "Single fetch"

DFI@ *(addr —) (F: — r)*

On this system, the same as **FI@**. "Double fetch"

FI! *(addr —) (F: r —)*

Store the top floating-point stack item, rounded to a 64-bit integer, in the address on the data stack. "Integer store"

DFI! *(addr —) (F: r —)*

On this system, the same as **FI!**. "Double store"

SFI! *(addr —) (F: r —)*

Store the top floating-point stack item, rounded to a 32-bit integer, in the address given on the data stack. "Single store"

FI, *(—) (F: r —)*

Convert the top floating-point stack value to a rounded 64-bit integer and compile it into the dictionary. "Integer comma"

DFI, *(—) (F: r —)*

On this system, the same as **FI,**. "Double comma"

SFI, *(—) (F: r —)*

Convert the top floating-point stack value to a rounded 32-bit integer and compile it into the dictionary. "Single comma"

10.2.6.1 Stack Operators

The SwiftForth floating-point option contains the words **MAKE-FLOOR** and **MAKE-ROUND**. These allow you to control whether truncation or rounding takes place when transferring numbers from the floating-point stack to the integer data stack. Compliance with ANS Forth requires truncation, so the floating-point option executes **MAKE-FLOOR** when it is loaded.

These words are supplied in addition to operators documented in *Forth Programmer's Handbook*.

Glossary

F2DUP *(—) (F: r_1 r_2 — r_1 r_2 r_1 r_2)*

Duplicate the top two floating-point stack items.

F?DUP *(— flag) (F: r — r |)*

Test the top of the floating-point stack for non-zero. If the number is non-zero, it is left and a *true* value is placed on the data stack; if the number is zero, it is popped and a zero (*false*) is placed on the data stack.

FWITHIN *(— flag) (F: r l h)*

Return a *true* value on the data stack if the floating-point value *r* lies between the floating-point values *l* and *h*, otherwise return *false*.

/FSTACK *(—) (F: i*r —)*

Clear the numeric stack.

?FSTACK *(—) (F: i*r — i*r)*

Check the numeric stack and abort if there are no numbers on it.

MAKE-FLOOR *(—)*

Configure SwiftForth to use truncation when transferring numbers from the floating-point stack to the data stack. This is the ANS Forth convention, and is the default in SwiftForth.

MAKE-ROUND *(—)*

Configure SwiftForth to use rounding when transferring numbers from the floating-point stack to the data stack. This is the FPU convention.

S>F *(n —) (F: — r)*

Remove a 32-bit value from the data stack and push it on the floating-point stack.

F>S *(— n) (F: r —)*

Remove the top floating-point stack value, round it to a 32-bit integer, and push it on the data stack.

D>F *(d —) (F: — r)*

Remove a 64-bit value from the data stack and push it on the floating-point stack.

F>D *(— d) (F: r —)*

Remove the top floating-point stack value, round it to a 64-bit integer, and push it on the data stack.

F2* *(—) (F: r_1 — r_2)*

Multiply the top floating-point stack value by 2.

F2/ *(—) (F: r_1 — r_2)*

Divide the top floating-point stack value by 2.

1/N *(—) (F: r_1 — r_2)*

Replace the top floating-point stack value with its reciprocal value.

SIN *(—) (F: x — r)*
Return the sine of *x*, where *x* is in degrees.

COS *(—) (F: x — r)*
Return the cosine of *x*, where *x* is in degrees.

TAN *(—) (F: x — r)*
Return the tangent of *x*, where *x* is in degrees.

COT *(—) (F: x — r)*
Return the cotangent of *x*, where *x* is in degrees.

SEC *(—) (F: x — r)*
Return the secant of *x*, where *x* is in degrees.

CSC *(—) (F: x — r)*
Return the cosecant of *x*, where *x* is in degrees.

10.2.6.2 Matrix-Defining Words

Two-dimensional matrix data structures are supported by the floating-point math option. A matrix is constructed as a standard dictionary entry, with the number of bytes per row stored in the first cell of the parameter field. The first matrix storage location follows the count and, thus, is located at `<parameter field address CELL+>`. Matrix storage locations are arranged with column indices varying more rapidly then row indices. The relative address of any element is thus:

```
<row index> × <# of bytes per row> + <column index>
```

Matrices for 32-bit and 64-bit floating-point data types are constructed and displayed with the words in the glossary at the end of this section.

When words defined by **SMATRIX** and **LMATRIX** are referenced by name, each returns the memory address of the specified matrix row and column. Subscripts range from zero to one less than the declared row or column sizes. For example, if you define:

```
3 4 SMATRIX DATA
```

you have a matrix whose name is **DATA**, with three rows of four columns each. The phrase **0 0 DATA** returns the address of the first value, and **2 3 DATA** returns the address of the last value. No subscript range checking is performed.

Glossary

SMATRIX <name> *(nr nc —)*
Construct a matrix containing space for *nr* rows and *nc* columns, with 32 bits per entry.

LMATRIX <name> *(nr nc —)*
Construct a similar matrix with 64-bit storage locations.

SMD <name> *(nr nc —)*
Display a previously defined short (32-bit entries) matrix. The number of rows and

columns must agree with the number in the definition.

LMD <name> *(nr nc —)*
Similar to **SMD** but displays a long (64-bit entries) matrix.

10.3 FPU Assembler

The FPU assembler extends the instruction set with floating-point instructions. Most FPU instructions are implemented. The memory reference addressing modes are a subset of the CPU modes, because the FPU relies on the CPU to generate the address of memory operands. This documentation is intended as a supplement to the SwiftForth i386 assembler; see Section 10.3.3.3 for further references.

References i386 assembler, Section Section 6:

10.3.1 FPU Hardware Stack

The FPU assembler instructions work on the FPU's internal hardware stack and are not directly connected with the task-specific numeric stacks implemented in SwiftForth. Thus, before the FPU instructions can work on numeric stack items, the items must be transferred to the hardware stack. Likewise, the results must be returned to a task's numeric stack after an instruction has been completed.

To facilitate this process, four macros assemble the necessary FPU instructions to perform these transfers. They are listed in the glossary below.

Either **>f** or **>fs** should be used at the beginning of floating-point primitives, and **f>** or **fs>** should be used at the end.

Glossary

>f *(—)*
Assemble FPU instructions to transfer one numeric stack item to the hardware stack.

f> *(—)*
Assemble FPU instructions to transfer one hardware stack item to the numeric stack.

>fs *(n —)*
Assemble FPU instructions to transfer *n* numeric stack items to the hardware stack. Stack order is preserved.

fs> *(n —)*
Assemble FPU instructions to transfer *n* hardware stack items to the numeric stack. Stack order is preserved.

