# SwiftForth®

## Development System for Windows

# 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

## Section 6: i386 Assembler



## Section 7: Multitasking and Windows



## Section 8: Windows Programming in SwiftForth



## Section 9: Defining and Managing Windows Features

## List of Figures

## List of Tables

# Welcome!

## What is SwiftForth?

SwiftForth is FORTH, Inc.'s interactive development system for the Windows and Linux 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 Windows and Linux 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 the Windows OS.

## 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 Windows environment. It assumes general knowledge of the Windows 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 sample

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 FORTH, Inc. and Amazon.com, as can the programming tutorial *Forth Application Techniques*.

## 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.

## 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 Windows, including information necessary to help you install the system and to become familiar with its principal features.

## 1.1 Components of SwiftForth

SwiftForth consists of the following components:

- Executable image of the SwiftForth development system, including the Interactive Development Environment (IDE), Windows interface functions, and programming aids including the disassembler and other tools.
- Forth language source files supplied depend on the version:
- The evaluation version includes source for all options and examples.
- A purchased SwiftForth adds source for the entire SwiftForth system, including the cross-compiler that allows advanced programmers to modify the underlying system.
- On-line documentation, including PDF versions of all manuals.

The SwiftForth directory structure is shown in Figure 1. The evaluation version does not include **the console, kernel, or xcomp directories** (the source for the debug window, kernel, and cross compiler).

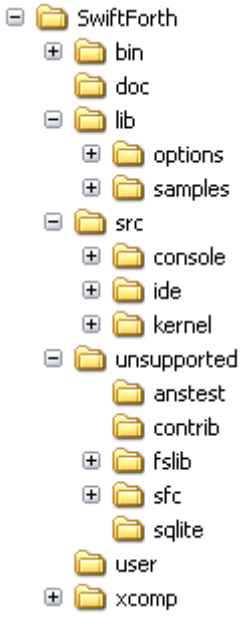


**Figure 1. SwiftForth directory structure**

## 1.2 SwiftForth System Requirements

In order to use this package, you need the following:

- PC running Windows[1]. At least 256 MB of RAM is recommended. The package will occupy approximately 36 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.

### 1.3.1 Installing the Host Software

Follow this procedure to install your SwiftForth software:

1. Turn on your computer and start Windows.
2. If you are installing from a downloaded file, simply launch the SwiftForth installer you downloaded. If you are installing from a CD, the SwiftForth Installer should launch automatically when you insert the CD; if it does not, you may launch the SwiftForth Installer on the CD manually. The install procedure may prompt you for additional information or choices.

   The default main directory for purchased versions of SwiftForth is `C:\Forthinc\SwiftForth`; SwiftForth evaluation versions will install in the default directory `C:\ForthInc-Evaluation\SwiftForth`. If you prefer another directory name, you will be given a chance to change it during the installation. However, this manual will assume this path as the "root" for all examples.

**We strongly recommend that you avoid using a path with spaces in it, such as "Program Files", as many standard Forth functions use a space as a delimiter.**

   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).

   The installation procedure creates a SwiftForth program group, from which you may launch SwiftForth or view the release notes. It also provides PDF copies of the *SwiftForth Reference Manual* and the *Forth Programmer's Handbook*. These may be launched from SwiftForth's Help menu.

   You may launch SwiftForth from the Start menu or from its icon in the **\Bin** directory.

1.Any WIN32 version, Windows 95 or later.

### 1.3.2 Linking to a Text Editor

The SwiftForth Interactive Development Environment (IDE) contains a number of programmer aids (discussed in Section 2.4) that are facilitated by a direct link to a text editor. The default editor is Microsoft Notepad; alternatively, you may configure SwiftForth to use an editor of your choice by specifying certain command formats (see "Options Menu" on page 25).

## 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

Launch SwiftForth from the Start menu or by double-clicking the **Sf.exe** icon. The main window SwiftForth presents is called the *command window*. Within this window, SwiftForth will attempt to execute everything you type. You may also use your mouse to select words and to perform other functions. The command window is described in Section 2.3.1.

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.

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. *Windows calls, however, are case-sensitive*, and are spelled with mixed case following standard Windows nomenclature; these words should always be spelled exactly as shown. For this reason, 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**; however, we do not recommend operating for extended periods in **CASE-SENSITIVE** mode, as aspects of the object system (Section 10) may not function properly in this mode.

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 range from simple functions to moderately complex games with simple graphics and sound. 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 Windows-specific functions. 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 ships 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. Many may be loaded from the dialog box invoked by the Tools > Optional Packages > Generic Samples or Win32 Samples menu items.

*References* **INCLUDE** (loading source files), Section 3.1
Tools menu selections, Section 2.3.5

### 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 command window (you can also copy definitions from the command window or keyboard history window and paste them into files).
- Interpret an entire source file using the File > Include menu option, or by typing **INCLUDE** `<filename>`.
- 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 window, Section 2.4.8

## 1.5 Licensing Issues

When you install SwiftForth, the installation process obtains your consent to a license agreement describing the terms under which you may use this product. You may find a copy of this license agreement in **SwiftForth\doc\license.rtf**.

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 Windows applications. Your applications 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 application software. 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 are welcome to purchase additional copies of SwiftForth documentation.

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 Swift-

Forth 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 Windows applications that can deliver very fast performance, and full access to standard Windows features, DLLs, and other powerful software found in this environment.

This introductory section gives a general view of the design of the SwiftForth 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 textbook offered by FORTH, Inc. 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 code 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 12). 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 virtually any Windows application. 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 Windows programming. The complete package includes:

- Forth language compiler
- i386-family assembler
- Windows system interface functions
- 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 17.**

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 user interface that may be managed with toolbar buttons and pull-down menus, or directly from the command line. This section summarizes its principal features.

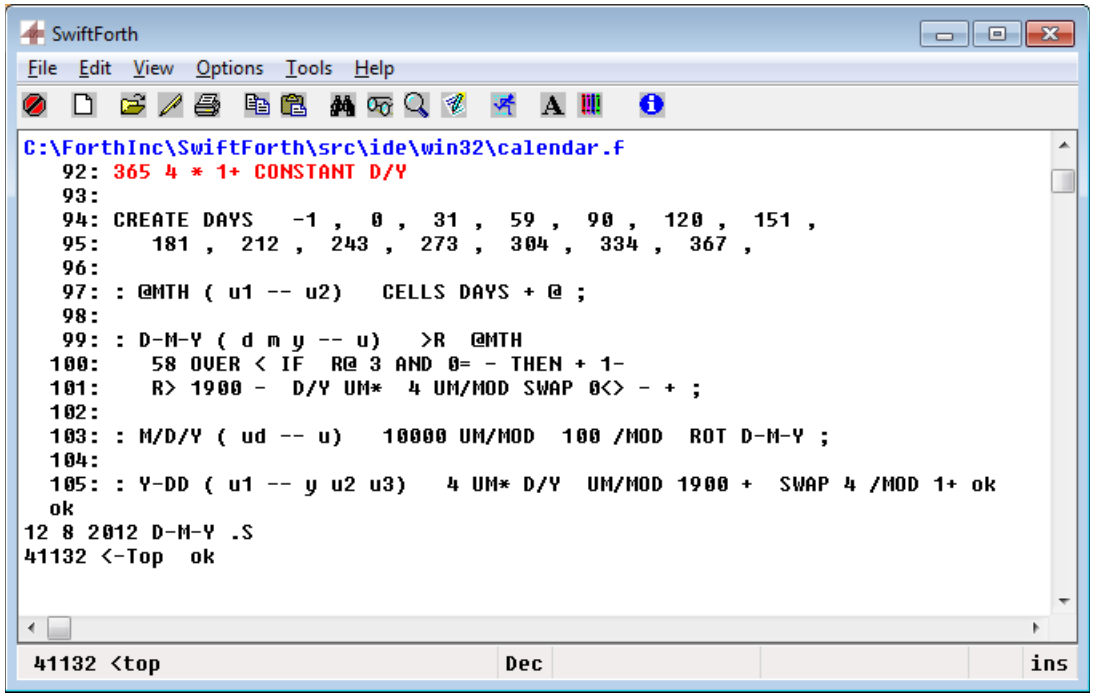


**Figure 2. The SwiftForth command window**

### 2.3.1 The Command Window

Your main interface with SwiftForth is through the *command window*, which is displayed when the system boots. In this window, you may type commands, which will

be executed by SwiftForth.

All information displayed in the command window (including commands you type, and system responses or displays) is kept in a large buffer while the IDE is running; you may scroll through it, using the scroll bar or the PageUp and PageDown keys, to see previous parts of the session. You may also print or save the entire buffer, or a portion of it that you select using the mouse.

The toolbar at the top of the command window provides one-click access to several menu options described in the following sections (see Figure 3).

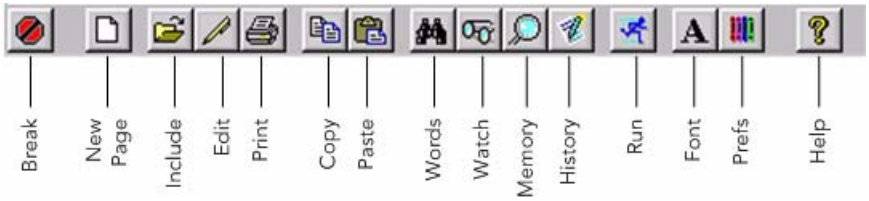


**Figure 3. Toolbar items**

The status bar at the bottom of the command window shows the stack depth and the actual values of the top several items, the current number base (the default is decimal), and the current insert/overwrite typing mode (see Figure 4). Clicking on the Base and Insert/Overwrite areas will toggle through possible values.

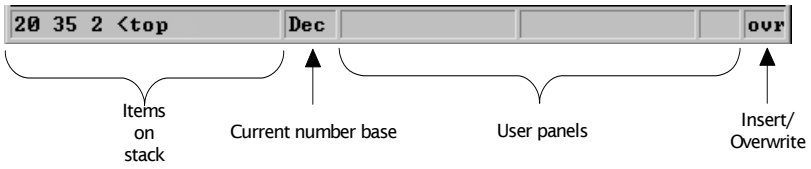


**Figure 4. Status bar indicators**

In addition to the buffer containing the general history of the command window's contents, SwiftForth remembers the last several commands you type, storing them in a circular queue. The up-arrow and down-arrow keys allow you to retrieve these command lines from the 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 toggle through entries in the circular command-line buffer.

Double-clicking any word in the command window will select it. The right mouse button presents a menu of operations you can apply to a selected word, shown in Figure 5.

- *Locate* displays the source for any word defined in your current scope by double-clicking the word. If the source is not present (e.g. you are running a turnkey but do not have the source files that generated it) or if the word you are trying to LOCATE was defined interactively in the command window, you will see the error message "Source file not available." This feature is discussed further in Section 2.4.1.
- *Edit* launches or switches to your linked editor (see Sections 2.3.5 and 2.4.1) positioned at the source for the word (if it's available).
- *See* disassembles the word, as described in Section 2.4.4.
- *Cross Ref* generates a cross-reference for the word, as described in Section 2.4.3.
- *Execute* executes the word, just as though you had typed it and pressed `Enter`.
- *Copy* copies it to the clipboard.
- *Paste* pastes whatever is currently in the clipboard at the current cursor position in the command window.

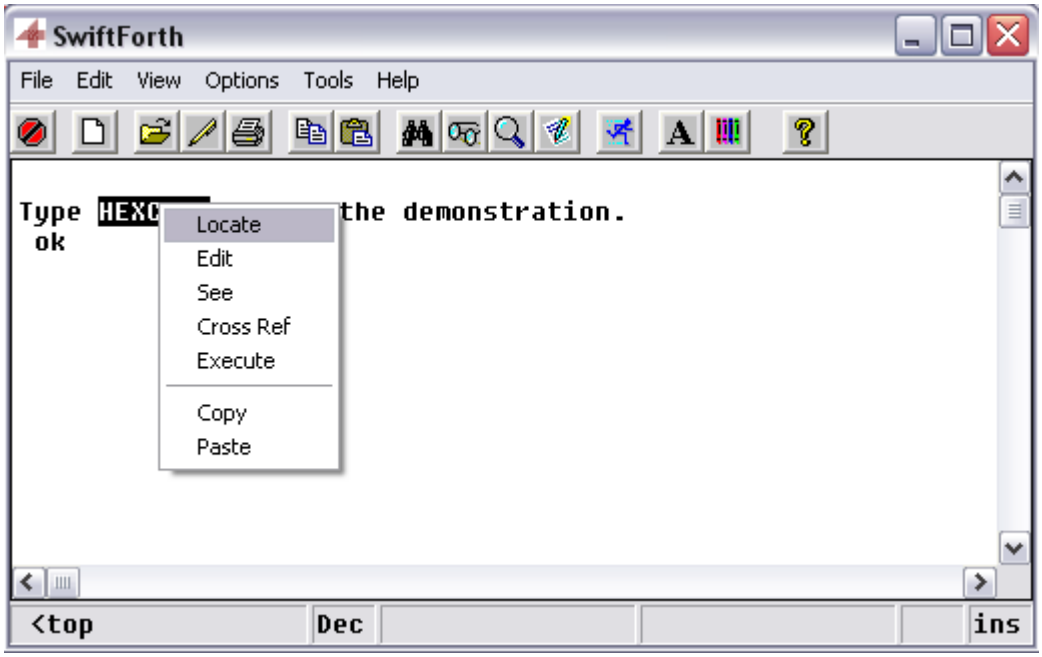


**Figure 5. Right mouse button actions on a selected word**

Copy and Paste (below the horizontal line in the menu) are a little bit different. The functions above the line require that you have selected a recognizable Forth word (or, in the case of Execute, an executable word or number). Copy does not depend on the selected text being a defined Forth word, and Paste will ignore any selected text.

Table 1 summarizes the special keyboard functions available in the command win-

dow.

**Table 1: Command window keyboard controls**

| Key | Action |
|---|---|
| PageUp<br>PageDown | Scroll through the history of the current session. |
| UpArrow<br>DownArrow | Retrieve commands you have typed. |
| LeftArrow<br>RightArrow | Move cursor on command line. |
| Insert | Toggle the insert/overwrite mode of typing. |
| Enter | Execute the command line the cursor is on. |
| Ctrl-C<br>Ctrl-V | Copy from, or paste text into, the window.<br>(See Section 2.3.3.) |
| Double-click on a word | Selects the word, making it available for the right-click functions. |
| Right-click | Display pop-up menu options: Locate, Edit, See, Cross Ref, Execute, Copy, and Paste. |
| Ctrl-Home | Show history |
| Ctrl-Shift-Del | Delete to end of line |
| F3 | Recall last line entered |

The following sections describe the options available from the command window's menu and toolbar. Where a letter in a menu item is underlined, the Alt key plus that letter is a keyboard equivalent. In each case, we list the menu item, equivalent commands (if any, in addition to the Alt versions), and a description.

### 2.3.2 File Menu

The File menu offers the selections described in Table 2.

**Table 2: File menu options**

| Item | Command | Action |
|---|---|---|
| Include | **INCLUDE** <filename> | Interpret a file (load it and any files it loads). Displays a file-selection dialog box; **INCLUDE** processes the file *filename*. |
| Edit | **EDIT** <path> or<br>**EDIT** <wordname> | Launch a linked editor, allowing you to select a source file or word. |
| Print | | Print the command window.<br>In the print dialog, you may choose to print the entire contents or a selected portion. |

**Table 2: File menu options *(continued)***

| Item | Command | Action |
|---|---|---|
| Save Command Window | | Record the current contents of the command window in a text file. |
| Save Keyboard History | | Record all the commands you've typed in this session in a text file. |
| Session Log | | Start recording all actions for this session in a text file. |
| Break | | Force the main console task in SwiftForth to abort; used for error recovery. |
| Exit | **bye** | Exit SwiftForth. |

The Edit menu option opens a file for editing, using your linked editor (see Sections 2.3.5 and 2.4.1).

Both File > Include and File > Edit and their corresponding toolbar buttons display a "Browse" dialog box with which you can find your file. Both reset SwiftForth's path to the one for the file you select. However, **`INCLUDE`** typed from the keyboard does *not* change SwiftForth's path.

Save Command Window, Save Keyboard History, and Session Log are discussed further in Section 2.4.8.

### 2.3.3 Edit Menu

Most editing in SwiftForth is done with your associated editor (see Section 2.3.5 and Section 2.4). However, you can copy text—from a file in another window or from elsewhere in the command window—and paste it into the command window, which will have the same effect as typing it. You may also select text for typing, saving, or copying into another window. Edit menu options are summarized in Table 3. (Cut and Delete are not available in the command window, since the purpose of the command window is to maintain a record of your actions during this programming session.)

**Table 3: Edit menu options**

| Item | Keystroke | Action |
|---|---|---|
| Copy | Ctrl-C | Copy selected text to your clipboard. |
| Paste | Ctrl-V | Paste the current clipboard contents on a new command line and interpret its contents. |
| Select all | | Selects the entire contents of the command window. |
| Wipe all | | Clear the entire command window. |

Toolbar buttons are available for Copy and Paste.

### 2.3.4 View Menu

The View menu provides alternate views of the command window. If the feature is active, a checkmark appears next to it on the menu. See Table 4.

**Table 4: View menu options**

| Item | Action |
|---|---|
| Status line | When checked, displays the status line at the bottom of the debug window. |
| Toolbar | When checked, displays the toolbar at the top of the debug window. |

### 2.3.5 Options Menu

The Options menu provides more ways to customize SwiftForth. Options menu selections are summarized in Table 5.

**Table 5: Options menu selections**

| Item | Action |
|---|---|
| Font | Select a font for the command window.<br>Only fixed-width (i.e., non-proportional) fonts are listed. |
| Editor | Select and set parameters for your editor. |
| Preferences | Set text and background colors for normal and highlighted displays and other options. |
| Warnings | Enable/disable various system warnings, and establish how they and error messages will be displayed. |
| Include Monitoring | Sets options for diagnostic features to be performed during file **INCLUDE** operations. See Section 3.3 for details. |
| FPMath Options | If floating-point math support has been loaded (Tools > Optional Packages > Generic Options > FPMath), customize related features. |
| Save Options | Save all current settings. |

To use an editor other than Notepad, which is the default when SwiftForth is shipped, use the Options > Select Editor menu item, and type your editor's pathname into the box or click the browse button to search for it. After you have provided the pathname, the User Defined radio button should be highlighted.


**Figure 6. Select Editor dialog**

Next, you must specify how SwiftForth is to pass line-number and filename parameters to your editor. The specification format is:

```
<line-selection string> %l  "<file-selection string>%f"
```

Note that the file-selection string and **`%f`** require quotation marks around them, since the filename may contain spaces.

When SwiftForth calls the editor, it will provide the line number at the place in this string that has a **`%l`** (lower-case L), and will provide the filename at the place that has a **`%f`**. Sample parameter strings for some editors are shown in Table 6. SwiftForth already knows the appropriate strings for these and other popular editors; if the one you've selected is among them, it will automatically enter its strings. Otherwise, you may fill them in.

**Table 6: Examples of editor parameter sequences**

| Editor name | Parameter string |
|---|---|
| CodeWright | "%f" -g%l |
| E | -n%l  "%f" |
| ED4W | `-1 -n -l %l "%f"`<br>(*note:* the first is minus one, the others are lower-case Ls) |
| EMACS | +%l "%f" |
| MultiEdit | %f /L%l" |
| TextPad | -am -q %f(%l,0)" |
| TSE | -n%l %f" |
| Ultra Edit | %f/%l |

When your editor's pathname and parameter string are correct, click Done. This information will be saved when you exit SwiftForth.

SwiftForth's Options > Preferences dialog (or its equivalent Toolbar button) lets you specify a number of configuration items, shown in Figure 7.

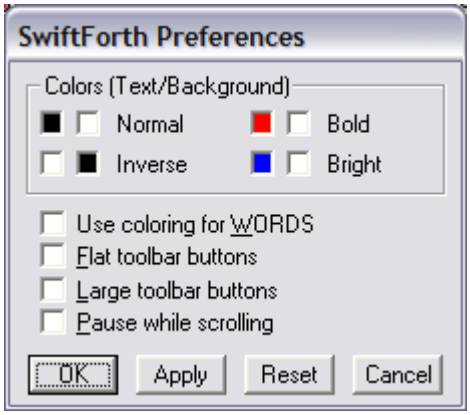


**Figure 7. Preferences dialog**

The Colors section controls the color scheme for the command window.

The checkbox “Use coloring for WORDS” vectors the command **WORDS** (described in Section 2.4.2). If this box is checked, typing **WORDS** in the command window will produce a color-coded display in the command window; otherwise, it will launch the Words browser window. The behavior of the Words browser when launched from the toolbar is not affected by this option.

“Save options on exit” means your selections will be recorded so that they will still be in effect the next time you launch SwiftForth. Note that the “Reset” button will restore all options to the system defaults.

The “Flat” and “Large” toolbar buttons affect the appearance of the toolbar, if it is displayed.

The Options > Warnings dialog (Figure 8) provides configuration settings that determine whether, and where, error messages and system warnings will appear.

Even if warnings are disabled, error messages will always be displayed. See Section 4.7 (”Exceptions and Error Handling”). If the **PROTECTION** option (discussed in Section 8.2.2) is loaded, this dialog box will be enhanced to provide options for controlling its address checking parameters.

Since SwiftForth is normally case-insensitive, the box labeled “Warn for case ambiguity” is normally un-checked. If it is checked, it will warn of name conflicts based on case alone.

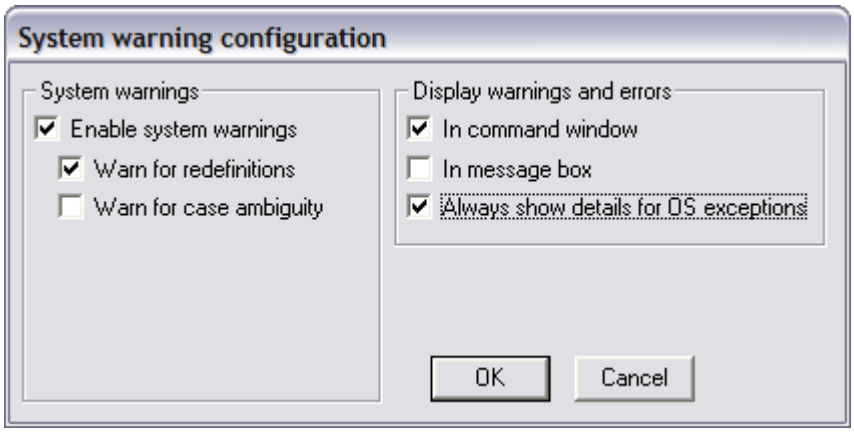


**Figure 8. System warning configuration dialog**

Options > Include monitoring configures diagnostics that can be used when you **INCLUDE** a source file. These are discussed in detail in Section 3.3.

Options > FPMath Options displays a dialog box that provides several ways to customize floating-point options. This menu selection is only available if floating-point math support has been loaded (Tools > Optional Packages > System Options > FPMath). See Section Section 12: for details.

SwiftForth is built using a kernel plus a set of additional features. These are incorporated into the **Sf.exe** program you usually boot. If your version of SwiftForth includes system source code, you can customize it by adding or removing features.

*References*

**WORDS** command, Section 2.4.2
**INCLUDE**, Section 3.1
Floating-point math library, Section Section 12:

### 2.3.6 Tools Menu

This menu provides tools that may be helpful in the development process.

**Table 7: Tools menu options**

| Item | Command | Action |
|---|---|---|
| Words | **WORDS** | Display the words available in the current scope. |
| Watch | **WATCH** | Set up *watch points*, or monitored locations. |
| Memory | **DUMP** | Display a region of memory. |
| History | | Open command history window, and start recording |
| Run | **RUN** | Run an auxiliary program, such as a checkout procedure for a version-control program. |
| Optional Packages | | Select from among many optional libraries. |

Watch allows you to configure a few, single-cell regions of memory to be monitored on a regular basis while the program runs. On the other hand, Memory provides a single window into a contiguous region of memory, which can be extensive and is scrollable. Both of these incur some overhead by maintaining the display, so they should be used sparingly.

In addition, a number of optional features are included with the package. They may be loaded using the Options > Optional Packages sub-menu, which offers several choices; each displays a dialog box from which you can select the items you prefer. There are several major options, and a number of minor ones, including the sample programs described in Section 1.4.2.

The many options offered by SwiftForth are organized into four groups, shown in Table 8.

**Table 8: SwiftForth Optional Packages**

| Group | Description |
|---|---|
| Generic Options | Extensions to SwiftForth that are not platform-dependent, such as floating point math, extensions to the object package, and various math functions. |
| Win32 Options | Windows-dependent extensions, such as various forms of dialog boxes and other kinds of controls, a console debugging feature, a simple DDE client, some graphics tools, etc. |
| Generic Samples | Some examples of floating point and other non-Windows coding tricks. |
| Win32 Samples | A rich collection of samples of code that exploit various Windows features, from TCP/IP communication to graphics, including the popular Sokoban and Tetris games. |

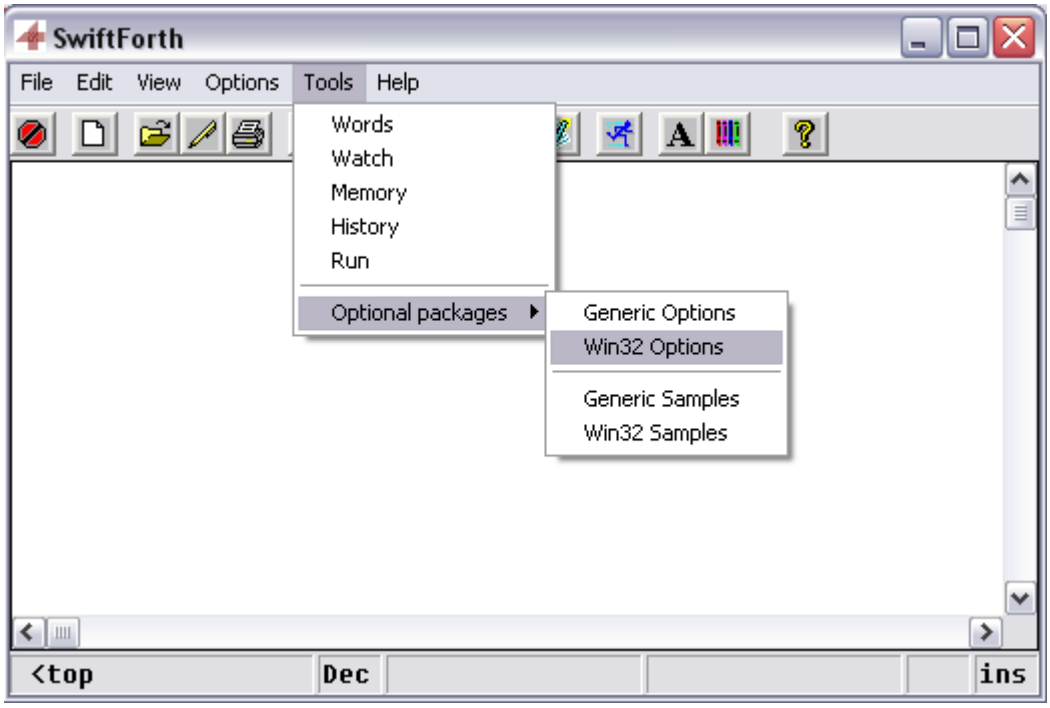


**Figure 9. Optional packages selection**

When you select a group, you will be presented a dialog box listing the packages available in that group, such as the one in Figure 10. Use your mouse or arrow key to move up and down in the list. As you select each item, you will be shown a brief description of it.

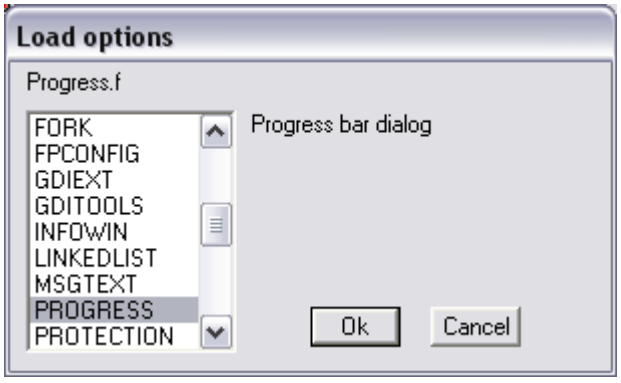


**Figure 10. Load options dialog**

The major items listed under Win32 Options are described in Table 9.

**Table 9: Some representative "Win32 Options"**

| Filename | Description |
|---|---|
| **BLKEDIT** | polyFORTH block editor (includes **PARTSMAP**). |
| **DDECLIENT** | API for DDE client services. |
| **FPMATH** | Hardware (Intel FPU) floating-point math library (Section Section 12:). |
| **PARTSMAP** | Block support (ANS Forth Blocks wordset plus extensions). |
| **PFASSEMBLER** | polyFORTH-compatible assembler (see polyFORTH documentation for details). |
| **PROTECTION** | Re-defines most common memory access words to ensure action within legally accessible memory. |
| **RANDOM** | Parametric multiplicative linear congruential random-number generator. |
| **RATIONALAPPX** | Rational approximations. |

In addition to these options, SwiftForth also includes the Forth Scientific Library, a collection of mathematical algorithms implemented in ANS Forth by a group of independent programmers. These routines may be found in the **SwiftForth\Unsupported\FSLib** directory, along with **Info.txt** which describes the library and its contributors. FORTH, Inc. provides this library as a service to SwiftForth programmers but, since we did not develop it, we cannot support it or assume any responsibility for it.

To save a SwiftForth system configured to your taste, load the options of your choice, then type:

```
PROGRAM <filename>
```

to record a turnkey version of the system as **filename.exe**.

This feature is not available in the SwiftForth evaluation version.

*References*

Watch points, Section 2.4.5.3
Memory window, Section 2.4.5.2
Command history window, Section 2.4.8
Using **PROGRAM** to make a turnkey image, Section 4.1.2
DDE Client Support, Section 8.3

### 2.3.7 Help Menu

The Help menus provide on-line documentation for your SwiftForth system.

**Table 10: Help menu options**

| Item | Content |
|---|---|
| Windows API | Opens the Windows API help system |
| MSDN Windows API Reference | MSDN Windows API reference web page |
| Handbook | *Forth Programmer's Handbook* (PDF version) |
| Reference Manual | This book, in PDF format. |
| ANS Forth Standard | The Standard approved in 1994 and ratified in 1999, in PDF format. |
| Go Online | FORTH, Inc. web site (www.forth.com) |
| Version History | Displays the release history |
| About | Product version number and related information. |

The Windows API help system, whose main component is **SwiftForth\doc\win32api.hlp**, can be used directly as a Help file. You may also search it from the SwiftForth command window using the form:

```
API <win-name>
```

…where *win-name* is the name of a Windows function, constant, message, etc. It will be located in the Windows API help system.

*Glossary*

**API** <win-name> *( — )*
Causes the Windows API Help system to display the entry for *win-name*.

## 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>
```

is equivalent to selecting a word in the command window (as discussed in Section 2.3.1) and selecting the Locate right mouse menu option. 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 how large your command window is just now; the display will use two-thirds of the available lines.

**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 pointer 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:

```
C:\ForthInc\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.

The actual amount of text displayed will depend on the size of your command window. SwiftForth calculates the number of lines to display based on the current debug window size

If you select a word in the command window, the popup menu generated by a right-click will offer (among others) the option Edit. Selecting that command will launch your linked editor (or switch to it if it is already open) with the cursor positioned on the source line containing the word. This feature lets you immediately edit the source, if you wish, and examine the rest of the file. (Use of the Options menu to link to an editor other than the default Notepad is described in Section 2.3.5.)

If the compiler encounters an error and aborts, you may directly view the file and line at which the error occurred by typing **L**. This is particularly convenient if you have a linked editor (see Section 2.3.5), because you can immediately repair the error and recompile. If you don't have a suitable editor, SwiftForth will display the source path and line number in the command window, and you will have to manually switch to your editor to fix the problem.

*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 range of lines following a **LOCATE** display.

**B** *( — )*

Display the previous (Back) range of lines 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.

**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.

**G** *( — )*

Following a compiler error, opens the linked editor with the cursor positioned on the line where the error occurred.

### 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). If the option "Use coloring for WORDS" is selected in the Options > Preferences dialog box (Section 2.3.5), typing **WORDS** will display the words color-coded to indicate their category as shown in Table 11, and you can double-click any such displayed word to select it and **LOCATE** its source or any of the other "right mouse button" actions described on page 22.