10.3.2 CPU Synchronization

The FPU assembler functions as part of the SwiftForth 80386 assembler. FPU instructions are assembled in the same manner and format.

For maximum throughput, FPU instructions are not synchronized with the 80386 unless specifically coded. When a CPU memory reference instruction that operates on data to or from the FPU is needed, you should explicitly code a **WAIT** instruction to synchronize the transfer.

All **CODE** words that use FPU instructions should end with **FNEXT**.

10.3.3 Addressing Modes

The FPU employs two types of addressing modes:

- memory reference
- hardware stack top relative

Memory reference modes use the 80386 operand formats, whereas the hardware stack top relative mode is unique to the FPU.

10.3.3.1 Memory Reference

The FPU uses the i386 operand addressing modes to transfer data to and from memory. There are three valid operand types:

- Register indirect
- Register indexed with displacement
- Direct

Register indirect and register indexed with displacement use the same format as the SwiftForth i386 assembler; see Section 6.3 for SwiftForth register usage. The direct addressing mode specifies an absolute memory address.

Some FPU opcodes require a memory format specifier to select data size and type. These specifiers are listed in Table 11.

Table 11: Memory-format specifiers

Format Code	Meaning
WORD	16-bit integer
DWORD	32-bit integer or floating
QWORD	64-bit floating

The format specifier must precede the addressing operand of the FPU instruction. Here's an example of a FPU memory reference:

```
CODE SF@ ( a -- ) ( -- r )       \ Fetch real from addr
```

```
0 [EBX] DWORD FLD            \ Load 32-bit real from addr
0 [EBP] EBX MOV  4 # EBP ADD \ Pop data stack
f> FNEXT                     \ Real to local FP stack
```

10.3.3.2 Stack Top Relative Addressing

FPU hardware stack operations that deal with two stack parameters have an operand format that specifies the location of the item relative to the top of the numeric stack. **ST(0)** references the top stack item, **ST(1)** references the second item, etc. Some examples of stack addressing are given in Table 12.

Table 12: Examples of FPU stack addressing

Command	Action
ST(1) FXCH	Equivalent code for hardware stack **SWAP**.
ST(0) FLD	Equivalent code for hardware stack **DUP**.
ST(1) FLD	Equivalent code for hardware stack **OVER**.
ST(0) FSTP	Equivalent code for hardware stack **DROP**.

10.3.3.3 FPU References

For more information concerning the details of the operation and accuracy of the FPU, consult the Intel 64 and IA-32 Architectures Software Developer's Manuals. Links to these manuals can be found on the *SwiftForth* page of www.forth.com.

SECTION 11: RECOMPILING SWIFTFORTH

There are two binary program files included in the SwiftForth distribution; both are installed in the SwiftForth/bin directory:

- **sfk** — The SwiftForth kernel with no extensions loaded. This is a "bare-bones" console application.
- **sf** — The extended SwiftForth interactive development environment. Includes many features beyond the basic kernel: debug tools, dynamic library interface, SWOOP, and turnkey program generator, just to name a few. The **sf** binary is itself a turnkey SwiftForth application.

The complete source code for all these components is supplied with the full SwiftForth distribution. The following sections detail how to recompile part or all of the SwiftForth binaries.

11.1 Recompiling the SwiftForth Turnkey

If you are generating a turnkey version of your own application, please refer to Section 4.1.2 for simple instructions on preparing a turnkey image.

Launch the SwiftForth kernel program, **SwiftForth/bin/sfk**. Then compile your electives. This is most easily done by typing the **HI** command. **HI** looks for the source file **hi.f** first in the current working directory, then in the default location, **SwiftForth/src/ide/<os>**.

If you are generating a customized interactive development environment, you should make your customizations and load them from a "local" **hi.f** in your own directory, away from the SwiftForth-installed file hierarchy. Use these local copies to avoid losing your changes when you update to a new release of SwiftForth.

After loading the electives, save a new turnkey with the **PROGRAM** command as detailed in Section 4.1.2.

11.2 Recompiling the Kernel

The full source to the SwiftForth kernel is supplied in the directory **SwiftForth/src/kernel**. The target compiler used to generate **sfk** is in **SwiftForth/xcomp.** To recompile the kernel, launch the full **sf** turnkey (**sfk** by itself does not have enough extensions loaded to support the target compiler) with the working directory set to **SwiftForth/src/kernel/<os>**, then include the "make" file:

```
INCLUDE make
```

The output will be saved in **SwiftForth/bin/<os>/sfk** and the SwiftForth turnkey will exit. You can change the output file name and destination directory by editing the source file **make.f** included above.

SwiftForth is a copyrighted work and FORTH, Inc. reserves all rights to its use. Please do not modify the copyright notice displayed by **sf** or **sfk**.

APPENDIX A: MANAGING BLOCK FILES

Early Forth systems ran in "native" mode on computers, meaning there was no operating system other than Forth. These systems organized disk in "blocks" of 1024 bytes each. Blocks were mapped to physical addresses on the disk drive, so there was no disk directory. These systems were extremely fast and reliable. However, as personal computers became popular in the 1980's, many users came to prefer versions of Forth that ran co-resident under general-purpos operating systems such as MS-DOS. In order to maintain portability between native and OS-based Forths, the concept of block-oriented disk access was preserved, but blocks resided in OS files. Nowadays block-oriented Forths are sufficiently rare that most Forths (including SwiftForth) have elected to manage disk only as OS files, without the block layer.

To maintain portability between SwiftForth and block-based Forths, optional support for block files is provided, along with a block editor and block-oriented source-management tools, described here. Basic principles of block handling are described in *Forth Programmer's Handbook*.

A.1 MANAGING DISK BLOCKS

To load the full set of block-handling features, use **REQUIRES blockedit**. You can also use **REQUIRES blocks** to load block support without the block editor.

In SwiftForth, Forth blocks are kept in host sytem files and are accessed through the operating system. Multiple files may be open at once; the blocks occupying a single file are referred to as a numbered *part* (of all accessible blocks).

The correspondence between Forth block numbers and host files is called the *blockmap* and is maintained in an array named **PARTS**. For each file, the blockmap contains:

- an index to the filename
- the file handle
- access mode (read/write or read-only)
- starting (absolute) block number for the file
- number of blocks in the file

The system as shipped is capable of mapping 256 files into the blockmap. The word **CHART** takes care of the assignment of block numbers to files. Files with the extension **.src** are assumed to contain source and shadow blocks.