**Table 11: Color-coding in WORDS**

| Color | Category |
|---|---|
| Black | Default (no recognized category) |
| Blue | Inline words (i.e., words expanded to inline code) |
| Red | **CONSTANT**s |
| Green | **VARIABLE**s |
| Cyan | Menu items |
| Magenta | Immediate words |
| Dark yellow | User variables |
| Dark blue | **VALUE**s |

**Table 11: Color-coding in WORDS**

| Color | Category |
|---|---|
| Dark red | Wordlists |
| Dark green | WinProcs |
| Dark cyan | Windows functions |
| Dark magenta | Switches |

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*
3 words found. ok
```

In other words, these three words contained the sequence `M*` in their names.

If you type **BROWSE**, select the Tools > Words menu option, or press the Words toolbar button, the list of words will be displayed in a separate dialog box that you may keep open while you work.

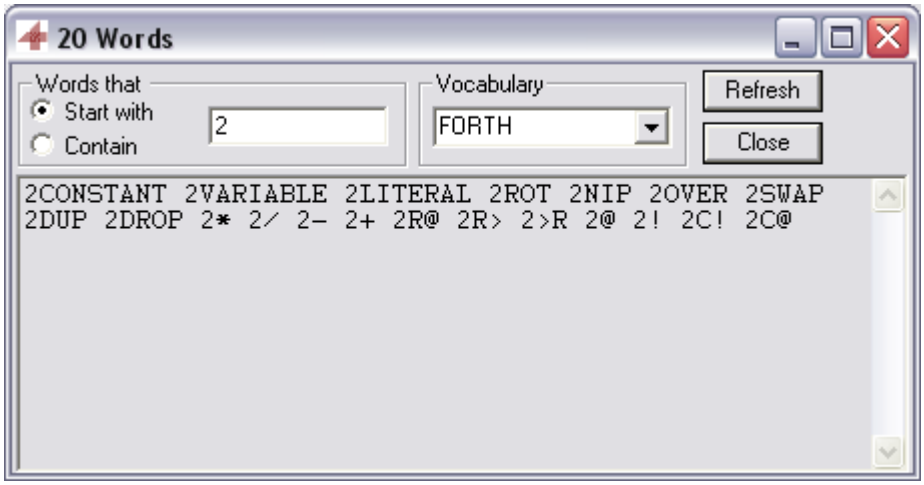


**Figure 11. Words display dialog**

The Words browser dialog box, shown in Figure 11, provides places to specify not only words *containing* a particular string, but also the ability to see words *beginning*

with a string. A drop-down list allows you to select from currently available vocabularies. The Refresh button will refresh the display after you change a selection parameter or add definitions.

*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 shortcut:

```
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 in the sample program **scribble.f**):

```
WH POINT#

WORDLIST: FORTH
C:\ForthInc\SwiftForth\lib\samples\win32\scribble.f
125  13| 0 VALUE POINT#
125  24| POINT# 1 > IF
125  27| POINT# 1- 0 DO
125  28| POINT# 1 DO
125  38| POINT# #POINTS < IF
125  39| LPARAM  POINT# CELLS POINTS + !
125  40| 1 +TO POINT#
125  87| SCRIBBLING ?EXIT  0 TO POINT#
```

The first line shows the wordlist name of the wordset in which the word was found. The next lines show where it was defined, and the lines following show all references to it. The numbers to the left of the vertical bar are line numbers in the file; if you click on one of these, you will select it, making it available for the "right mouse button" actions described on page 22.

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**. This will list all such definitions, as well as WINPROCs in open DLLs. As with the **WHERE** display, you may double-click the line numbers to select that source for further examination. Note that the fact that a word appears in this list doesn't *necessarily* mean you want to get rid of it.

*Glossary*

**WHERE** <name>, **WH** <name> *( — )*
Display a cross-reference showing the definition 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. Addresses and data are displayed in hex.

**IDUMP** *( addr u — )*
Displays *u* bytes, starting at *addr*, as 32-bit signed 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

#### 2.4.5.2 Dynamic Memory Display

You may launch a separate window to provide a dynamic view of a region of memory. This may be launched using Tools > Memory, or the equivalent Toolbar button or keyboard command **MEM**. If you launch it from the Tools menu or Toolbar button, it will display the beginning of the SwiftForth kernel (see Section 5.2 for a memory map).

From the keyboard, you may control the origin of this display by using the command:

```
<addr> MEM
```

...where *addr* is a memory address (as returned by SwiftForth data objects). All of memory that may be legally viewed under Windows is available to you. This includes the entire 4 GB range of virtual address space. Within this space are “islands” of virtual memory that Windows has allocated to SwiftForth for various purposes. Other spaces may be allocated differently at different times, but always in 4K chunks.

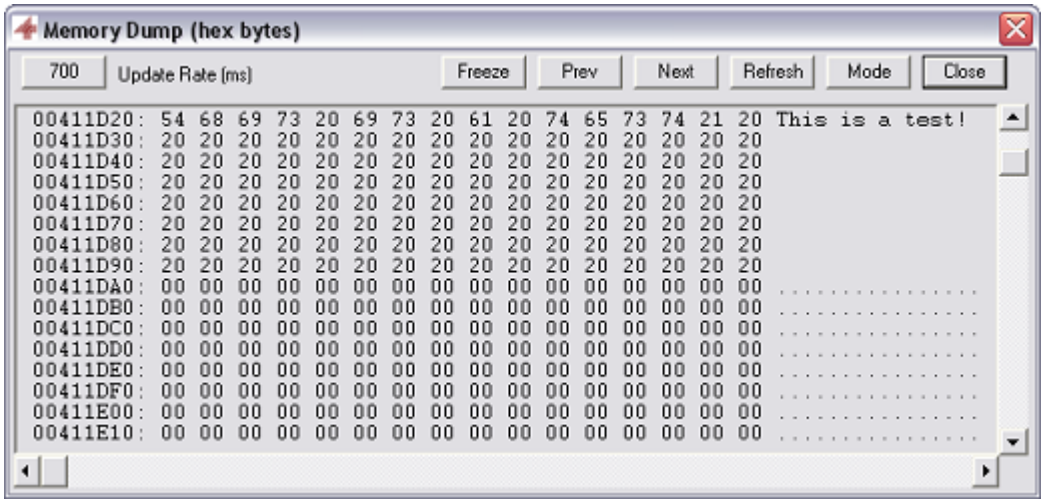


**Figure 12. Memory Dump Window**

The scroll bar on the right side of the Memory Dump window scrolls through available memory. A click on the up- or down-arrows moves 256 bytes; a click in the region above or below the thumb bar moves 4K bytes. If the window gets an address outside of viewable memory, you will see a message, “Invalid memory region selected at <addr>.”

If you wish, you may press the Next button to view the next higher island. This function will advance in 4K increments beyond the current visible island, skipping whatever inaccessible region may follow, to the start of the next island. The Prev button works similarly, but in the direction of low memory.

The horizontal scroll bar initially reflects where the starting address is on a 16-byte line, with the leftmost position corresponding to a hex address whose low-order digit is zero, and the rightmost position corresponding to a low-order digit of F. This scroll bar may be used to align the leftmost column on an even 16-byte boundary (i.e., a hex address ending in zero).

You may adjust the rate at which the display is updated by pressing the Update Rate button. This will toggle through the available rates. You may also freeze the display by pressing the Freeze button. When the display is frozen, you may update it by pressing the Refresh button.

A good way to try the Memory window is to request **`TIB MEM`**. This shows your terminal input buffer. You can type, and see your typing reflected in the display. Or you may request **`PAD MEM`** and then put things in **`PAD`** and watch the results.

The Memory Dump display has three viewing modes, which you may toggle through by pressing the Mode button. These are:

- Hex bytes (shown in Figure 12) with ASCII decoding
- Hex cells (used for viewing numeric data in hex)
- Decimal cells (used for viewing numeric data in decimal)

*Glossary*

**MEM** *( addr — )*

Start a window providing a dynamic view of a region of memory starting at *addr*. If a memory window is already open, sets its initial address to *addr*.

*References* **PAD**, *Forth Programmer's Handbook*
Memory organization in SwiftForth, Section 5.2

### 2.4.5.3 Watch Points

SwiftForth lets you set up a separate Memory Watcher window to dynamically monitor a list of selected memory regions. The individual locations being monitored are called *watch points*.

You may launch a watch window using the Tools > Watch menu item or Toolbar button; alternatively, one will be launched automatically by your first **WATCH** command.

To set an individual single-cell (four-byte) watch point, use the command **WATCH**, as follows:

```
<addr> WATCH
```

...where *addr* is a data space address.

If the address you supply was provided by a word defined by **VARIABLE** or **CREATE**, **WATCH** can display its name; if it was provided in some other way (e.g., **PAD** or the sequence **'** <name> **>BODY**), it will display only the address.

The watch point will use whatever your current number base is when you set it. For example, if you have defined a **VARIABLE** named **DATA**, and are in the default **DECIMAL** base when you type **DATA WATCH**, the contents of **DATA** will be monitored in decimal—even if you change bases later. If you type:

```
HEX PAD WATCH
```

the first four bytes of **PAD** will be monitored in hex.

To demonstrate this, type the following:

```
VARIABLE DATA
DATA WATCH
HEX PAD WATCH DECIMAL
```

Then observe what happens in your watch window when you type:

```
1000 DATA !
PAD 4 ACCEPT [cr] ABCD
```

You should see a display like the one in Figure 13

.

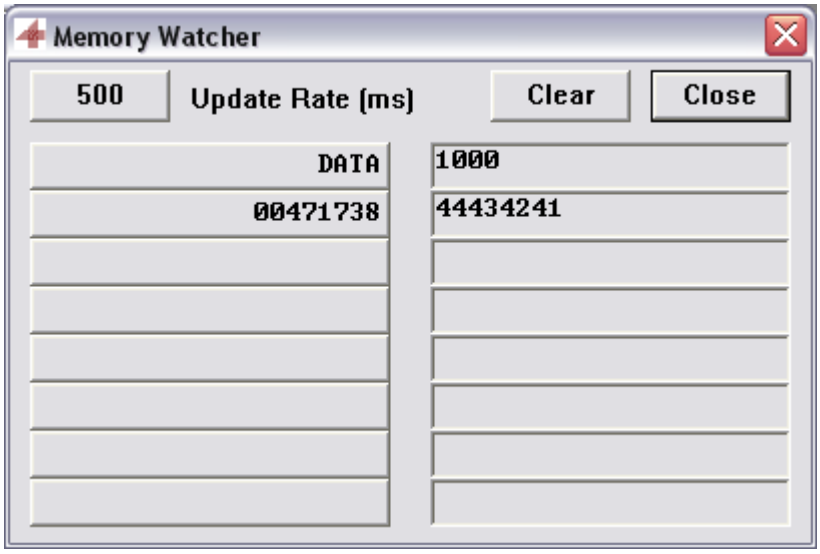


**Figure 13. Example of a watch window**

If you have dynamically changing data, for example in a separate thread, you may adjust the update rate of the watch window by pressing the update rate successively, until you get the desired rate.

To remove a watch point, just click on its name or address.

*Glossary*

**WATCH** *( addr — )*

Add *addr* to the list of cells being monitored as watch points, and launch the watch window if it is not already active. The contents of *addr* will be displayed in the number base that is current when **WATCH** is invoked.

### 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
```

### 2.4.7 Console Debugging Tool

The console debugging tool lets you get debugging information that is not available interactively—such as traces during execution of dialog box code or Windows callbacks—as the console is completely independent of the Windows GUI interface.

An example use is in the sample program **bugsample.f**, reproduced below:

```
REQUIRES CONSOLEBUG\ Include console debug routines

1 TO BUGME\ 0 means debug not active

: TEST
  [BUG CR ." TESTING" .S BUG]\ Anything output-oriented
  [BUG KEY DROP BUG]\ KEY for PAUSE also available
   DUP * DROP ;

: TRY
   10 0 DO
      I TEST
   LOOP ;
```

Any words between **[BUG** and **BUG]** will be executed if **BUGME** is not zero.

A common use of this feature is to display the parameters on each entry to a callback being debugged.

### 2.4.8 Managing the Command Window

The command window is implemented internally as a circular buffer; a long session may “wrap” this buffer, so early commands may be lost. Three of the File menu options allow you to record the events of a development session in various ways:

- **Save Command Window** records a snapshot of the present contents of the command window in a text file. This is useful if, for example, you have just encountered a strange behavior you would like to record for later analysis.
- **Save Keyboard History** records into a text file only the commands you’ve typed. Such a file may be edited, if you like, using your text editor. You can replay these commands by including the file. This is useful for developing scripts or for reproducing a bug.
- **Session Log** opens a file and starts recording everything that happens thereafter in the session, until you turn it off by re-selecting this menu item. While it is active, its

menu item displays a check mark.

You may display a window containing the keyboard history by selecting Options > History or the Toolbar button. You can edit the contents of this window, using it as a scratch area, and you may copy and paste selections from this window into the command window.

*References* File menu, Section 2.3.2

# Section 3: Source Management

Traditionally, Forth source and data were maintained in 1024-byte blocks on disk. This format was appropriate for early systems, many of which used Forth as the only operating system, with native disk drivers. In such an environment, a block number mapped directly to the physical head/track/sector location on disk, and was an extremely fast and reliable means of handling the disk. As more Forths 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 in 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 a linked editor of your choice as described in Section 2.4.1. This section describes tools for managing text files.

## 3.1 Interpreting Source Files

You may load source files using the File > Include menu option, the Include button on the Toolbar, or by typing:

```
INCLUDE <filename>
```

…in the command window. The functional difference between these is that the button or menu options display a browse window in which you can select the file, and leaves SwiftForth's path *set to that of the selected file*, whereas the typed command handles a specified file (including relative path information) and *doesn't affect your current path*.

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

```
INCLUDE HEXCALC
```

(where the filename extension **.f** is assumed if no extension is given).

The standard DOS/Windows rules for describing paths apply:

1. **Absolute path:** The name starts with a \ or <drive>:\ or \\<name>\. Any of these indicates an absolute path to the file. This type of path is typically discouraged, because it must be changed manually if the directory structure changes.
2. **Relative path to subdirectories:** 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 backslash). Each series raises the location one level from the current sub-

directory. After you have raised the level sufficiently, you can use the technique in #2 to go down subdirectory levels.

In addition, SwiftForth can manage paths relative to the location of the actual executable (normally in the **SwiftForth\bin** directory). Such paths are indicated by the starting symbol %, and the actual root is two levels above wherever the executable is. For example, the basic error handling functions are loaded by:

```
INCLUDE %SwiftForth\src\ide\win32\errmessages.f
```

If you have launched SwiftForth from a project file or shortcut in your project directory (described in Section 3.1), your default path is that directory, so you don't need to preface local files with any path information. So, 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 Windows command line, except there must be a space between **CD** and any following string. The **CD** command followed by *no* string will display your current path. No spaces are permitted in the pathname, and no other command may appear on the line. **CD** is not appropriate for use in definitions.

If you wish to change your directory path temporarily, you may take advantage of SwiftForth's *directory stack*. The word **PUSHPATH** pushes your current path information onto the directory stack, and **POPPATH** pops the top path from the directory stack to become the current path.

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**s throughout the source. 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. The directory specified by the environment variable **SFLOCAL_USER**
3. **%SwiftForth\lib**
4. **%SwiftForth\lib\options**
5. **%SwiftForth\lib\options\win32**
6. **%SwiftForth\lib\samples**
7. **%SwiftForth\lib\samples\win32**

The syntax is simply:

```
REQUIRES <filename>
```

The default filename extension is .f.

Another function helps you to avoid loading the same file more than once, and also makes it possible to add files to the lists displayed by the Tools >Optional Packages > Load Options dialogs.

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 not exceed 200 characters in length.

**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 contructed and the rest of the line will be ignored.

If, and only if, **OPTIONAL** is the first word of one the first 10 lines of a file that exists in one of the directories…

**%SwiftForth\lib\samples**
**%SwiftForth\lib\samples\win32**
**%SwiftForth\lib\options**
**%SwiftForth\lib\options\win32**

…*name* will appear in one of the **LOAD-OPTIONS** dialogs (selected via the Tools > Optional Packages menu), and the associated *description* will be displayed in that dialog box.

*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** differs from the File > Include menu option and Toolbar button in that it does not offer a browse dialog box and does not change your current path.

**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.

If this appears as the first word in the first line of a file that exists in one of the directories specified above (Section 3.1), *name* and *description* will appear in one of the Load Options dialogs.

**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 to wish to have commentary 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 braces as functionally equivalent to parentheses except for taking a terminating brace. So 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 by using a flexible utility enabled by the command **VERBOSE** and disabled by **SILENT**. The level of monitoring is controlled by the dialog box, shown in Figure 14, that can be invoked by using the Options > Include monitoring menu item. The default behavior is "Display the text of each line."

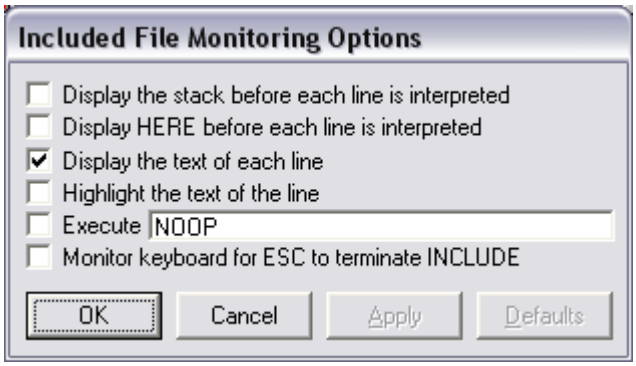


**Figure 14. Included file monitoring configuration**

For 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 launch of **Sf.exe.** Any application definitions that you don't want discarded with the overlay should be loaded as system options (see Section 2.3.5).

**EMPTY** discards all definitions and releases all allocated data 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.

In particular, any time you have passed an address to Windows, such as a callback, you must provide a mechanism for un-doing that connection before you discard the code containing this address.

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.

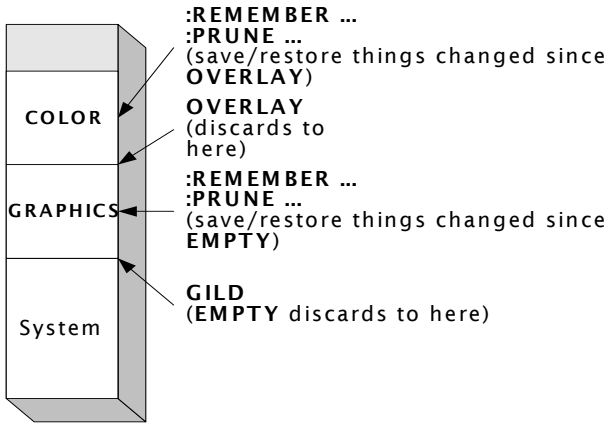


**Figure 15. Multiple overlays**

The following example (and Figure 15) 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>[.<ext>] [<icon>]
```

This records a "snapshot" of the current running system in the current path. The file extension is optional. If it is omitted, **.exe** will be assumed and a Windows executable file will be built.

The optional icon specification lets you associate an icon with this executable that is different from the standard SwiftForth icon. The icon must have a file format of exactly 32x32 pixels, 16 colors, and must contain exactly one icon. You specify it by giving its filename, which must have the extension **.ico**.

This feature is not available in the evaluation version of SwiftForth.

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 Windows application, however, requires more detailed attention to startup issues. These issues are discussed further in Section 8.7.

*Glossary*

**PROGRAM** <filename>[.<ext>] [<icon>] *( — )*

Record a "snapshot" of the current running system in the current path (or in the path specified with *filename*). If the optional file extension is omitted, **.exe** is assumed and a Windows executable file will be built. If an optional icon is specified, it must be the name of a file whose format is 32x32 pixels, 16 colors, and exactly one icon. The icon file must end with the extension **.ico**.

## 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 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 a Windows 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 temporarily. All standard Forth words are in upper case in SwiftForth, as are most SwiftForth words. Calls to Windows procedures are in mixed case, as shown in Windows documentation; these calls are *case sensitive* regardless of the state of the SwiftForth option.

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 Forth, however we strongly recommend against operating in this mode in general, as unintended conflicts may be introduced that are difficult to find and debug.

### 4.2.2 Detecting Name Conflicts

Since Forth is extremely modular, there are very many words and word names. As shipped, SwiftForth has over 3,500 named words, plus the Windows constants and procedures that are callable by other means. As a result, programmers are understandably concerned about inadvertently *burying* a name with a new one.

Insofar as possible, SwiftForth factors 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**s enabled. General control of this feature is in the dialog box Options > Warnings (see Section 2.3.5).

*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] \ Reports memory use
```

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 12. 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 12: 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 12:):

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

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 12: 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 Swift-

Forth. 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.
- **(WINLONGDATE)** provides a Windows long date, e.g., Wednesday, June 3, 1998.
- **(WINSHORTDATE)** provides the Windows short date format, e.g., 6/3/98.

**(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), $u_6$ year (1900–2079).

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

Convert the stack arguments $u_1$ seconds (0–59), $u_2$ minutes (0–59), $u_3$ hours (0–23), $u_4$ day (1–31), $u_5$ month (1–12), $u_6$ year (1900–2079) to internal form and store them as the system date and time.

### 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-MMM-YYYY)** *( $u_1$ — *addr* $u_2$ )*

Format the MJD $u_1$ as a string with the format `dd-MMM-yyyy, where MMM is a three-letter month abbreviation`, returning the address and length of the string. This is the default behavior of **(DATE)**.

**(WINLONGDATE)** *( $u_1$ — *addr* $u_2$ )*

Format the MJD $u_1$ as a string using the Windows long date format. This is a possible alternative behavior for **(DATE)**.

**(WINSHORTDATE)** *( $u_1$ — *addr* $u_2$ )*

Format the MJD $u_1$ as a string using the Windows short date format. This is a possible alternative behavior for **(DATE)**.

**.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 Windows timer, the accuracy of the measured interval depends upon the overall Windows multitasking environment. In general, the error on an interval will be on the order of the duration of a clock tick; however, it can be much longer if a privileged operation is running. The exact distribution of errors is highly subject to the behavior of other Windows programs that may be running.

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
Multitasking impact of **MS**, Section 7.2.3

## 4.5 Specialized Program and Data Structures

The complex needs of the Windows environment have 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 (including the

kind of transient task instantiated by a callback) 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. Beacuse **PAD** is in the user area, each task or callback has a private version. **PAD** is defined relative to **HERE**, and so will move if additional memory is allotted for definitions or data.

**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
User areas, Section 7.2.1
Callbacks, Section 8.1.2

### 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. Windows uses a standard string format that is terminated by a binary zero, referred to as an ASCIIZ string. Also, Unicode strings are required in a few places. In this implementation, 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 13 are supported. Note that characters following the \ are case-sensitive.

The string-management extensions in SwiftForth are documented below.

**Table 13: Character sequence transformations**

| Character sequence | Compiled byte(s), hex | Description |
|---|---|---|
| \\ | 5C | backslash |

**Table 13: Character sequence transformations *(continued)***

| Character sequence | Compiled byte(s), hex | Description |
|---|---|---|
| **\"** | 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 | CR |
| **\t** | 09 | horizontal tab |
| **\v** | 0B | vertical tab |
| **\xcc** | *cc* | General constant, expressed ashex *cc* |
| **\z** | 00 | null |

*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 13.

**,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 13. (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 13.

**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 13.

**.\"** *( — )*

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 13.

**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 13.

**,"** `<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 13. 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 both in a **.exe** program and a DLL), 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* are used to process Windows messages, Forth **THROW** codes, and other encoded information for which specific responses are required.

Message switches are defined using this syntax:

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

where there are two possible **RUN**x words:

- The word **RUNS** followed by a previously defined word will set that switch entry to execute the word.
- 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 may 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 can be very useful in Windows, as the number of parameters used to manage Windows structures can be large. 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.

In addition, you may configure the way in which both warnings and error messages (from **THROW**) are displayed: either in the command window, in a separate dialog box, or both. (Even if warnings are disabled, error messages will always be displayed.) This configuration is handled by the Options > Warnings menu selection, discussed in Section 2.3.5.

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. Typically, **CATCH** is 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 issuing an error message in a dialog box that must be acknowledged. 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 the Words dialog box specifying "Words that start with" **IOR_**; to see them all, set the Vocabulary to *All*.

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 Windows exception handler. When a processor exception is generated, SwiftForth will generate a **THROW**. If your application **CATCH**es such an exception, you can handle it like any other SwiftForth exception. If you wish additional information, you may call **WINERROR** to get the address and length of the text string describing the exception. The string

returned matches the Win32 API standard status message names, except that the SwiftForth strings omit a preceding **STATUS_** prefix. In other words, the Win32 exception that returns the code **STATUS_ACCESS_VIOLATION** will, in SwiftForth, return the string **ACCESS_VIOLATION**.

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

SwiftForth's standard error handler can be configured (as described in Section 2.3.5) to display a simple dialog box, a message in the command window, or both. For example, if you type **0 @** (an illegal attempt to read memory location zero) you'll get an **ACCESS_VIOLATION** processor exception. The resulting dialog box would look like the one shown in Figure 16. An internal SwiftForth error (e.g., stack underflow or application-generated exception) would be the same, except that the register portion of the Details button is valid only for Windows exceptions.


**Figure 16. Dialog box from a Windows exception**

If you press the Details button, the dialog box is extended to show the processor registers, the top eight data stack items, and the top eight return stack items. SwiftForth attempts to translate the return stack addresses, to provide additional debugging information. The result is shown in Figure 17.

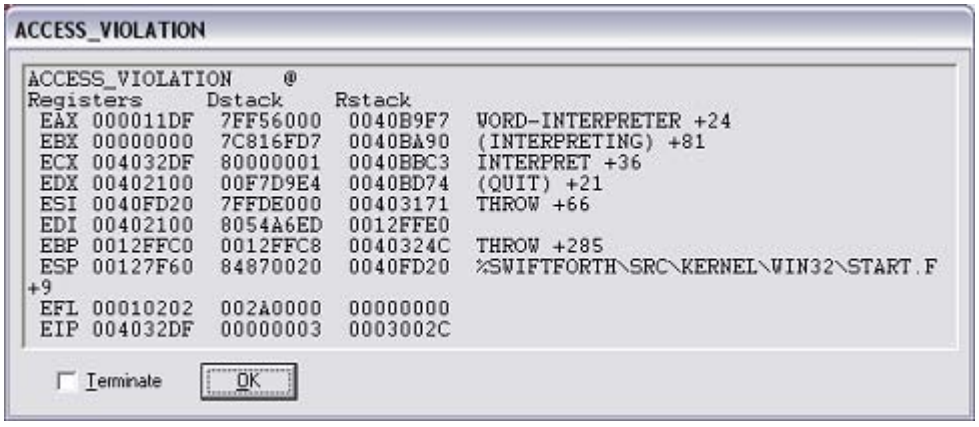


**Figure 17. Processor Exception dialog box**

On any exception, if you check the "Terminate" box, pressing the OK button will

close SwiftForth. Otherwise, it will cause an internal abort, clearing both stacks and returning control to the command window. This will usually work, except in the case of a very serious violation (e.g., a deep return stack underflow).

*Glossary*

**>THROW** *( n addr u — n )*

Associate the string *addr u* with the **THROW** code *n* 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.

**WINERROR** *( — addr u )*

Following a Windows exception caught by **CATCH**, returns the address and length of the string provided by Windows identifying the exception.

## 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 **\Unsupported\Anstest**. To run it, type:

```
INCLUDE anstest.f.
```

Sample output is shown in **Anstest.txt**.

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 14 summarizes SwiftForth support for the wordsets defined in ANS Forth.

**Table 14: SwiftForth support for Standard Forth wordsets**

| Wordset | Support provided |
|---|---|
| **CORE** | All words provided. |
| **CORE EXT** | All words provided except **EXPECT**, **SPAN**, **[COMPILE]**. **EXPECT** and **SPAN** are marked "obsolescent." |
| **BLOCK** | All words provided. |
| **BLOCK EXT** | All words provided. 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. |

**Table 14: SwiftForth support for Standard Forth wordsets *(continued)***

| Wordset | Support 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 a Windows 32-bit environment. 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, subroutine-threaded Forth running as a GUI application under the Win32 subsystem of Windows 95 and later.

*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 code inline 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 over 200 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 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 address of the start 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 used 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 in Windows virtual memory. SwiftForth is position-independent, which means it can run wherever Windows has instantiated 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 Windows, if desired, without change. All references in this book to "addresses" as stack arguments refer to these full, absolute addresses.

By contrast, some Windows Forths keep the absolute address of the beginning of the assigned memory area in a register, and internally use addresses assigned from a zero base. When an address is passed to Windows, the register value must be added to the internal address; when Windows passes it an address, the register value must be subtracted to get the internal address.

By being position-independent, SwiftForth simplifies and speeds up all Windows interactions. This feature is a natural consequence of subroutine-threading, since 80x86 calls use relative addresses. Forth *execution tokens* are zero-relative to the start of the run-time memory space, but they are used only internally and, thus, do not need conversion.

SwiftForth and turnkey executable programs made from SwiftForth are always instantiated into the same virtual address space. However, DLLs made from SwiftForth cannot control where they are instantiated.

Therefore, 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 DLL. Instead, you should always *execute the object name at run time to get the address*. The only situation in which this is difficult is in assembler code; Section 6.5.5 discusses ways of doing this if you really need to compile or permanently store data addresses.

*References* Creating DLLs from SwiftForth, Section 8.2.2
**PROTECTION** option (absolute address warnings), Section 2.3.5

### 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 Windows. 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. This is because the names of Windows functions can be quite long, and because it is desirable to be able to keep long filenames as normal dictionary entries.

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 Windows-provided memory space whose size is fixed when SwiftForth is loaded. The default size is 4 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**. SwiftForth's loader ensures that it is always instantiated at a fixed address. The dictionary grows toward high memory.

The general layout of SwiftForth's memory is shown in Figure 18.

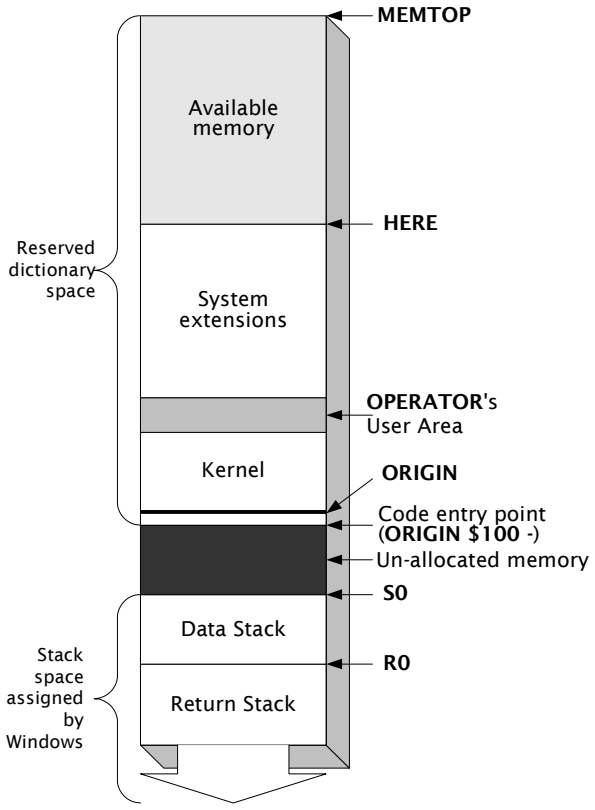


**Figure 18. SwiftForth memory map**

The Standard Forth command **UNUSED** returns on the stack 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.

A total of 1 MB of stack space is allocated by Windows. SwiftForth reserves the top portion of this for its data stack, and the balance is left for the return stack. The return stack is the processor stack, which is used extensively by Windows.

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. Additional tasks each get a private 1 MB space for user variables and stacks; this space is completely independent of SwiftForth's dictionary allocation.

*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.

**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 ;
```

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

**+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 Windows' virtual memory pool, which is outside SwiftForth's data space.

This is a particularly useful strategy for large buffers, because it avoids enlarging the size of a saved program (which would be the result of a **CREATE … ALLOT** sequence). It's also a useful strategy for assigning private data space to each instance of a window; the handle may be stored in the window data structure.

Virtual memory **ALLOCATE**d by your program does not have to be explicitly released, as it will be automatically released when the program terminates. However, memory **ALLOCATE**d by a window must be released when the window is destroyed.

**ALLOCATE**d memory is not instantiated until you write to it.

*References* Temporary memory allocation for callback use, Section 8.1.2

## 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.

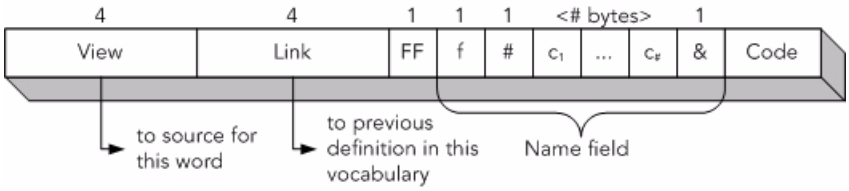


**Figure 19. Dictionary header fields**

Headers are laid out as shown in Figure 19. 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 20).
- # 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.