PART sets the value of **OFFSET** equal to the starting block number of the given part and makes the given part current. For example, the phrase:

```
2 PART 1 LIST
```

will display Block 1 of the file mapped to Part 2. Note, however, that **PART** numbers have absolutely no relationship to block numbers unless you enforce such a rela-

tionship. Otherwise you must treat them as arbitrary handles (as does the system). The first file mapped is in Part 0.

The entries in the blockmap are shown in Table 13. Each item occupies one cell.

Table 13: Blockmap format

Name	Description
#BLOCK	Starting block number (-1 if unused part).
#LIMIT	Ending block number + 1.
FILE-HANDLE	File handle.
FILE-UPDATED	Flag: set to 1 when file is **UPDATE**d (written) but not **FLUSH**ed.
FILE-MODE	File access mode (R/O, R/W).
#FILENAME	Index into the **'FILENAMES** array of pointers to counted filename strings.

The command <n> **>PART** where *n* is the part number, ranging from 0 to the number of parts minus one, will vector these names to refer to the information for the given numbered part.

To change the blockmap, there are a number of words for mapping and unmapping parts. The word **MAPS** (or **MAPS"** inside a definition) takes an unused **PART** number, reads a previously created filename, and sets up the mapping data for opening that file. **UNMAP** takes a part number and unmaps the corresponding file, **FLUSH**es, marks the part unused, and closes the file. The access words **READONLY** and **MODIFY** each take the starting block number of the file. They open the file with read or modify access, and finally enter the block offset, completing the blockmap entry. For example:

```
2 MAPS /user/appl.src 1200 MODIFY
```

Note that the filename can be an unambiguous filename or a full pathname. The above example assumes you have created a file called **appl.src** in the **user** subdirectory.

To review the mapping of files to parts and block numbers, use the word **.MAP**. It displays a table with one entry for each part. Each entry indicates the starting block number, access type, file size, and filename for the part.

New block files can be created and mapped with **NEWFILE**, which expects the number of blocks to be created in a new file. The file is created and then opened for modifying. If a filename extension is given, it will be used; if it is omitted, the extension **.src** will be added. For example:

```
60 NEWFILE myblocks
```

creates a new file called **myblocks.src** containing 60 Forth blocks that will be mapped starting at the next available block in the parts map.

The filename created by **NEWFILE** is compiled into the dictionary and, thus, a subsequent **EMPTY** (such as that executed by loading a utility—see Section 4.1.1) will remove the file from the blockmap unless there has been an intervening **GILD**.

The most common words used with disk block mapping are given in the glossary below.

Glossary

PART *(n —)*

Set **OFFSET** to the start of part *n*.

MAPS <name> *(n —)*

Given a free part number and a file *name* in the input stream, map the file. For example:

```
UNMAPPED MAPS forth.src
```

MAPS" <name>**"** *(n —)*

Same as **MAPS**, but used inside a colon definition.

USING <name> *(— n)*

Take a file *name*, and return the part number where that file is mapped. **CHART**s the file if it is not found in the current block map.

CHART <name> *(—)*

Map the file *name* into the lowest free part. When used interpretively, does a **.MAP**. If the file extension is omitted, a **.src** extension is assumed; you must include the dot (.) in the filename to override this feature.

.PARTS *(—)*

Display the file-mapping information.

.MAP *(—)*

Display the file-mapping information and reset the screen scrolling region to the top of the screen in order to display a large number of files.

UNMAP *(n —)*

Take a part number and remove it from the blockmap.

UNMAPS *(n_1 n_2 —)*

Take a range of part numbers, starting with n_1 and ending with n_2, and remove them from the blockmap.

-LOAD *(n —)*

Take a block number, load the block, and then unmap the file it is in, unless the file is in **0 PART**, in which case it is retained.

NEWFILE <name> *(n —)*

Take a file *name* and create a file with *n* blocks. Print the starting block and mapping part number. If the file extension is omitted, a **.src** extension is assumed. You must include the dot (.) in the filename to override this feature.

FLUSH *(—)*

Write all updated buffers to disk.

READONLY *(n —)*

Given a relative starting block number, open the file for reading.

MODIFY *(n —)*
Given a relative starting block number, open the file for reading and writing.

LOADUSING <name> *(n —)*
Take a file *name*, open the file for reading, and **LOAD** block *n*. If the file extension is omitted, a **.src** extension is assumed; you must include the dot (.) in the filename to override this feature.

-MAPPED *(addr n — f)*
If the ASCII filename at address *addr* and of length *n* is not in the current parts map, return *true*. Otherwise, return *false* and select the part where this file is mapped.

UNMAPPED *(— n)*
Return the number of the lowest free **PART**.

AFTER *(— n)*
Return the lowest block number beyond all mapped blocks. Also defined in lower case (**after**).

?PART *(n — n_1)*
Take an absolute block number and return the number of the **PART** which maps it. This is useful in **UNMAP**ping a file whose **PART** number is unknown.

A.2 Source Block Editor

This system provides a resident string editor for editing source kept in blocks. Before any of the commands described in this section may be issued, it is necessary to select a program block for editing. Then, to obtain access to the string editor vocabulary, type **EDITOR**. Note that, for convenience, **T** and **L** (described below) are in the vocabulary **FORTH**—use of either of these will automatically select **EDITOR** for you.

References Vocabularies, Section 5.5.2

A.2.1 Block Display

To display a block and at the same time select it for future editing, type:

```
<n> LIST
```

where *n* is the logical block number of the desired block.

Once selected, the current program block may be (re)displayed (and the **EDITOR** selected) by the following command:

```
L
```

The number of the block is displayed on the first line; the block is displayed below it, formatted as sixteen lines of 64 ASCII characters.

Each line is numbered on the left side as 0–15 or 0–F, depending on whether the user is currently in **DECIMAL** or **HEX** base. These line numbers are not stored with the text but are added to the display for easy reference.

The characters ok will appear at the end of the final line of the block, indicating that the display is complete and that Forth is ready for another command. Regardless of the position in which they appear in the display, these characters do not appear in the actual text of the block.

The current block number is kept in the user variable **SCR**. **SCR** is set by **LIST** or **LOCATE**; all editing commands operate on the block specified by **SCR**.

Before a program block has been used, it contains data of an undefined nature. The command **WIPE** will fill the block with ASCII spaces. The block is considered unused if the entire first line (first 64 characters) of the block are all ASCII spaces, so *when editing a block do not leave this line entirely blank*.

For convenience, three additional block display commands exist: **N**, **B**, and **Q**. **N** (Next) adds 1 to **SCR**, then displays the next block. **B** (Back) subtracts 1 from **SCR**, to display the previous block. **Q** adds or subtracts the current documentation shadow block offset into **SCR**, so typing **Q** alternates the block display between a source block and its documentation shadow block.