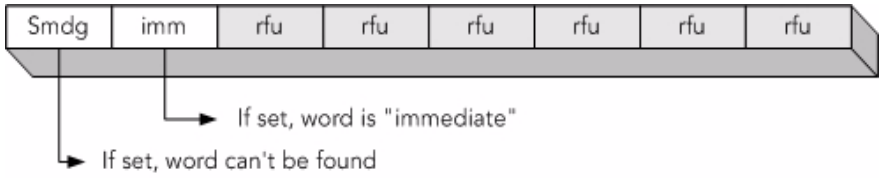


**Figure 20. Structure of the "flags" byte**

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

Forth provides for the dictionary to be organized in multiple linked lists. This serves several purposes:

- to shorten dictionary searches
- to allow names to be used in different contexts with different meanings (as often occurs in human languages)
- to allow internal words to be protected from inappropriate or unintentional use or redefinition
- to enable 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 top of the search order. A vocabulary is defined using the word **VOCABULARY** (also discussed in *Forth Programmer's Handbook*).

Normally, 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 by using it as the argument to **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 have no interest to the "outside" world.

Packages are a technique for encapsulating 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 hidden (*private*) in order to keep dictionary searches short and avoid name collisions. 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 *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 application, but are considered not to be generally useful or interesting. 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."

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

```
PACKAGE MYAPPLICATION
```

The word **PACKAGE** will create a new package only if it doesn't already exist; so 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 Windows Constants

Windows programming uses a large number of named constants, such as **GENERIC_READ**. The system would be burdened to include all of these as Forth definitions. The SwiftForth compiler contains a link to a file with access to the full list, called **WINCON.DLL** (included with SwiftForth), 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 Windows constants are compiled as literals, so after compilation is complete, **WINCON.DLL** is no longer needed. It is also not required by non-extensible turnkey applications.

### 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 special Windows function wordlist **KNOWN-PROCS** (see Section 5.5.2).
5. Check **WINCON.DLL**.
6. Abort, with the error message `<unknown-word> ?`

*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 serial-type destinations; all the serial 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 command 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 or callback 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 15. 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 15: 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). |
| **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. |

**Table 15: Terminal personality elements *(continued)***

| Item | Stack | Description |
|---|---|---|
| **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. |

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 ;

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

                        \ Vector     Offset (cells)
    ' NOOP ,            \ INVOKE( -- )                  0
    ' NOOP ,            \ REVOKE( -- )1
    ' NOOP ,            \ /INPUT( -- )2
    ' DROP ,            \ EMIT  ( char -- )3
    ' 2DROP ,           \ TYPE  ( addr len -- )4
    ' 2DROP ,           \ ?TYPE ( addr len -- )5
    ' DROP ,            \ CR    ( -- )6
    ' NOOP ,            \ PAGE  ( -- )7
    ' DROP ,            \ ATTRIBUTE( n -- )8
    ' NULL ,            \ KEY   ( -- char )9
    ' NULL ,            \ KEY?  ( -- flag )10
    ' NULL ,            \ EKEY  ( -- echar )11
    ' NULL ,            \ EKEY? ( -- flag )12
    ' NULL ,            \ AKEY  ( -- char )13
    ' 2DROP ,           \ PUSHTEXT( addr len -- )14
    ' 2DROP ,           \ AT-XY ( x y -- )15
```

```
' 2NULL ,            \ GET-XY( -- x y )16
' 2NULL ,            \ GET-SIZE( -- x y )17
```

An example is in **SwiftForth\src\ide\win32\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 a document in full before acting on it. For example, SwiftForth uses it to generate the "details" display of an exception dialog box (Section 4.7). It's necessary to generate the entire output picture of the registers, etc., then use that text for a dialog control, because the program must be running in the Forth context to generate the output, but the dialog context could display pre-written text only.

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.

Another example featuring a personality that performs I/O on a serial COM port may be found in **SwiftForth\lib\samples\win32\serial.f**.

*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

**Table 16: VK_ codes recognized by SwiftForth**

| Constant | Value | Constant | Value |
|---|---|---|---|
| VK_PAUSE | $13 | VK_F7 | $76 |
| VK_PRIOR | $21 | VK_F8 | $77 |
| VK_NEXT | $22 | VK_F9 | $78 |
| VK_END | $23 | VK_F10 | $79 |
| VK_HOME | $24 | VK_F11 | $7A |
| VK_LEFT | $25 | VK_F12 | $7B |
| VK_UP | $26 | VK_F13 | $7C |
| VK_RIGHT | $27 | VK_F14 | $7D |
| VK_DOWN | $28 | VK_F15 | $7E |
| VK_SNAPSHOT | $2C | VK_F16 | $7F |
| VK_INSERT | $2D | VK_F17 | $80 |
| VK_DELETE | $2E | VK_F18 | $81 |
| VK_F1 | $70 | VK_F19 | $82 |
| VK_F2 | $71 | VK_F20 | $83 |
| VK_F3 | $72 | VK_F21 | $84 |
| VK_F4 | $73 | VK_F22 | $85 |
| VK_F5 | $74 | VK_F23 | $86 |
| VK_F6 | $75 | VK_F24 | $87 |

The words **EKEY** and **EKEY?** return *keyboard events*, beyond simple characters. There are, in fact, four keyboard-related messages Windows can generate:

- **WM_SYSKEYDOWN** occurs when a key of interest to the system (e.g., a key combination such as Alt-F that is associated with a menu) has been pressed.
- **WM_SYSCHAR** may occur along with a **WM_SYSKEYDOWN**, to deliver a system character.
- **WM_KEYDOWN** occurs when a key that is not a system key is pressed (such as a function key that has no menu association).
- **WM_CHAR** may occur along with a **WM_KEYDOWN** to deliver a character.

Some keypresses, like At-Tab, are completely processed by Windows, and will never be passed to a program. System key events must be dispatched to the owning window; SwiftForth programs can process them locally, as well. **WM_KEYDOWN** events not accompanied by **WM_CHAR** events are processed immediately; simple characters are placed in an input queue and can be retrieved by **KEY** or **EKEY**.

Windows defines *virtual keys*, 16-bit combinations whose low-order byte contains an ASCII code and whose high-order byte contains bits indicating the state of the Alt, Shift, and Ctrl keys. These *named Windows constants* have names in the form **VK_**<xxx>. SwiftForth recognizes the **VK_**<xxx> keystrokes shown in Table 16. If the message parameter identifies a key in this table, **EKEY** returns the exact Windows

representation of its associated constant; otherwise, it synthesizes a value according to Figure 21.

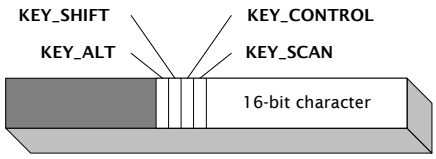


**Figure 21. Character encoding for EKEY**

The 16-bit character space in the low-order half of the cell contains whatever information is provided with the character itself, while the next nibble is decoded using four named masks that indicate whether it is a function key (**KEY_SCAN** is *true*) and/or is accompanied by a Control, Shift, or Alt key.

*Glossary*

**KNOWN-VKEYS** *( — addr )*

Return the address of a table of **VKEYS**. If an incoming <xxx>**KEYDOWN** event matches an entry in this table, it will not generate a **WM_CHAR** message, and SwiftForth will return the value from the table, masked as appropriate with the bits shown in Figure 21.

*References* Windows constants in SwiftForth, Section 5.5.3
Windows message handling, Section 8.1.3

### 5.6.3 Printer Support

Output via standard Windows printer functions is supported. Any console output may be redirected to a Windows printer or to an arbitrary text file. For example:

```
>PRINT HERE 100 DUMP
```

…will prompt the user to select a printer and print the memory dump on it.

*Glossary*

**>PRINT** <commands> *( — )*

Execute whatever commands follow to the end of the line, routing any output to the current printer.

### 5.6.4 Serial Port Support

Serial ports (e.g., COM1) can be read or written, using a small library of words provided that are equivalent to **KEY?**, **KEY**, and **EMIT**. See the source file **SwiftForth\lib\options\win32\sio.f** for details.

## 5.7 Timer Support

Windows supports four different timers:

- **System clock**, which responds to the function **GetSystemTicks**. This clock is used by the SwiftForth words **COUNTER** and **TIMER**, and has a resolution of about 1–55 ms., depending on hardware and system configuration.
- **Sleep timer**, which controls the interval that a task is inactive when it executes the **Sleep** function. Its resolution is about one millisecond.
- **High-performance timer**, which operates at about 1 MHz or faster. This is supported by the SwiftForth words **uCOUNTER** and **uTIMER**.
- **Multimedia timer**, for which SwiftForth provides no built-in support. However, it is easy to add the WINPROCs if you wish to use it, following the instructions in Section 8.2.

It is important to remember, however, that timing in Windows 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
Sleep timer, Section 7.2.3
Interfacing to WINPROCs, Section 8.2

## 5.8 Custom I/O Drivers

Windows does not encourage the development of custom I/O drivers. Under Windows 95 and 98, you can read and write I/O registers directly, although some system overhead is imposed. Interrupts are not supported. Under Windows NT, direct I/O reads and writes are prohibited.

However, SwiftForth includes a special third-party driver that permits you to develop custom I/O drivers, including interrupt handlers, for all Win32 systems. It is installed in the **SwiftForth\bin** directory. To use it, load the file **SwiftForth\lib\samples\portio.f**, which also includes documentation for the API.

Please note that response time is never guaranteed under the Windows environment, and that FORTH, Inc. cannot be held responsible for programs using third-party drivers.

# 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 20 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 22

.

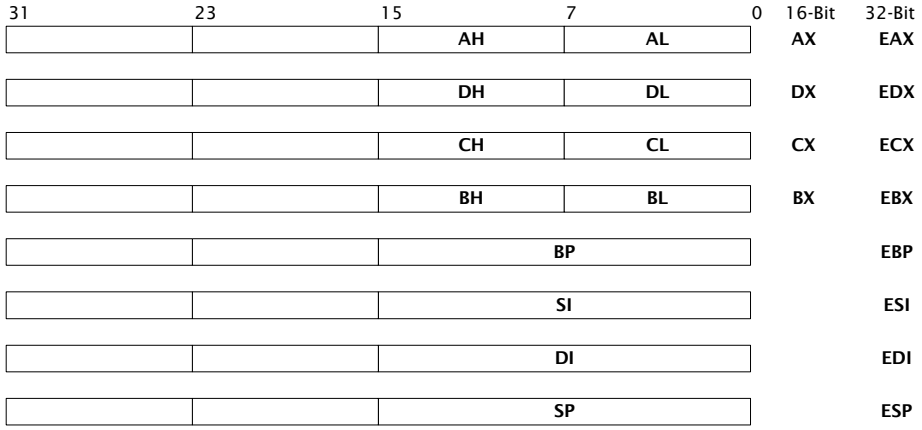


**Figure 22. General registers**

Neither the segment registers nor the status and control registers (not shown here) may be used directly under Windows.

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). The names of these registers are derived from the names of the general registers on the 8086 processor, the **AX**, **BX**, **CX**, **DX**, **BP**, **SP**, **SI**, and **DI** registers. As Figure 22 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, 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 6.5.5.2

## 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 Windows programming. Under Windows 95, all I/O is virtual, and it is not allowed at all under Windows NT. Instead, one does I/O using Windows calls (for standard devices or ports) or special drivers supplied as DLLs.

*References* Calling DLLs from SwiftForth, Section 8.2.1

### 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 by Windows. 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.

**VARIABLE**s 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 DLLs may be instantiated in different places at different times. This means that code that might *ever* be used in a DLL must

avoid compiling references to absolute addresses. If you are writing code that may ever be used inside a DLL, we recommend that you test your code using the optional **PROTECTION** option described in Section 8.2.2.

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
Creating DLLs from SwiftForth, Section 8.2.2
**PROTECTION** option (absolute address warnings), Section 2.3.5

### 6.5.5.2 Addressing with an Offset

Offset addressing combines parameters that are added to a base or index register, with other parameters, as shown in Figure 23.

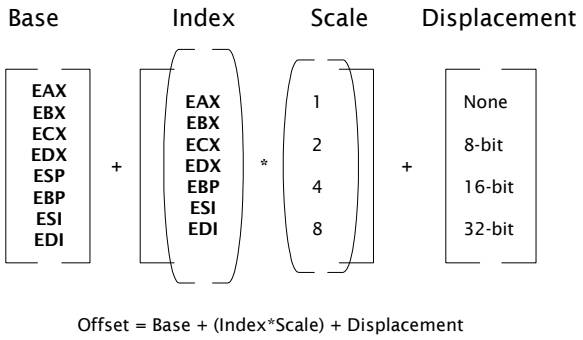


**Figure 23. Offset (or effective address) computation**

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 17.

**Table 17: Forms for scaled indexing**

| Scale (from Figure 23) | 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**.

#### 6.5.5.3 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 MOV**
  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**).

#### 6.5.5.4 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 *userarea*.

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 used in Windows applications.)

### 6.6.1 Size Specifiers

The size specifiers defined in SwiftForth are listed in Table 18. 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 18: 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 19, it will repeat until one of the terminating conditions is encountered.

**Table 19: 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 113. 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 20 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 20: SwiftForth condition codes and conditional jumps**

| Intel Instruction | SwiftForth Condition (with IF or UNTIL) |
|---|---|
| JA | **U> NOT** |
| JAE | **U<** |
| JAE | **CS** |
| JB | **U< NOT** |
| JECXZ | **ECXNZ** |
| JE | **0= NOT** |
| JZ | **0= NOT** |
| JG | **> NOT** |
| JGE | **0> NOT** |
| JL | **< NOT** |
| JLE | **S>** |
| JLE | **0>** |
| JNA | **U>** |
| JNAE | **U< NOT** |
| JNAE | **CS NOT** |
| JNB | **U<** |
| JNB | **CS** |
| JNBE | **U> NOT** |
| JNC | **U<** |

| Intel Instruction | SwiftForth Condition (with IF or UNTIL) |
|---|---|
| JBE | **U>** |
| JC | **U< NOT** |
| JC | **CS NOT** |
| JCXZ | **ECXNZ** |
| JNE | **0=** |
| JNG | **>** |
| JNG | **0>** |
| JNGE | **< NOT** |
| JNL | **<** |
| JNLE | **> 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 113.

*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 6.5.5.3). 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
    RET 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.

### Example 3: Data address computation

```
: ADDR ( addr reg -- )
    DUP R32? [FORTH] 0= THROW >R  [ASM]
    R@ EAX = [FORTH] IF [ASSEMBLER]
    ORIGIN 5 - CALL
    ORIGIN - # EAX ADD
    [FORTH] ELSE [+ASSEMBLER]
    EAX PUSH
    ORIGIN 5 - CALL
    ORIGIN - [EAX] R@ LEA
    EAX POP
    [FORTH] THEN [+ASSEMBLER] R> DROP ;
```

This is the macro described in Section 6.5.5.1 for getting the address of a data object, adjusting for the fact that data objects normally return an absolute address that is dependent on where Windows positioned SwiftForth in address space during this session. To do this, it is necessary to obtain the address given by the call to **ORIGIN 5 -** and compute the difference between that and *addr*, the address returned by invoking the name of the data item.

The major issue highlighted here is the use of **[FORTH]** and **[ASSEMBLER]** to manipulate addresses within a definition, so that the *high-level* versions of **0=**, **IF**, and **THEN** are used, but assembler words are used elsewhere.

*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 and Windows

Windows is an inherently multitasked environment. The actual task management algorithms vary between variants of the basic Win32 model, but the interface from the perspective of a program such as SwiftForth and its applications is the same.

The most important fundamental concept is that Windows is an event-driven environment. An event may be I/O (e.g., receipt of a keystroke or mouse click), receipt of a message from another window or program, or an event generated by the OS itself.

## 7.1 Basic Concepts

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

### 7.1.1 Definitions

Multitasking under Windows environments works at several levels:

- **Multiple applications** (e.g., Word, Excel, Firefox, SwiftForth) can be launched 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 DLLs. They might also communicate with one another, but they are still separate programs. An application is frequently called a *process*.
- **Multiple threads** can be launched from within a process. A thread is a piece of code (often a loop) that the Windows OS can be asked to execute separately. The thread uses the resources of the process that launched (and "owns") it, meaning it is executing code that resides in the owner's memory, using its registers, etc. However, the code is running asynchronously to the owning application, with the Windows OS controlling the scheduling according to its priority with respect to others. The owning process launches the thread and sets its priority; it can start and stop it, or kill it.
- **Multiple windows** can be launched by a process (or by one of its threads), as well. These are considered *child windows*. A child window has a procedure (`WindowProc`) that it executes, which is in many respects like an interrupt. It must start up, initialize itself, perform whatever the event causing the interrupt demands, and exit. The `WindowProc` is not permitted to have an ongoing behavior or loop, it runs solely in response to events and messages.

A SwiftForth program can support both multiple threads and multiple windows. 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, since the Windows 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 Windows applications, such as Word or Excel, you can also launch multiple instances of SwiftForth. These function as completely independent programs, each of which might have its own threads and/or windows. It is relatively easy for one SwiftForth program to send messages to another.

### 7.1.2 Forth Reentrancy and Multitasking

When more than one task can share a piece of code, that code can be called *reentrant*. Reentrancy is valuable in a multitasking system, because it facilitates shared code and simplifies inter-task communication.

Routines that are not reentrant are those containing elements subject to change while the program runs. Thus, self-modifying code is not reentrant. Routines that use *global variables* are not reentrant.

Forth routines can be made completely reentrant 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 reentrancy, by providing private versions of these variables to each task; such variables are called *user variables*.

Since reentrancy 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 Windows 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 will be 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 Windows, containing its data and return stacks and user variables. SwiftForth may read and write a task's user variables, but cannot 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 pri-

vate 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 particular 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 reentrancy, 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 CHUCK BASE HIS !
```

will set the user variable **BASE** of the task named **CHUCK** 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 **CHUCK**.