A.2.2 String Buffer Management

The string **EDITOR** contains two buffers used by most of the editing commands. These are called the *find buffer* and the *insert buffer*. The find buffer is used to save the string that was searched for most recently by one of the three character-editing commands **F**, **D**, or **TILL**; it is at least sixty-four characters in length. The insert buffer is also at least sixty-four characters long; it contains the string that was most recently inserted into a line by the character-editing commands **I** or **R**, or the line most recently inserted or deleted by the line editing commands **X**, **U**, or **P**. The command **K** interchanges the contents of the find and insert buffers, without affecting the text in the block.

The existence of these buffers allows multiple commands to work with the same string; understanding which commands use which buffers will enable you to use the **EDITOR** more economically. The convention is this: commands that may accept a string as input will expect to be followed immediately either by a space (the command's delimiter) and one or more additional characters followed by a carriage return, or by a carriage return only.

In the former case, the string will be used and will also be placed in the string buffer that corresponds to the command (find or insert). In the latter case (carriage return only), the string that is currently in the appropriate buffer will be used (and will remain unchanged).

For example, the character-editing command:

```
F WORDS TO FIND
```

will place **WORDS TO FIND** in the find buffer and will find the next occurrence of the string **WORDS TO FIND**. Subsequent use of the command **F** immediately followed by a carriage return will find the next occurrence of **WORDS TO FIND**. Table 14 summarizes buffer usage.

Table 14: Block-editor commands and string buffer use

Find Buffer	Insert Buffer
F	**I**
S	**R**
D	**X**
TILL	**U**
K	**P**
	K

A.2.3 Line Display

Any single line of the current block (whose number is in **SCR**) may be selected by using the following command:

```
<n> T
```

where *n* (which must be in the range 0–15) is the line number to be selected.

The **T** (Type) command sets the user variable **CHR** to the character position of the beginning of the line. This value may later be used to identify the line to be changed, using the commands defined in the following section. Since **CHR** is used to store the cursor position for the character-editing commands, using **T** (i.e., initializing **CHR**) specifies that any search will start at the beginning of that line. The new cursor position is marked in some convenient way.

The contents of the string buffers and of the block are unchanged by use of **T**.

A.2.4 Line Replacement

The command **P** ("Place") will replace an entire line with the text that follows it, leaving a copy of that new text in the insert buffer, or with the current content of the insert buffer (if **P** is followed by a carriage return).

The line number used by the **P** command is computed from the value in **CHR**. **P** is normally used after the **T** command, as illustrated by the following example:

```
4 T (command)
^THIS IS THE OLD LINE 4(response)
P THIS IS THE NEW LINE 4(command + text)
```

Thus, a line may be placed in several locations in a block by the use of:

- **P** followed by text (the first time).

- Alternate use of **T** (to select the line and confirm that this is the line to be replaced) and **P** followed by a carriage return.

  A **P** followed by two or more spaces and a carriage return will fill the line with spaces. This is useful for blanking single lines.

A.2.5 Line Insertion or Move

The command **U** ("Under") is used to insert either the text that follows or the current contents of the insert buffer into the current program block under the line in which the current value of **CHR** falls. Normally **U** is used immediately after the **T** command, where the line number specifies the line under which a new line is to be inserted.

Handling of text and the insert buffer is the same for **U** as for the command **P**.

The word **M** ("Move") in the editor brings lines into the block you're currently displaying. If you wish to move one or more lines from one block to another, first list the source block. Note the block number and the first line number. Next, list the destination block and select the line just above where you want the line you're going to bring in to be inserted. Now enter `<blk#> <line#> M`. The designated source line will be inserted below the current line. It won't be removed from the source block. The current line will be one below where it was before. Additionally, the source block number and line number will still be on the stack but the line number will be incremented—this sets you up to do another **M** without entering additional arguments. The word **M** checks the stack to be certain it contains only two arguments. It **ABORT**s if the depth isn't two. This saves you from accidentally knocking the last line off your block by inadvertently entering an **M**.

A.2.6 Line Deletion

You may use the command **X** to delete the current line (i.e., the line in which the current value of **CHR** falls). You will normally use **X** immediately after a **T** command that specifies the line to be deleted.

When a line is deleted, all higher-numbered lines shift up by one line, and Line 15 is cleared to spaces. In addition, the contents of the deleted line are placed in the insert buffer, where they may be used by a later command. Thus, **X** may be combined with **T** followed by **P** or **U** to allow movement of one line within a block. The following sequence would move Line 9 to Line 4, changing only the ordering of Lines 4 through 9.

```
9 T  X  3 T  U
```

Note that if a line is being moved to a position later in the block, the **X** operation will change the positions of the later lines. To move the current Line 4 to a position after the current Line 13, use the following command sequence:

```
4 T  X  12 T  U
```

Line 12 is specified as the insert position, since the **X** operation moves the current

Line 13 to the new Line 12.

A.2.7 Character Editing

The string **EDITOR** vocabulary also includes commands to permit editing at the character level. Except in the case of **F** and **S**, the character-editing commands work within a specified range, controlled by the user variable **EXTENT**. **EXTENT** is normally set by default to 64, so that the range will be confined to the current line of the current block. The line is selected by the regular **EDITOR** command:

```
<line#> T
```

A cursor (indicated by a ^ or some form of highlighting) marks the position within the line at which insertions will take place and from which searches will begin. The **T** command sets this cursor to the beginning of the line.

When **EXTENT** is set to 64, insertions will cause characters at the end of the line to be lost; they will not spill over onto the next line. Deletions will cause blank fill on the right end of the line.

EXTENT's value may be set to 1024 in some situations (such as word processing or multiple-line operations) when it's desirable to allow edits to propagate through the entire block. The command **CLIP** sets **EXTENT** to 64 (one line), and the command **WRAP** sets **EXTENT** to 1024 (one block).

In the list of commands below, *text* indicates a string of text. If the text is omitted, the current contents of the find buffer will be used (for the commands **F**, **S**, **D**, and **TILL**) or the current contents of the insert buffer will be used (for **I**). If text is present, it will be left in the appropriate buffer.

The maximum length of a string is determined by the length of the two string buffers being used, at least 64 characters. In all cases, the string is terminated by a carriage return or a caret. If a string is typed that is too long, the string will be truncated to the buffer's size.

APPENDIX B: STANDARD FORTH DOCUMENTATION REQUIREMENTS

This section provides the detailed documentation requirements specified in Standard (ANSI and ISO) Forth. General system documentation is provided first, followed by sections for each wordset with specific documentation requirements. See Section 4.8 for a list of wordsets supported.