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** *( $addr_1$ $addr_2$ — $addr_3$ )*

Given a task address $addr_1$ and user variable $addr_2$, 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.

#### 7.2.2.1 Facility variables

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 which is running **GET**.

**GET** checks a facility repeatedly until it is available. **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.

**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.

*Glossary*

**GET** *( addr — )*

Obtain control of the facility variable at *addr*. 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 does nothing.

#### 7.2.2.2 Critical sections

Occasionally, it is necessary to perform a sequence of operations that cannot be interrupted by other SwiftForth tasks. Such a sequence is a *critical section*, and there are Windows functions to ensure that a critical section can be performed without interruption. SwiftForth's API to this is in the form of a pair of words, **[C** and **C]**, which begin and end a critical section. No other SwiftForth task will be permitted to run during the execution of whatever functions lie between these words. Note the use of critical sections in the definitions of **GET** and **RELEASE** above.

*Glossary*

**[C** *( — )*

Begin a *critical section*. Other SwiftForth tasks cannot execute during a critical section.

**C]** *( — )*

Terminate a *critical section*.

#### 7.2.2.3 Avoiding deadlocks

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.

### 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, Windows 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. The minimum value for *size* is 4,096 bytes; a typical value is 8,192 bytes. The task's return stack, which is also used for Windows calls, is always 16,384 bytes. When invoked, *taskname* will return 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 (or until reactivated, killed or halted by another task).

Transitory behavior may be terminated by calling the word **TERMINATE** or simply by returning, in which case SwiftForth will automatically terminate it. A task that has terminated in this fashion may be activated again, to perform the same or a different transitory behavior.

Persistent behavior must include the infinite loop and, within that loop, provision must be made for the task to relinquish the CPU using **PAUSE** or **STOP**, or a word that calls one of these (such as **MS**). These words are discussed in the glossary below. If this is not done, the task cannot be halted or killed and the process that owns it will not terminate properly.

A task that **ACTIVATE**s another task is that task's *owner*. A task may **SUSPEND** another task; **RESUME** it, if it has been **SUSPEND**ed; or **KILL** (uninstantiate) it. A task that is **SUSPEND**ed will always **RESUME** at the point at which it was **SUSPEND**ed.

A task may also **HALT** another task, which causes it to cease operation permanently the next time it executes **STOP** or **PAUSE**, but leaves it instantiated. The operational distinction between **HALT** and **KILL** is that the task remains instantiated after **HALT**, when it is **ACTIVATE**d again it will remember any settings in its user variables not directly affected by the task control words.

A task might manage one or more windows. If it does, it must frequently check its *message queue* and process any pending messages. Message management is described in Section 8.

*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 user area and dictionary of the task whose TCB is at *addr* and leave a pointer to the task's memory in the first cell of the TCB. If the task has already been instantiated (i.e. the first cell of the TCB 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* (if not already done by **CONSTRUCT**), 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** or **STOP** within the loop so task control can function properly.

**TERMINATE** *( — )*
Causes the task executing this word to cease operation and release its memory. A task that terminates itself may be reactivated.

**SUSPEND** *( addr — ior)*
Force the task whose TCB is at *addr* to suspend operation indefinitely.

**RESUME** *( addr — ior)*
Cause the task whose TCB is at *addr* to resume operation at the point at which it was **SUSPEND**ed (or where the task called **STOP**).

**HALT** *( addr — )*
Cause the task whose TCB is at *addr* to cease operation permanently at the next **STOP** or **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** *( — )*
Relinquish the CPU while checking for messages (if the task has a message queue).

**STOP** *( — )*
Check for messages (if the task has a message queue) and suspend operation indefinitely (until restarted by another task, either with **RESUME** or **ACTIVATE**).

**Sleep** *( n — ior)*

Relinquish the CPU for approximately *n* milliseconds. If *n* is zero, the task relinquishes the rest of its time slice. **Sleep** is a Windows call used by **MS** and **PAUSE**, and is appropriate when the task wishes to avoid checking its message queue.

*References*

**MS**, used for interval timing, Section 4.4.2
Windows message handling, Section 8.1.3

# Section 8: Windows Programming in SwiftForth

In this section, we describe how to use SwiftForth to access Windows features, as well as some higher-level implementations of the most common structures. Because SwiftForth was designed from the outset as a Windows system, great care has been given to make the interface to Windows as clean and as easy to use as possible, given the inherent complexity of the Windows environment. We designed this interface to map closely to the usages documented for the many Windows API commands and the examples in the most popular Windows books.

It is beyond the scope of this book to provide a detailed explanation of Windows programming. A list of Windows references is provided at www.forth.com/swiftforth/; we urge you to use one or more of these as your source for Windows functions, parameters, and rules of usage. In addition, Appendix C contains a glossary of the most common terms used in Windows programming.

## 8.1 Basic Window Management

The basic definition of a window is a *dataset* with an associated *callback*. Most windows have visible features, but visibility is not a necessary attribute. There are many classes of windows, ranging from very simple to very complex. SwiftForth's command window is an example of a complex window made up of multiple *child* windows, including the title bar, menu bar, menus, toolbar buttons, horizontal and vertical thumb bars, each pane of the status bar, and the white space in which you can type.

Each window is an instance of a particular *class* of windows defined by the application. Each window has a *handle*. When an event occurs, the OS sends a message to the task that owns the window, which dispatches the message with the window's handle; on receipt of the message, the window's **WindowProc** executes. It's an entirely event-driven paradigm; a window cannot have a persistent, continuing behavior in the sense that a task can.

Not all tasks own windows, but most do. If a task has *any* windows, it must have a message loop in which it frequently checks for messages and dispatches them to the appropriate window. The relationship between a task and its windows is much like the relationship between a program and the interrupt routines that handle devices the program controls, in that the window message processing is asynchronous to the sequential activities of the task.

When an event occurs, it is initially handled by Windows. Windows assembles a *message*—a data packet containing information about the event—and passes it to the thread (task, discussed in Section 7.2) responsible for it. Determination of responsibility depends on the nature of the event: keystrokes go to the currently active window, mouse clicks depend on the screen location, etc. Some events are acted on immediately; others are posted in the task's message queue.

A message contains a *handle* parameter called **HWND** that identifies which window it

should go to, a message number called **MSG**, and other parameters dependent on the nature of the message.

When a task gets a message, typically in response to the Windows command **GetMessage** (which polls its message queue), it dispatches it using the Windows command **DispatchMessage**, which sends it to the window to which it belongs (indicated by **HWND**). Windows puts the message parameters in a stack frame and calls the **WindowProc** callback associated with the window, which processes the message.

These are the basic steps that must be followed in order to construct and manage a window:

1. Define a message handler (typically a single definition or a switch).
2. Define the window's callback, whose function is to get the message parameter and process it, usually by calling the message handler.
3. Define and register the window class.
4. Make one or more instances of the class.
5. Destroy the window when it's no longer needed (or the application shuts down).

These actions, along with related issues, will be discussed in detail in the next sections.

### 8.1.1 Parameter Handling

To communicate with Windows, parameters must be passed back and forth. When you issue a Windows call, often there are quite a few parameters, especially when describing a window, dialog box, or control you wish Windows to draw and maintain. Conversely, when you get a callback, a number of parameters are provided. A standard window callback has four parameters:

- **HWND**, the handle of the window to which the message is addressed;
- **MSG**, the message number;
- **WPARAM** and **LPARAM**, parameters whose meaning depends upon the message number.

Other, more specialized, callbacks have different parameters.

All parameters are single, 32-bit stack items. Many are integers; some are absolute addresses. In some cases, incoming parameters need to be masked to get the high or (more commonly) low 16-bit "word." SwiftForth provides words that facilitate this.

In SwiftForth, all outbound parameters (i.e., *to* Windows) are passed on the stack. SwiftForth will take care of putting them in the right place for the call, providing you supply them in the same order (i.e., the first parameter is lowest, the last on top) as described in Windows documentation (e.g., the Win32API Help system).

Incoming parameters to callbacks are *not* passed on Forth's data stack. Instead, SwiftForth provides a set of words that fetch the values of the top eight parameters. They behave like constants, and are read-only. The words that fetch callback parameters are given in the glossary at the end of this section. Callback parameters

remain available for the duration of the callback.

Very few Windows messages require more than eight parameters. Should you encounter one that does, or should you wish to rename one, you may define additional parameter names using the form:

```
<n> NTH_PARAM <name>
```

…where *n* is the ordinal position of the parameter in the stack frame (0 is the topmost item). We recommend using the names used in Windows documentation.

*Glossary*

**HWND** ( — *n* )
Return the handle of the window receiving the current message.

**MSG** ( — *n* )
Return the message number of the current message.

**WPARAM** ( — *n* )
Return the third parameter for the current message. Its content is message-dependent.

**LPARAM** ( — *n* )
Return the fourth parameter for the current message. Its content is message-dependent.

**_PARAM_0**, **_PARAM_1**, **_PARAM_2**, **_PARAM_3** ( — *n* )
Return the first through the fourth parameters for the current message, respectively. The content of each is message-dependent. These words are synonyms for **HWND**, **MSG**, **WPARAM**, and **LPARAM**, respectively, and are available for use when those names are inappropriate descriptors for the parameters.

**_PARAM_4**, **_PARAM_5**, **_PARAM_6**, **_PARAM_7** ( — *n* )
Return the fifth through the eighth parameters for the current message, respectively. The content of each is message-dependent.

**NTH_PARAM** <name> ( *n* — )
Defines a Windows callback parameter, appearing in the *n*th position from the top of the stack frame (0 = top).

**LOWORD** ( $n_1$ — $n_2$ )
Return the low-order two bytes of $n_1$ as $n_2$.

**HIWORD** ( $n_1$ — $n_2$ )
Return the high-order two bytes of $n_1$ as $n_2$.

**HILO** ( $n_1$ — $n_2$ $n_3$ )
Return the upper and lower halves of $n_1$ as $n_2$ (high-order part) and $n_3$ (low-order part).

### 8.1.2 System Callbacks

A *callback* is an entry point in your program. It is provided for Windows to call in certain circumstances. For example, every individual window is required to have a callback referred to in Windows documentation as **WindowProc** to handle messages dispatched to that window; the standard **WindowProc** mechanism in SwiftForth is a switch named **SFMESSAGES**. Some Windows calls require callbacks for local processing; for example, the code given below for listing system fonts requires a callback to display each individual font line. Many *events* require callbacks for processing.

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 reentrant 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 in the callback. The most common value for *n* is four, but some take more or (rarely) fewer. A callback must always return a single integer result, whose interpretation is described in Windows documentation depending on the event to which it is responding. 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 words described in Section 8.1.1 in any routine responding to the callback.

Since 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.

No user variables are initialized on entry to the callback except **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 by all windows of the same class as well as any other code that references them. If you need persistent data space that is unique for each instance of a window class, you may **ALLOCATE** memory when the window is created, and store its handle in a cell in the window data structure. You must remember to **FREE** this space when the window is destroyed!

For example, to enumerate all fonts in the system:

```
: SHOWFONT ( -- x )             \ Display one line of font data
   HWND MSG WPARAM LPARAM FontFunc
   1 ;

' SHOWFONT 4 CB: &SHOWFONTS

FUNCTION: EnumFonts ( hdc lpFaceName lpFontFunc lParam -- n )

: .FONTS ( -- )
    CR 60 spaces ."  ht wide esc ornt wt I U S set p cp q  fp"
    HWND GetDC                  \ Get device context
    0                           \ No particular font name
    &SHOWFONTS                  \ Address of enumerate callback routine
    0                           \ No application supplied data
    EnumFonts                   \ WINPROC call
    DROP ;                      \ Discard result
```

This defines a four-parameter Windows callback, **&SHOWFONTS**, which calls the Forth routine **SHOWFONT**. When **SHOWFONT** executes, it has free access to the four parameters passed to it via the built-in names **HWND**, **MSG**, **WPARAM**, and **LPARAM**, and it passes them to **FontFunc** which uses them to generate one line of the display. The function **.FONTS** passes the address of **&SHOWFONTS** to the Windows API function **EnumFonts** as the address for the callback.

To see this in action, **INCLUDE %SwiftForth\lib\samples\win32\showfonts.f** and type **.FONTS**.

When the callback function is relatively simple, it's convenient to define it with **:NONAME**, which returns its execution token, or *xt*. For example:

```
:NONAME ( -- res )
   MSG LOWORD  MY-SWITCH ;   4 CB: MY-CALLBACK
```

In this example, the actual callback code only derives the message number, which is then passed to a switch that executes the appropriate response code for each message it supports and returns result code *res*. This is the most common form of callback in SwiftForth.

*Glossary*

**CB:** *( xt n — )*

Defines a Windows callback, which will execute the code associated with *xt* and take *n* parameters.

*References* Switch data structures, Section 4.5.4
**FUNCTION:**, Section 8.2.1
Dynamic memory allocation (**ALLOCATE** and **FREE**), *Section 5.4*.
Memory allocation, *Forth Programmer's Handbook*.

### 8.1.3 System Messages

System messages are usually handled via a compiled switch mechanism (described in Section 4.5.4), which can be extended to include any new messages that need to

be handled. In all cases, a switch used as a message handler expects on the stack a message number and must return a result, whose meaning depends on the message. Consult your Windows documentation for detailed information on any message you wish to support.

For example, here is code to extend SwiftForth's primary Windows message handler **SFMESSAGES** to include keystroke events:

```
[+SWITCH SFMESSAGES ( n -- res )
    WM_SYSKEYDOWN                RUNS KDOWN1
    WM_KEYDOWN                   RUNS KDOWN0
    WM_CHAR                      RUNS CDOWN0
    WM_SYSCHAR                   RUNS CDOWN1
SWITCH]
```

The message names are all of the form **WM_<xxx>** and refer to Windows constants that return a value. These constants are not defined individually in SwiftForth; just use the documented Windows name, and SwiftForth will find them in the list of Windows constants supported by **WINCON.DLL**.

*References*
Switch data structures, Section 4.5.4
Resolution of Windows constant names, Section 5.5.3

### 8.1.4 Class Registration

Having defined a message handler and callback, you have the minimum infrastructure required to register a Windows class. Now you can construct an actual window (an *instance* of a registered class) as many times as you need to.

To register a class, you must provide the message handler and callback, plus a number of parameters that describe the class. The general way of doing this in SwiftForth is by using the word **DefineClass**, which takes ten parameters and returns a handle. This word calls the Windows function **RegisterClass**. The parameters are shown in Table 21; to see the details, **LOCATE DefineClass**.

Because it is fairly cumbersome to specify all these parameters, many of which are not used by most windows, SwiftForth provides a simpler registration word called **DefaultClass**. This takes only two parameters, the address of the zero-terminated string giving the name of the class and the address of the class callback routine. It registers a class with the default values shown in Table 21.

**Table 21: Class registration parameters, with default values**

| Parameter (10 = TOS) | Name | Description | Default value |
|---|---|---|---|
| 1 | style | The class style(s). Styles can be combined by using the bit-wise **OR** operator. | **CS_HREDRAW CS_VREDRAW CS_OWNDC** |
| 2 | callback | *xt* of the window callback procedure. | *you provide* |

**Table 21: Class registration parameters, with default values *(continued)***

| Parameter (10 = TOS) | Name | Description | Default value |
|---|---|---|---|
| 3 | xclass | The number of extra bytes to allocate following the window-class structure. The OS initializes the bytes to zero. | 0 |
| 4 | xwin | The number of extra bytes to allocate following the window instance. The OS initializes the bytes to zero. | 0 |
| 5 | inst | The instance the window procedure of this class is within. | **HINST** (of SwiftForth) |
| 6 | icon | The SwiftForth default icon. | **HINST 101 LoadIcon** |
| 7 | cursor | The class cursor. This must be a handle of a cursor resource. If this member is zero, an application must explicitly set the cursor shape whenever the mouse moves into the application's window. | **IDC_ARROW** (default cursor) |
| 8 | brush | The class background brush. This member can be a handle to the actual brush to be used for painting the background, or it can be a color value (1–20). | **WHITE_BRUSH** (default background) |
| 9 | menu | Address of a zero-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file. | 0 (none) |
| 10 | name | Address of a zero-terminated string giving the window class name. | *you provide* |

*Glossary*

**DefineClass** ( $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$ $x_8$ $x_9$ $x_{10}$ — $n_h$ )

Accepts parameters on the stack and registers the class. Refer to Table 21 for the parameter list and descriptions. Returns the class handle.

**DefaultClass** ( $addr_1$ $addr_2$ — $n_h$ )

Registers a class whose name is the zero-terminated string at $addr_1$ and whose callback routine (defined by **CB:**) is $addr_2$, with default parameters as shown in Table 21. Returns the class handle.

### 8.1.5 Building a Windows Program

This section presents a step-by-step tutorial describing the steps required to develop a simple Windows application, which is included as an example. The steps are in an order that has proven to work, but this is by no means the only order in

which they could be done. The program described here is provided as the file **SwiftForth\lib\samples\win32\clicks.f**.

A Windows program is not like a traditional program that has a single entry point and that executes more-or-less sequentially till finished. It has an entry point and a main routine, but almost all the work is accomplished during callbacks in response to Windows events.

The example program will count mouse clicks (aka button presses) inside a window and will display the tally for both right and left buttons. To do this, we need the following:

- a window
- a routine to recognize mouse clicks and count them
- a routine to display the results

Here are the steps that provide these things:

- Define the message handler and callback routine, which together constitute a *class* of Windows objects.
- Register the class.
- Create and display the window (an *instance* of the class).
- Run the message loop until the window is closed.
- Encapsulate the functionality provided in steps 1–4 into a program.
- Define the window behavior in terms of message responses.

Note that, in the code that follows, all references of the form **WM_<xxx>** are constants associated with Windows messages. References to words such as **WS_OVERLAPPEDWINDOW** are also constants. These constants are automatically resolved as literals by the compiler from the **Wincon.dll** file, as described in Section 5.5.3.

#### 8.1.5.1 Define the Window Class

Our first steps involve creating a Windows class, which consists of a message handler and callback routine. We will also define a string containing its name, as this saves the trouble of providing it as a literal string (with consistent spelling!) every time we need to use it.

This step doesn't have to be first, but it is convenient to have the framework already defined, and then to extend it as the program grows. A message switch is required to define the callback, and the callback is required before the class can be registered, and the class must be registered before the window can be created.

In the code below, **hAPP** holds the application handle. This is used to detect that the application is active, and provides a means for accessing the application from the SwiftForth interactive environment.

```
0 VALUE hAPP
CREATE CLASSNAME ,Z" Counting"
```

The message handler is initially an empty switch (see the discussion of switch structures in Section 4.5.4) whose default behavior is to call the default Windows message handler. We will add specific message handlers to the switch later.

```
[SWITCH MESSAGE-HANDLER DEFWINPROC ( -- res )
    ( no behavior yet)
SWITCH]
```

Having defined the switch, we can now define the callback. We use the defining word **:NONAME** (see "Colon Definitions" in *Forth Programmer's Handbook*, ), since the function is useless in any other context.

The parameters to the function are held in the pseudo-values **HWND MSG WPARAM** and **LPARAM**—not on the stack! These parameters (described in Section 8.1.2) are available to any routine running during the execution of the callback. The function must return a numeric result, which will later be used with the switch to find the right behavior.

```
:NONAME ( -- n )
    MSG LOWORD MESSAGE-HANDLER ;
4 CB: APPLICATION-CALLBACK
```

It is very nice to be able to repeatedly reload an application being debugged. During the **PRUNE** operation (described in Section 4.1.1), SwiftForth needs to be able to close the window if it is open and to unregister the class so things don't crash. Once a class is registered, Windows may exercise the callback at any time. If SwiftForth is going to prune the callback, it must first *unregister* the class so that Windows will not attempt to call it.

```
:PRUNE   ?PRUNE -EXIT
    hAPP IF  hAPP WM_CLOSE 0 0 SendMessage DROP  THEN
    CLASSNAME HINST UnregisterClass DROP ;
```

However, depending on the complexity of your program, it may not be practical to use **PRUNE** in SwiftForth apps.

#### 8.1.5.2 Register the Class

We register the class here via **DefaultClass**, described in Section 8.1.4. Note that **DefaultClass** is *not* a Windows function, but an abstraction of the Windows **RegisterClass** function and the Windows **WNDCLASS** structure. **DefaultClass** sets the default parameters shown in Table 21, plus the *xt* of your callback and your application name.

```
: REGISTER-CLASS ( -- )
    CLASSNAME APPLICATION-CALLBACK DefaultClass
    DROP ;                          \ Discard return parameter
```

#### 8.1.5.3 Create and Show the Window

Only after the class is registered are we allowed to create a window of that class.

Here we create a window, specifying its styles, size, and title. The return value is a handle to the window.

If we are debugging this application from SwiftForth, and we *intend* always to run it from SwiftForth, this is all we need to do. The reason is that SwiftForth already has a message loop running, and all Windows created by SwiftForth will be processed automatically.

```
: CREATE-WINDOW ( -- handle )
   0                           \ extended style
   CLASSNAME                   \ window class name
   Z" Counting clicks"         \ window caption
   WS_OVERLAPPEDWINDOW         \ window style
   10 10 600 400               \ position and size
   0                           \ parent window handle
   0                           \ window menu handle
   HINST                       \ program instance handle
   0                           \ creation parameter
   CreateWindowEx ;
```

#### 8.1.5.4 Start the Message Loop

We need to chain together the parts defined so far. Note that the message handler is already linked to the class via the class's callback routine. We return zero if we did not create the window, and non-zero (the window's handle) if we did.

The **START** routine starts the message loop running. It is usable in the SwiftForth environment and will allow interactive testing of the running window; i.e., you may change the value of either variable and execute **SHOW** and the window will update.

```
: START ( -- flag )
   REGISTER-CLASS CREATE-WINDOW DUP IF
         DUP SW_NORMAL ShowWindow DROP
         DUP UpdateWindow DROP
         DUP TO hAPP
   THEN ;
```

**START** can also be used to start a message loop in a standalone program, as described next.

#### 8.1.5.5 Encapsulate the Functionality of the Program

If we want to run this as a standalone application, we must provide a full message loop and application termination.

The word **GO** below does not return to SwiftForth—on entering it, the system will run the application until it is closed, and then will close SwiftForth as well. This is the behavior of a standalone application, and is not useful in the development environment.

```
: GO ( -- )
   START IF DISPATCHER
    ELSE 0 Z" Can't create window" Z" Error" MB_OK
    MessageBox
```

```
    THEN  'ONSYSEXIT CALLS  0 ExitProcess ;
```

To establish this program as an application, we place the *xt* of **GO** into **'MAIN** and save the image via **PROGRAM-SEALED**. This is appropriate after everything is tested and you are ready to launch your program as a turnkey (see Section 4.1.2).

```
' GO 'MAIN !
PROGRAM-SEALED Test.exe
```

Even though we don't actually do this until the work described in the next section is complete, we are bringing this in now so you see how **'MAIN** is set, as its content is checked in the next section.

*Glossary*

**PROGRAM-SEALED** <filename>[.<ext>] [<icon>] *( — )*

Same syntax as **PROGRAM** (see Section 4.1.2), but generates a "sealed" turnkey image (a program with no user-level access to the Forth interpreter or dictionary), which does not request the full dictionary allocation. The virtual memory specified in the header is the same as the size of the SwiftForth program image.

**'MAIN** *( — addr )*

Return the address of an execution vector containing the main program word to be launched at startup. This will occur after the DLLs have been loaded, but *before* the **STARTER** word set in a turnkey is executed. The default for **sf.exe** is **START-CONSOLE** (which launches the command window).

#### 8.1.5.6 Define Window Behavior

So far, we have defined a very default window with no custom responses to anything. In order for this to be an application, we have to implement message behaviors appropriate to our goals.

Almost none of the words described in this section are directly executable from the Forth interpreter; they must be called via the Windows callback routine. The message responses may be either named words referenced by **RUNS** or un-named and referenced by **RUN:** (both described in Section 4.5.4).

This application will simply count left and right mouse clicks in its client area. It is designed to run directly from SwiftForth, to aid in debugging it, but will have hooks to let it run as a standalone application when finished.

**REFRESH** simply tells Windows to update the contents of the current window, which is identified by **HWND**.

```
: REFRESH ( -- )
   HWND 0 0 InvalidateRect DROP ;
```

Next, we define variables to hold the button-click counts.

```
VARIABLE RIGHT-CLICKS
VARIABLE LEFT-CLICKS
```

Mouse-button clicks are sent as messages to the active window—so now we add

message *actions* for right and left button presses.

```
[+SWITCH MESSAGE-HANDLER ( -- res )
    WM_RBUTTONDOWN RUN: 1 RIGHT-CLICKS +! REFRESH 0 ;
    WM_LBUTTONDOWN RUN: 1 LEFT-CLICKS +! REFRESH 0 ;
SWITCH]
```

Having provided the ability to count clicks, we consider the problem of displaying the tallies. Note that normal display words such as **EMIT** may not be used during a callback, but output formatting words such as **(.)** may. **HERE** is a valid (and reentrant) address during a callback, so we build our output string at **HERE**.

**(SHOW-CLICKS)** formats the window's text at **HERE**, returning the string:

```
: (SHOW-CLICKS) ( -- addr n )
    LEFT-CLICKS @ (.) HERE PLACE
           S" left right " HERE APPEND
    RIGHT-CLICKS @ (.) HERE APPEND
           HERE COUNT ;
```

**SHOW-CLICKS** is the core of the **REPAINT** function. It formats the text at **HERE**, determines the size of display on which to show the text, and draws the text:

```
: SHOW-CLICKS ( hdc -- )
    (SHOW-CLICKS)
    HWND PAD GetClientRect DROP PAD
    DT_SINGLELINE DT_CENTER OR DT_VCENTER OR
    DrawText DROP ;
```

Unfortunately, nothing is automatic in Windows! We can display messages easily, but if another window obscures our window, the information previously displayed is lost. We must be able to repaint the window on demand. This means handling the **WM_PAINT** message.

```
: REPAINT ( -- res )
    HWND PAD BeginPaint ( hdc)
    HWND HERE GetClientRect DROP
    ( hdc) SHOW-CLICKS
    HWND PAD EndPaint DROP 0 ;

[+SWITCH MESSAGE-HANDLER ( -- res )
    WM_PAINT RUNS REPAINT
SWITCH]
```

The last messages we must handle are related to closing the window. We need to destroy the window and release resources when we receive **WM_CLOSE**. The act of destroying the window sends the **WM_DESTROY** message, which we must also respond to by posting the quit message, which notifies the dispatcher that the application is finished.

```
[+SWITCH MESSAGE-HANDLER ( -- res )
    WM_CLOSE RUN: HWND DestroyWindow DROP 0 TO hAPP 0 ;
    WM_DESTROY RUN: 0 PostQuitMessage DROP 0 ;
SWITCH]
```

The dispatcher will see **PostQuitMessage** terminate the application *and,* as a side-

effect, will terminate SwiftForth as well. This is not acceptable for a development session, and may not be acceptable even for a main application.

To fix this, we will redefine the **WM_DESTROY** handler to be a little smarter. The new definition will override the previous one. **-APP** returns true if we are not a stand-alone application. We determine this by checking the vector in **'MAIN** at run time against the vector at compile time.

```
: -APP ( -- flag )
    'MAIN @ [ 'MAIN @ ] LITERAL = ;

[+SWITCH MESSAGE-HANDLER ( -- res )
    WM_DESTROY RUN: 0 -APP IF EXIT THEN
    0 PostQuitMessage DROP ;
SWITCH]
```

## 8.2 SwiftForth and DLLs

The dynamic-link library (also written without the hyphen), or DLL, is Microsoft's implementation of the shared library in the Windows operating system. These libraries usually have the file extension **.dll**.

To write Windows applications, you must have access to the functions imported from DLLs. SwiftForth imports many Windows DLL functions and also supplies a simple mechanism to import any library functions needed by your application.

SwiftForth also allows you to create DLLs with exported functions.

### 8.2.1 Importing DLL functions

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. The default extension for *filename* is **.dll**. Windows first searches for "known" DLLs, such as **Kernel32** and **User32**. It then searches for the DLLs in the following sequence:

1. The directory where the executable module for the current process is located.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. The directories listed in the **PATH** environment variable.

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 documentation page or in a "header" file). The right side should be a single item (the return value) or it may be empty (nothing returned).

The names of imported library functions are case sensitive, and must exactly match the form given in their documentation.

**FUNCTION:** searches all available DLLs (those declared by **LIBRARY**) for the named procedure and, if found, creates a Forth word (with the same name) that calls it. For example:

```
FUNCTION: MessageBox ( hWnd lpText lpCaption uType -- n )
```

This defines a call to the **MessageBox** function, which can then be used simply by referencing it as a Forth word:

```
HWND Z" Hello, world!" Z" Greetings" MB_OK MessageBox DROP
```

The Windows API definition of **MessageBox** looks like this:

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

Note that the order of the parameters in the SwiftForth **FUNCTION:** definition is the same left-to-right order specified by the Windows API documentation. Each stack item is one cell in size and strings are normally null terminated.

**FUNCTION:** may also be used with **AS** to provide a different internal Forth name for an imported function. This is useful when you are importing a function with the same name as a word already defined in the Forth dictionary.

The syntax is:

```
AS <Forth-name>  FUNCTION: <external-name> ( params -- return)
```

Important: The **AS** clause and the **FUNCTION:** definition must be on a single line.

All DLL functions known to SwiftForth may be displayed via the **.IMPORTS** command, which displays the resolved address for the routine, the library in which it is found, and its name. All libraries that have been opened with **LIBRARY** may be displayed with **.LIBS**.