References to sections in the Standard are shown in bold, with the section number followed by its heading.

B.1 SYSTEM DOCUMENTATION

This section provides the required system documentation.

Table 15: Implementation-defined options in SwiftForth

Option	SwiftForth Support
Aligned address requirements (**3.1.3.3 Addresses**)	No requirement
Behavior of **6.1.1320 EMIT** for non-graphic characters	All characters transmitted.
Character editing of **6.1.0695 ACCEPT** and **6.2.1390 EXPECT**	**ACCEPT**: BS deletes; CR terminates. Horizontal arrow keys move cursor; typing behavior depends on INS/OVR selection. **EXPECT**: not supported.
Character set (**3.1.2 Character types**, **6.1.1320 EMIT**, **6.1.1750 KEY**)	7-bit ASCII
Character-aligned address requirements (**3.1.3.3 Addresses**)	No requirement
Character-set-extensions matching characteristics (**3.4.2 Finding definition names**)	Case-sensitivity is optional (see Section 1.4.1). Standard words are defined and used in upper case.
Conditions under which control characters match a space delimiter (**3.4.1.1 Delimiters**)	When interpreting a text file, all characters with values lower than BL (20_H) are interpreted as spaces, with the exception of CR ($0D_H$), which is a line terminator.
Format of the control-flow stack (**3.2.3.2 Control-flow stack**)	On data stack during compilation; contains addresses.
Conversion of digits larger than thirty-five (**3.2.1.2 Digit conversion**)	Number conversion follows case-sensitivity preference; otherwise, characters beyond Z not accepted.

Table 15: Implementation-defined options in SwiftForth *(continued)*

Option	SwiftForth Support
Display after input terminates in **6.1.0695 ACCEPT** and **6.2.1390 EXPECT**	**ACCEPT**: in command window, text is displayed following receipt of each character. **EXPECT**: not supported.
Exception abort sequence (as in **6.1.0680 ABORT"**)	Implemented as **-2 THROW**; displays the last word typed followed by the string
Input line terminator (**3.2.4.1 User input device**)	Enter key
Maximum size of a counted string, in characters (**3.1.3.4 Counted strings**, **6.1.2450 WORD**)	255
Maximum size of a parsed string (**3.4.1 Parsing**)	255
Maximum size of a definition name, in characters (**3.3.1.2 Definition names**)	254
Maximum string length for **6.1.1345 ENVIRONMENT?**, in characters	254
Method of selecting **3.2.4.1 User input device**	Set keyboard vectors in “personality” (see Section 5.6.1).
Method of selecting **3.2.4.2 User output device**	Set display vectors in “personality” (see Section 5.6.1).
Methods of dictionary compilation (**3.3 The Forth dictionary**)	The implementation model is subroutine-threaded. See Section 5.5 for information on dictionary structure.
Number of bits in one address unit (**3.1.3.3 Addresses**)	8
Number representation and arithmetic (**3.2.1.1 Internal number representation**)	Two’s complement
Ranges for *n*, *+n*, *u*, *d*, *+d*, and *ud* (**3.1.3 Single-cell types**, **3.1.4 Cell-pair types**)	Single: 32 bits Double: 64 bits
Read-only data-space regions (**3.3.3 Data space**)	None
Size of buffer at **6.1.2450 WORD** (**3.3.3.6 Other transient regions**)	At least 256 bytes.
Size of one cell in address units (**3.1.3 Single-cell types**)	4
Size of one character in address units (**3.1.2 Character types**)	1

Table 15: Implementation-defined options in SwiftForth *(continued)*

Option	SwiftForth Support
Size of the keyboard terminal input buffer (**3.3.3.5 Input buffers**)	256 bytes
Size of the pictured numeric output string buffer (**3.3.3.6 Other transient regions**)	68 bytes
Size of the scratch area whose address is returned by **6.2.2000 PAD** (**3.3.3.6 Other transient regions**)	256 bytes
System case-sensitivity characteristics (**3.4.2 Finding definition names**)	Case-sensitivity is optional (see Section 1.4.1). Default is case-insensitive.
System prompt (**3.4 The Forth text interpreter**, **6.1.2050 QUIT**)	"ok" CR LF
Type of division rounding (**3.2.2.1 Integer division**, **6.1.0100 */**, **6.1.0110 */MOD**, **6.1.0230 /**, **6.1.0240 /MOD**, **6.1.1890 MOD**)	Symmetric
Values of **6.1.2250 STATE** when true	-1 (*true*)
Values returned after arithmetic overflow (**3.2.2.2 Other integer operations**)	Two's complement "wrapped" result
Whether the current definition can be found after **6.1.1250 DOES>** (**6.1.0450 :**).	No

Table 16: SwiftForth action on ambiguous conditions

Condition	Action
A name is neither a valid definition name nor a valid number during text interpretation (**3.4 The Forth text interpreter**)	See Section 5.5.5. If all searches fail, a **-13 THROW** occurs.
A definition name exceeded the maximum length allowed (**3.3.1.2 Definition names**)	Name is truncated.
Addressing a region not listed in **3.3.3 Data Space**	Addressing is allowed to the extent supported by the host OS.
Argument type incompatible with specified input parameter, e.g., passing a flag to a word expecting an *n* (**3.1 Data types**)	Ignore and continue.
Attempting to obtain the execution token, (e.g., with **6.1.0070 '**, **6.1.1550 FIND**, etc.) of a definition with undefined interpretation semantics	Allowed

Table 16: SwiftForth action on ambiguous conditions *(continued)*

Condition	Action
Dividing by zero (**6.1.0100 `*/`**, **6.1.0110 `*/MOD`**, **6.1.0230 `/`**, **6.1.0240 `/MOD`**, **6.1.1561 `FM/MOD`**, **6.1.1890 `MOD`**, **6.1.2214 `SM/REM`**, **6.1.2370 `UM/MOD`**, **8.6.1.1820 `M*/`**)	**`-10 THROW`**
Insufficient data-stack space or return-stack space (stack overflow)	Ignore and continue.
Insufficient space for loop-control parameters	Ignore and continue.
Insufficient space in the dictionary	**`-8 THROW`**
Interpreting a word with undefined interpretation semantics	Compilation semantics are executed.
Modifying the contents of the input buffer or a string literal (**3.3.3.4 Text-literal regions**, **3.3.3.5 Input buffers**)	Ignore and continue.
Overflow of a pictured numeric output string	Ignore and continue.
Parsed string overflow	Truncated to 255 characters.
Producing a result out of range, e.g., multiplication (using **`*`**) results in a value too big to be represented by a single-cell integer (**6.1.0090 `*`**, **6.1.0100 `*/`**, **6.1.0110 `*/MOD`**, **6.1.0570 `>NUMBER`**, **6.1.1561 `FM/MOD`**, **6.1.2214 `SM/REM`**, **6.1.2370 `UM/MOD`**, **6.2.0970 `CONVERT`**, **8.6.1.1820 `M*/`**)	Two's complement "wrapping"
Reading from an empty data stack or return stack (stack underflow)	Data stack checked by text interpreter; **`-4 THROW`** on detected underflow. Return stack not checked.
Unexpected end of input buffer, resulting in an attempt to use a zero-length string as a name	**`-16 THROW`**
`>IN` greater than size of input buffer (**3.4.1 Parsing**)	Abort
6.1.2120 `RECURSE` appears after **6.1.1250 `DOES>`**	Ignore and continue.
Argument input source different than current input source for **6.2.2148 `RESTORE-INPUT`**	**`ABORT`**
Data space containing definitions is de-allocated (**3.3.3.2 Contiguous regions**)	Ignore and continue.
Data space read/write with incorrect alignment (**3.3.3.1 Address alignment**)	Allowed (no alignment required)

Table 16: SwiftForth action on ambiguous conditions *(continued)*

Condition	Action
Data-space pointer not properly aligned (6.1.0150 `,`, 6.1.0860 `C,`)	Allowed (no alignment required)
Less than *u+2* stack items (6.2.2030 `PICK`, 6.2.2150 `ROLL`)	Ignore and continue.
Loop-control parameters not available (6.1.0140 `+LOOP`, 6.1.1680 `I`, 6.1.1730 `J`, 6.1.1760 `LEAVE`, 6.1.1800 `LOOP`, 6.1.2380 `UNLOOP`)	Ignore and continue.
Most recent definition does not have a name (6.1.1710 `IMMEDIATE`)	Ignore and continue; `IMMEDIATE` has no effect.
Name not defined by 6.2.2405 `VALUE` used by 6.2.2295 `TO`	`-32 THROW`
Name not found (6.1.0070 `'`, 6.1.2033 `POSTPONE`, 6.1.2510 `[']`, 6.2.2530 `[COMPILE]`)	`-13 THROW`
Parameters are not of the same type (6.1.1240 `DO`, 6.2.0620 `?DO`, 6.2.2440 `WITHIN`)	Allowed
6.1.2033 `POSTPONE` or 6.2.2530 `[COMPILE]` applied to 6.2.2295 `TO`	`[COMPILE]` not supported; `POSTPONE` not allowed with `TO`.
String longer than a counted string returned by 6.1.2450 `WORD`	Length truncated
u greater than or equal to the number of bits in a cell (6.1.1805 `LSHIFT`, 6.1.2162 `RSHIFT`)	Shift is modulo cell size.
Word not defined via 6.1.1000 `CREATE` (6.1.0550 `>BODY`, 6.1.1250 `DOES>`)	Allowed
Words improperly used outside 6.1.0490 `<#` and 6.1.0040 `#>` (6.1.0030 `#`, 6.1.0050 `#S`, 6.1.1670 `HOLD`, 6.1.2210 `SIGN`)	Allowed

Table 17: Other system documentation

Requirement	SwiftForth information
List of non-standard words using 6.2.2000 `PAD` (3.3.3.6 **Other transient regions**)	To obtain list of references, type `WH PAD` and follow line# links (see Section 2.4.3)
Operator's terminal facilities available	See Section 2.3.1
Program data space available, in address units	1 MB default; see Section 5.2.
Return stack space available, in cells	16,384
Stack space available, in cells	4,096

Table 17: Other system documentation *(continued)*

Requirement	SwiftForth information
System dictionary space required, in address units	About 300K bytes

B.2 BLOCK WORDSET DOCUMENTATION

Table 18: Block wordset implementation-defined options

Option	SwiftForth support
The format used for display by **7.6.2.1770 LIST** (if implemented)	16 lines by 64 characters, with line numbers.
The length of a line affected by **7.6.2.2535 ** (if implemented)	64 characters

Table 19: Block wordset ambiguous conditions

Condition	Action
Correct block read was not possible	**ABORT**
I/O exception in block transfer	**ABORT**
Invalid block number (**7.6.1.0800 BLOCK**, **7.6.1.0820 BUFFER**, **7.6.1.1790 LOAD**)	**ABORT**
A program directly alters the contents of **7.6.1.0790 BLK**	Ignore and continue.
No current block buffer for **7.6.1.2400 UPDATE**	There is always a current block buffer.

Table 20: Other block wordset documentation

Item	SwiftForth information
Any restrictions a multiprogramming system places on the use of buffer addresses	Valid only within a “critical section” or as controlled by **DISK GET**/**DISK RELEASE** (see ???).
The number of blocks available for source text and data	User specified; see Section A.1.

B.3 Double Number Wordset Documentation

Table 21: Double-number wordset ambiguous conditions

Condition	Action
d outside range of *n* in **8.6.1.1140 D>S**.	Value truncated.

B.4 Exception Wordset Documentation

Table 22: Exception wordset implementation-defined options

Option	SwiftForth support
Values used in the system by **9.6.1.0875 CATCH** and **9.6.1.2275 THROW** (**9.3.1 THROW values**, **9.3.5 Possible actions on an ambiguous condition**).	See source file: **SwiftForth/src/ide/errmessages.f**

B.5 Facility Wordset Documentation

Table 23: Facility wordset implementation-defined options

Option	SwiftForth support
Encoding of keyboard events (**10.6.2.1305 EKEY**)	See Section 5.5.2.
Duration of a system clock tick	Varies; typically 1 ms. to 55 ms.
Repeatability to be expected from execution of **10.6.2.1905 MS**.	At least the requested number of milliseconds.

Table 24: Facility wordset ambiguous conditions

Condition	Action
10.6.1.0742 AT-XY operation can't be performed on user output device.	Ignore and continue.

B.6 FILE-ACCESS WORDSET DOCUMENTATION

Table 25: File-access wordset implementation-defined options

Option	
File access methods used by **11.6.1.0765 BIN**, **11.6.1.1010 CREATE-FILE**, **11.6.1.1970 OPEN-FILE**, **11.6.1.2054 R/O**, **11.6.1.2056 R/W**, and **11.6.1.2425 W/O**	Methods are defined in terms of **GENERIC_READ** and **GENERIC_WRITE**. The **BIN** method is a no-op, as all access is binary.
File exceptions	-36 " Invalid file position" -37 " File I/O exception" -38 " Non-existent file" -39 " Unexpected end of file"
File line terminator (**11.6.1.2090 READ-LINE**)	CR ($0D_H$)
File name format (**11.3.1.4 File names**)	Linux/macOS standard.
Information returned by **11.6.2.1524 FILE-STATUS**	Zero if available; otherwise, *ior* is encoded per the list below.
Input file state after an exception (**11.6.1.1717 INCLUDE-FILE**, **11.6.1.1718 INCLUDED**)	Closed automatically if the error was **THROW**n.
ior values and meaning (**11.3.1.2 I/O results**)	-191 Can't delete file (**DELETE-FILE**) -192 Can't rename file (**RENAME-FILE**) -193 Can't resize file (**RESIZE-FILE**) -194 Can't flush file (**FLUSH-FILE**) -195 Can't read file (**READ-FILE**, **READ-LINE**) -196 Can't write file (**WRITE-FILE**) -197 Can't close file (**CLOSE-FILE**) -198 Can't create file (**CREATE-FILE**) -199 Can't open file (**OPEN-FILE**, **INCLUDE-FILE**) These always return 0 (success, or ignore and continue) **FILE-POSITION** **REPOSITION-FILE** SEEK-FILE **FILE-SIZE**
Maximum depth of file input nesting (**11.3.4 Input source**)	16
Maximum size of input line (**11.3.6 Parsing**)	256

Table 25: File-access wordset implementation-defined options *(continued)*

Option	
Methods for mapping block ranges to files (**11.3.2 Blocks in files**)	See Section A.1.
Number of string buffers provided (**11.6.1.2165 S"**)	8
Size of string buffer used by **11.6.1.2165 S"**.	128 each

Table 26: File-access wordset, ambiguous conditions

Condition	Action
Attempting to position a file outside its boundaries (**11.6.1.2142 REPOSITION-FILE**)	No error; file at EOF.
Attempting to read from file positions not yet written (**11.6.1.2080 READ-FILE**, **11.6.1.2090 READ-LINE**)	No error; no bytes read, zero length returned.
Fileid is invalid (**11.6.1.1717 INCLUDE-FILE**)	Dependent on value of *fileid*; system will attempt to use it.
I/O exception reading or closing *fileid* (**11.6.1.1717 INCLUDE-FILE**, **11.6.1.1718 INCLUDED**)	**-37 THROW**
Named file cannot be opened (**11.6.1.1718 INCLUDED**)	**-38 THROW**
Requesting an unmapped block number (**11.3.2 Blocks in files**)	**ABORT**
Using **11.6.1.2218 SOURCE-ID** when **7.6.1.0790 BLK** is not zero.	The block source has priority.

B.7 Floating Point Wordset Documentation

Table 27: Floating-point wordset, implementation-defined options

Option	Swiftforth support
Format and range of floating-point numbers (**12.3.1 Data types**, **12.6.1.2143 REPRESENT**)	IEEE 64-bit floating point numbers; see Section 10.1.
Results of **12.6.1.2143 REPRESENT** when *float* is out of range	FPU trap

Table 27: Floating-point wordset, implementation-defined options *(continued)*

Option	Swiftforth support
Rounding or truncation of floating-point numbers (**12.3.1.2 Floating-point numbers**)	See Section 10.2.
Size of floating-point stack (**12.3.3 Floating-point stack**)	32 items (may be one for each task)
Width of floating-point stack (**12.3.3 Floating-point stack**).	80 bits

Table 28: Floating-point wordset ambiguous conditions

Condition	Action
DF@ or **DF!** is used with an address that is not double-float aligned	No alignment requirement.
F@ or **F!** is used with an address that is not float aligned	No alignment requirement.
Floating point result out of range (e.g., in **12.6.1.1430 F/**)	FPU trap
SF@ or **SF!** is used with an address that is not single-float aligned	No alignment requirement.
BASE is not decimal (**12.6.1.2143 REPRESENT**, **12.6.2.1427 F.**, **12.6.2.1513 FE.**, **12.6.2.1613 FS.**)	Ignore and continue.
Both arguments equal zero (**12.6.2.1489 FATAN2**)	Ignore and continue.
Cosine of argument is zero for **12.6.2.1625 FTAN**	Ignore and continue.
d can't be precisely represented as *float* in **12.6.1.1130 D>F**	Ignore and continue.
Dividing by zero (**12.6.1.1430 F/**)	FPU trap
Exponent too big for conversion (**12.6.2.1203 DF!**, **12.6.2.1204 DF@**, **12.6.2.2202 SF!**, **12.6.2.2203 SF@**)	Ignore and continue.
float less than one (**12.6.2.1477 FACOSH**)	FPU trap
float less than or equal to minus-one (**12.6.2.1554 FLNP1**)	FPU trap
float less than or equal to zero (**12.6.2.1553 FLN**, **12.6.2.1557 FLOG**)	FPU trap
float less than zero (**12.6.2.1487 FASINH**, **12.6.2.1618 FSQRT**)	FPU trap
float magnitude greater than one (**12.6.2.1476 FACOS**, **12.6.2.1486 FASIN**, **12.6.2.1491 FATANH**)	FPU trap
Integer part of *float* can't be represented by *d* in **12.6.1.1470 F>D**	Truncated; invalid data.

Table 28: Floating-point wordset ambiguous conditions *(continued)*

Condition	Action
String larger than pictured-numeric output area (12.6.2.1427 **F.**, 12.6.2.1513 **FE.**, 12.6.2.1613 **FS.**)	Ignore and continue.

B.8 Local Variables Wordset Documentation

Table 29: Local variables wordset implementation-defined options

Option	SwiftForth Support
Maximum number of locals in a definition (13.3.3 **Processing locals**, 13.6.2.1795 **LOCALS|**)	16

Table 30: Local variables ambiguous conditions

Condition	Action
Executing a named local while in interpretation state (13.6.1.0086 **(LOCAL)**)	Outside a definition: name not found. Inside a definition (e.g., after **[**): compiles a reference.
Name not defined by **VALUE** or **LOCAL** (13.6.1.2295 **TO**)	**-32 THROW**

B.9 Memory Allocation Wordset Documentation

Table 31: Memory allocation wordset implementation-defined options

Option	Swiftforth support
Values and meaning of *ior* (14.3.1 **I/O Results data type**, 14.6.1.0707 **ALLOCATE**, 14.6.1.1605 **FREE**, 14.6.1.2145 **RESIZE**).	Zero indicates success. **ALLOCATE** returns -100 on failure; **FREE** returns -102 on failure; **RESIZE** calls **ALLOCATE** and **FREE**.

B.10 Programming Tools Wordset Documentation

Note: the obsolescent word **FORGET** is not supported by SwiftForth.

Table 32: Programming tools wordset implementation-defined options

Option	SwiftForth support
Ending sequence for input following **15.6.2.0470 CODE** and **15.6.2.0930 CODE**	**END-CODE** (see Section 6.2)
Manner of processing input following **15.6.2.0470 CODE** and **15.6.2.0930 CODE**	Assembles instructions, as described in Section Section 6:.
Search-order capability for **15.6.2.1300 EDITOR** and **15.6.2.0740 ASSEMBLER** (**15.3.3 The Forth dictionary**)	Both supported. **EDITOR** provides access to a block editor (see Section A.2).
Source and format of display by **15.6.1.2194 SEE**.	See example in Section 2.4.3.

Table 33: Programming tools wordset ambiguous conditions

Condition	Action
Deleting the compilation word-list (**15.6.2.1580 FORGET**)	**FORGET** not supported.
Fewer than *u*+1 items on control-flow stack (**15.6.2.1015 CSPICK**, **15.6.2.1020 CSROLL**)	Ignore and continue.
name can't be found (**15.6.2.1580 FORGET**)	**FORGET** not supported.
name not defined via **6.1.1000 CREATE** (**15.6.2.0470 ;CODE**)	Allowed
6.1.2033 POSTPONE applied to **15.6.2.2532 [IF]**	Ignore and continue.
Reaching the end of the input source before matching **15.6.2.2531 [ELSE]** or **15.6.2.2533 [THEN]** (**15.6.2.2532 [IF]**)	Ignore and continue.
Removing a needed definition (**15.6.2.1580 FORGET**).	**FORGET** not supported.

B.11 Search-order Wordset Documentation

Table 34: Search-order wordset implementation-defined options

Option	SwiftForth support
Maximum number of word lists in the search order (**16.3.3 Finding definition names**, **16.6.1.2197 SET-ORDER**)	16
Minimum search order (**16.6.1.2197 SET-ORDER**, **16.6.2.1965 ONLY**).	**FORTH**

Table 35: Search-order wordset ambiguous conditions

Condition	Action
Changing the compilation word list (**16.3.3 Finding definition names**)	Ignore and continue.
Search order empty (**16.6.2.2037 PREVIOUS**)	Can't find any words.
Too many word lists in search order (**16.6.2.0715 ALSO**).	Ignore and continue.

APPENDIX C: FORTH WORDS INDEX

This section provides an alphabetical index to the Forth words that appear in the glossaries in this book. Each word is shown with its stack arguments and with a page number where you may find more information. If you're viewing the PDF version of this document, you can click on the Forth word name to go to its glossary definition.

The stack-argument notation is described in Table 36. Where several arguments are of the same type, and clarity demands that they be distinguished, numeric subscripts are used.

Table 36: Notation used for data types of stack arguments

Notation	Description
addr	A cell-wide byte address.
b	A byte, stored as the least-significant 8 bits of a stack entry. The remaining bits of the stack entry are zero in results or are ignored in arguments.
c	An ASCII character, stored as a byte (see above) with the parity bit reset to zero.
d	A double-precision, signed, 2's complement integer, stored as two stack entries (least-significant cell underneath the most-significant cell). On 32-bit machines, the range is from -2**63 through +2**63-1.
flag	A single-precision Boolean truth flag (zero means *false*, non-zero means *true*).
*i*x, j*x, etc.*	Zero or more cells of unspecified data type.
n	A signed, single-precision, 2's complement number. On 32-bit machines, the range is from -2**31 through +2**31-1. (Note that Forth arithmetic rarely checks for integer overflow.) If a stack comment is shown as *n*, *u* is also implied unless specifically stated otherwise (e.g., **+** may be used to add either signed or unsigned numbers). If there is more than one input argument, signed and unsigned types may not be mixed.
+n	A single-precision, unsigned number with the same positive range as *n* above. An input stack argument shown as *+n* must not be negative.
r	A floating-point number in IEEE long floating-point format (ANS/IEEE 754-1985), as discussed in Section 10.2.2.
u	A single-precision, unsigned number with a range from 0 through 2**32-1 on 32-bit machines.
ud	A double-precision, unsigned integer with a range from 0 through 2**64-1 on 32-bit machines.
x	A cell (single stack item), otherwise unspecified.
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.

Table 37: Index of Forth Words

Table 37: Index of Forth Words

Table 37: Index of Forth Words

Table 37: Index of Forth Words

Table 37: Index of Forth Words

Table 37: Index of Forth Words

Word	Stack	Page
IF	*(cc — addr)*	*90*
IMMEDIATE	*(—)*	*67*
INCLUDE <filename>[<.ext>]	*(—)*	*29*
INCLUDING	*(— c-addr u)*	*29*
IS <name>	*(xt —)*	*55*
KILL	*(addr —)*	*100*
L	*(—)*	*20*
L:	*(n —)*	*88*
L#	*(n — addr)*	*88*
LABEL <name>	*(—)*	*79*
LIBRARY <filename>	*(—)*	*102*
LMATRIX <name>	*(nr nc —)*	*130*
LMD <name>	*(nr nc —)*	*130*
LOADUSING <name>	*(n —)*	*138*
LOCALS| <$name_1$> <$name_2$> ... <$name_n$> **|**	*(x_n ... x_2 x_1 —)*	*56*
LOCATE <name>	*(—)*	*20*
M/D/Y	*(ud — u)*	*46*
MAKE-FLOOR	*(—)*	*128*
MAKE-ROUND	*(—)*	*128*
MAKES <name>	*(hclass — addr)*	*111*
MAPS <name>	*(n —)*	*137*
MAPS" <name>**"**	*(n —)*	*137*
MEMBERS	*(—)*	*119*
MEMTOP	*(— addr)*	*65*
MESSAGE:	*(n —)*	*114*
MODIFY	*(n —)*	*138*
MS	*(n —)*	*48*
N	*(—)*	*20*
N.	*(—); (F: r —)*	*125*
NAME>	*(addr — xt)*	*68*
NAMES <name>	*(addr hclass —)*	*111*
NEVER	*(— cc)*	*92*
NEW	*(hclass — addr)*	*111*
NEWFILE <name>	*(n —)*	*137*
NEXT-WORD	*(— addr u)*	*57*
NH	*(— addr)*	*43*
NOT	*(cc_1 — cc_2)*	*92*

Table 37: Index of Forth Words

Table 37: Index of Forth Words

Table 37: Index of Forth Words

INDEX

For a comprehensive index of SwiftForth commands, see Appendix C.