A library or imported function may be marked as optional. For example:

```
[OPTIONAL] LIBRARY TTY.DLL
```

```
[OPTIONAL] Function: MyProc ( 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.

*Glossary*

**LIBRARY** <filename.dll> *( — )*
Add *filename* to the list of DLLs currently available to SwiftForth.

**[OPTIONAL]** ( — )
Specifies that the next library or function declaration is optional.

**.LIBS** *( — )*
Display a list of all available libraries (previously defined with **LIBRARY**).

**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. Imported functions return at most one value on the stack.

**.IMPORTS** *( — )*
Display a list of all currently available imported library functions (previously defined with **FUNCTION:**), showing for each its resolved address, the DLL in which it is defined, and its name.

**AS** <name> *( — )*
Provide a different internal Forth name for an imported function, or a different external name for a function exported from a SwiftForth DLL (Section 8.2.2).

### 8.2.2 Creating a DLL

SwiftForth can create a DLL with exported functions. A DLL made from SwiftForth contains the entire system.[1]

The steps to make a DLL are:

1. Define the Forth functions to be exported in a DLL. DLL functions may take a number of arguments, but always return exactly one argument. The implication for Forth functions in DLLs is that the stack within the DLL is not persistent across consecutive calls.
2. Export the Forth functions, associating them with case-sensitive names that are unique for all included DLLs, using **AS** (if necessary) and **EXPORT:** as follows for each Forth function to be exported. Pre-existing Forth functions may be exported without redefinition. For example, given:

```
: SQRT ( n1 – n2 )  ... ;  \ n2 is the square root of n1
```

1.In order to comply with FORTH, Inc. licensing policies (see Section 1.5), you may not export any access to SwiftForth development tools, including the compiler and assembler, without making appropriate arrangements with FORTH, Inc.

…you could then use the form:

```
AS <external-name>  <n> EXPORT: <Forth-name>
```

…where *n* is the number of parameters needed by *Forth-name*. Thus:

```
AS SquareRoot   1 EXPORT: SQRT
```

Important: The **AS** clause and the **EXPORT:** declaration must be on a single line.

3. Repeat to export other functions.
4. Create a DLL using the **PROGRAM** function, for example:

```
PROGRAM CALCULATOR.DLL
```

Consider this example. A DLL is to be created, having four calculator functions based on Forth math operators. The functions are exported as follows:

```
AS Plus         2 EXPORT: +
AS Minus        2 EXPORT: -
AS Times        2 EXPORT: *
AS Divide       2 EXPORT: /

PROGRAM CALC.DLL
```

Any program capable of calling DLL functions can import the exported functions. To reference these functions in SwiftForth, use:

```
LIBRARY CALC

FUNCTION: Plus ( n1 n2 -- n3 )
FUNCTION: Minus ( n1 n2 -- n3 )
FUNCTION: Times ( n1 n2 -- n3 )
FUNCTION: Divide ( n1 n2 -- n3 )
```

Strict prohibitions apply against absolute address references in DLLs (see Section 5.1.4). Therefore, when developing any code that might become part of a DLL, it is important to use SwiftForth's system option called **PROTECTION** while you are developing and testing *any code that might ever be used in a DLL*. **PROTECTION** augments the compiler such that it will attempt to identify compiled references to absolute addresses and give you a warning.

You may add **PROTECTION** by including the phrase **REQUIRES PROTECTION** at the beginning of your main **INCLUDE** file, or add it to the system using the menu item Options > Optional Packages > System Options, and then make a turnkey of the resulting, augmented SwiftForth.

If **PROTECTION** is loaded, the Options > Warnings dialog box will be enhanced to facilitate management of the address-checking options.

*Glossary*

**EXPORT:** <name> *( n — )*

Export the function *name* so it will be available to other programs that load a DLL you create. *n* indicates the number of parameters needed by *name*.

*References* Memory model and address management, Section 5.1.4
System warning configuration, Section 2.3.5
**PROGRAM** (turnkey images), Section 4.1.2

## 8.3 Dynamic Data Exchange (DDE)

DDE is a mechanism by which separate Windows applications can request data from each other. The requesting program is considered a *client*, and the program that manages the data acts as a *server*.

SwiftForth provides a simple API for client services based on the DDE Management Library (DDEML). This library provides about 30 function calls that are defined in SwiftForth in the file **ddeclient.f**.

There are three basic steps in a DDE conversation:

1. The client establishes a relationship with the server. This requires that the server be running; a **1005 THROW** will occur if it is not, or if a link cannot be established for any other reason.
2. The client establishes a *topic* on which it wishes to converse. All DDE servers support at least one topic, and most have several. For example, Excel can discuss its system functions (e.g., what formats or protocols it supports) as well as individual worksheets.
3. The client sends specific *items* (questions) relating to the topic, and the server responds appropriately.

Using DDEML, inter-application communication consists of the client passing string handles that specify the server, topic, and item; responses are received as strings, as well. The SwiftForth words **SERVER**, **TOPIC**, and **ITEM** each take strings from the input stream (using **ZSTRING**) and set them up where they may be easily passed to the DDE functions. The word **ZTELL** is provided as a diagnostic, to display a received string.

The words **DDE-INIT**, **DDE-REQ**, **DDE-SEND**, and **DDE-END** conduct specific steps in the conversation.

As examples of how these may be used, consider this definition of an inquiry:

```
: ASK ( -- )                    \ Request an item from a server
   ITEM                         \ Capture the request from input stream
   DDE-INIT                     \ Initialize the conversation
   DDE-REQ                      \ Send the request, get response
   DDE-END                      \ Terminate the conversation
   ZTELL ;                      \ Display the data
```

Here is an example using this:

```
SERVER EXCEL  ok
TOPIC SYSTEM
ASK SYSITEMS
SysItems Topics Status Formats Selection Protocols EditEnvItems ok
```

The last line represents the list of items returned from Excel. Here's an example

involving sending data:

```
: TELL ( addr n -- )             \ Send the string addr n
    ITEM                         \ Capture the data from input stream
    DDE-INIT                     \ Initialize the conversation
    DDE-SEND                     \ Send the data
    DDE-END ;                   \ Terminate the conversation

SERVER EXCEL
TOPIC 25JAN98.XLS                \ Topic is a worksheet file
ASK R1C1                         \ Fetch from row 1, col 1
Name:
S\" Rick VanNorman\n" TELL R1C2
```

In this example, row 1 column 1 contains the heading `Name:`, and the string `Rick VanNorman` is sent to row 1 column 2.

*Glossary*

**SERVER** <name> *( — )*

Set the string *name* as the server identifier.

**TOPIC** <string> *( — )*

Set *string* as the topic.

**ITEM** <string> *( — )*

Set *string* as a data item to be used as the request or transaction.

**DDE-INIT** *( — )*

Check for necessary data (server, topic, item) and open the conversation. If the server doesn't respond or doesn't recognize the topic, a **1005 THROW** will occur.

**DDE-REQ** *( — addr )*

Send a request item to the server, and receive a response as an ASCIIZ string at *addr* (which is at **PAD**).

**DDE-SEND** *( addr n — )*

Send the string *addr n* to the server.

**DDE-END** *( — )*

Terminate the conversation.

## 8.4 Managing Configuration Parameters

Windows applications can maintain configuration parameters in the Windows registry. SwiftForth uses this facility to maintain a number of different parameters, including items set from the Options menu and others.

To see dumps of the currently defined configuration parameters, use the command **.CONFIGURATION**. When SwiftForth is launched, it automatically initializes its local

copies of these parameters from the registry. The local copies are recorded in the registry when SwiftForth exits.

SwiftForth provides the defining word **CONFIG:** for you to add application configuration parameters to this list. This constructs a colon definition whose name is added to the list of configuration parameters. The behavior of a word defined by **CONFIG:** must be to return an address and length in bytes of a parameter or parameter list.

Consider these examples:

**Example 1.**

```
CREATE VALS   1 C,  2 C,  3 C,  4 C,
CONFIG: VALS-KEY ( -- addr n )
        VALS 4 ;
```

In this example, execution of the name **VALS** returns the address, and 4 provides the length.

**Example 2.**

```
100 VALUE SIZE
CONFIG: SIZE-KEY ( -- addr n )
        ['] SIZE >BODY  CELL ;
```

Here we couldn't execute **SIZE** in the definition, because that would have returned the value. Nor can we compile the address of **SIZE** in the definition using a phrase such as **[ ' SIZE >BODY ] LITERAL**, because that would have compiled the *current* absolute address of **SIZE** which would not be appropriate next time SwiftForth is booted. The correct approach is to derive the address at run time, as shown in the example.

*Glossary*

**CONFIG:** <name> *( — )*

Define a word that will be placed in the list of configuration parameters to be maintained in the Windows registry. When executed, *name* must return the address and length of its data area, which will be recorded and initialized by SwiftForth at startup.

*References* Options menu, Section 2.3.5
Addresses of data objects, Sections 5.1.4 and 4.5.2

## 8.5 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** *( addr len — n )*
Returns the argument count in the command line buffer *addr len*.

**ARGV** *( addr1 len1 i — addr2 len2 )*
Returns argument *i* from command line *add1 len1* as string *addr2 len2*. If *i* is an invalid argument number (i.e., not less than **ARGC**), a zero-length string is returned.

**CMDLINE** *( — addr len )*
Returns its address and count of the command line.

## 8.6 Environment Queries

The operating system environment variables can be queried with **FIND-ENV**. For example, if the environment contains the entry "`PATH=C:\Windows`" then the query

```
S" SHELL" FIND-ENV
```

returns the address and length of the string "`C:\Windows`". Do not include the "=" in the search string; **FIND-ENV** appends it to its search buffer.

The command **.ENV** 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** *( — )*
Displays the environment for diagnostic purposes.

**ATOI** *( addr len — n )*
Converts a counted character 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.

## 8.7 A Self-Contained Windows Application

It is possible to define a fully self-contained Windows application that does not depend in any way upon the SwiftForth command window. A simple example of such a *standalone* program is **SwiftForth\lib\samples\win32\hellowin.f**. This program is based on a sample program in Petzold, *Programming Windows* (see references at www.forth.com/swiftforth/) that displays a window and plays a **WAV** file that says, "Hello, Windows!" If you **INCLUDE** this file in SwiftForth and then save it as an executable using the phrase **PROGRAM HELLO**, the resulting **hello.exe** is a standalone program that may be run without any other component of SwiftForth, just its data file **hellowin.wav**.

The essential features of this standalone program are:

1. A class name, in this case **AppName**.
2. The switch **HELLO-MESSAGES**, which handles the callback messages for this class. It is via these messages that Windows executes a program. The minimum required is that we handle the **WM-DESTROY** message, which responds to the close box in the window's upper-right corner, among other things.
3. A callback for the class, **WNDPROC**. It extracts the content of a message and sends it to **HELLO-MESSAGES**.
4. A WNDCLASS structure called **MYCLASS**, which defines the overall characteristics of the windows we want to define.
5. An instance of the class constructed by **MYWINDOW**, whose name is the content of **AppName**.
6. In this example, we require an external DLL called **WINMM.DLL** to play the sound. It must be added, and a procedure from it (**PlaySound**) defined.
7. We also need a couple of definitions to play the sound and write the message in the middle of the window; these are **TADA** and **PAINT**, respectively.
8. We add two more messages to the switch **HELLO-MESSAGES**, to run these words.
9. We construct a main program called **WINMAIN** to launch the whole thing, and attach it by putting its *xt* in **'MAIN**.

Section 8.1.5.5 (page 136) discusses how to assign a start-up word to a turnkey application.

# Section 9: Defining and Managing Windows Features

There are many classes of Windows objects, including windows, dialog boxes, buttons, toolbars, status bars, and other kinds of controls. Each is, technically, a "window," in that it is a dataset with a callback. Most objects also respond to a set of messages characteristic of the object's class. This section discusses the most commonly occurring classes of objects; the principles by which SwiftForth handles them may be straightforwardly applied to other classes of objects described in the Windows API documentation.

## 9.1 Menus

Creating a menu in SwiftForth involves two steps: building a data structure that describes the elements of the menu, and instantiating the menu by means of a Windows call. The basic structure of a menu is:

```
MENU <name>
   <items>
END-MENU
```

**MENU** is a defining word which starts building a data structure which Windows can interpret as a menu via the **LoadMenuIndirect** function. All menus are defined by it. Although **MENU** creates a Forth word *name* for the data structure, there is no Windows-associated text for it.

Two kinds of elements can be referenced inside a **MENU** … **END-MENU** structure: *sub-menus* (which display a list of choices) and individual *menu items*.

**POPUP** … **END-POPUP** defines a sub-menu. **POPUP** is followed by the label that will appear for this sub-menu. The sub-menu is an element within the menu, it is not itself a named word. It may contain menu items and additional sub-menus. Individual menu items are created by these words:

- **MENU** adds a normal, enabled menu item.
- **GRAY** adds a disabled menu item.
- **CHECK** adds an enabled menu item which is initially checked.
- **SEPARATOR** inserts a separator bar into the menu structure.

The syntax of the first three of these is:

```
<id> xxxx "text"
```

where *id* is simply a user-defined number and *text* is normal ASCII text surrounded by quote marks. Note that the leading quote mark requires a space to its *left* but none to its right; the trailing quote mark requires no space at all.

A simple, useful example of a menu in the **hello.f** sample is:

```
MENU HELLO-MENU
```

```
   POPUP "&File"
      MI_EXIT MENU "E&xit"
   END-POPUP

   POPUP "&Help"
      MI_ABOUT MENU "&About"
   END-POPUP
END-MENU

: CREATE-HELLO-MENU ( -- hmenu )
   HELLO-MENU LoadMenuIndirect ;
```

which assumes that the **MI_** items are unique constants. The character preceded by **&** will appear underlined and, if that letter plus the Alt key are pressed, Windows will generate a message that this item has been selected.

This menu's data structure is named **HELLO-MENU** and is instantiated by the call to **LoadMenuIndirect**. Any menu created by **LoadMenuIndirect** must eventually be gotten rid of by **DestroyMenu**, except for the primary window menu itself, which will be automatically destroyed when the window is closed.

This defined a menu and made it appear in the window, but no behavior was assigned to it. Menu execution is assigned by using the **WM_COMMAND** message's **WPARAM** parameter. This is normally handled by a set of **[SWITCH** structures (see Section 4.5.4); referring again to the **hello.f** sample:

```
[SWITCH HELLO-COMMANDS ZERO
  MI_EXIT        RUN: ( -- res )   HELLO-CLOSE 0 ;
  MI_ABOUT       RUN: ( -- res )   HELLO-ABOUT 0 ;
SWITCH]

[SWITCH HELLO-MESSAGES HELLO-DEFAULT
  WM_COMMAND     RUN: ( -- res )  WPARAM LOWORD HELLO-COMMANDS ;
  WM_PAINT       RUN: ( -- res )  HELLO-PAINT 0 ;
  WM_CREATE      RUN: ( -- res )  HELLO-CREATE 0 ;
  WM_LBUTTONDOWN RUN: ( -- res )  PRESSES@ 1+ PRESSES!
                                  1 +TO PRESSES  0 ;
  WM_TIMER       RUN: ( -- res )  HWND 0 1 InvalidateRect DROP 0 ;
  WM_CLOSE       RUN: ( -- res )  HELLO-CLOSE 0 ;
SWITCH]

:NONAME ( -- res )   MSG LOWORD HELLO-MESSAGES ;
   4 CB: HELLO-WNDPROC
```

Sub-menus can be nested; for example, this defines SwiftForth's Options menu:

```
POPUP "&Options"
      MI_FONT        MENU "&Font"
      MI_EDITOR      MENU "&Editor"
      MI_PREFS       MENU "&Preferences"
      MI_WARNCFG     MENU "&Warnings"
      MI_MONCFG      MENU "&Include monitoring"
      MI_FPOPTIONS   GRAY "FP&Math Options"

         POPUP "&Optional packages"
```

```
      MI_EXTENSIONS   MENU "&System Options"
      MI_SAMPLES      MENU "Generic Samples"
      MI_WINSAMPLES   MENU "Win32 Samples"
      MI_WINTEMPLATES MENU "Win32 Templates"
                      SEPARATOR
      MI_WINOOP       MENU "SFC Packages"
      MI_SFCSAMPLES   MENU "SFC Samples"
   END-POPUP
                   SEPARATOR
MI_SAVEOPTIONS MENU "&Save Options"
END-POPUP
```

*Glossary*

**MENU** <name> *( — )*

Start building the data structure for a menu. Terminated by **END-MENU**. Between **MENU** and **END-MENU** may be sub-menus described with **POPUP** … **END-POPUP** and individual menu items. Use of *name* will return the data structure's address.

**POPUP** <label> *( — )*

Start building a popup sub-menu whose definition will be terminated by **END-POPUP**. Between **POPUP** and **END-POPUP** may be individual menu items. The sub-menu will appear as *label* on the parent menu. Sub-menus may be nested.

**MENU** "text" *( n — )*

Add an enabled menu item whose ID is *n*, which will be labeled *text* on the menu. Must be used within a menu or sub-menu definition.

**GRAY** "text" *( n — )*

Add a disabled (gray) menu item whose ID is *n*, which will be labeled *text*. Must be used within a menu or sub-menu definition.

**CHECK** "text" *( n — )*

Add an enabled menu item whose ID is *n*, which will be labeled *text*. Must be used within a menu or sub-menu definition. The item will be initially marked with a checkmark.

**SEPARATOR** *( — )*

Add a separator bar at the current position in a menu. Must be used within a menu or sub-menu definition.

## 9.2 Dialog Boxes

Dialog boxes are supported via a simple dialog compiler, which parallels the Microsoft resource compiler. An example of a dialog box (shown at right) is given in the file **SwiftForth\lib\samples\win32\simple.f**. The design and management of dialog boxes is discussed in this section.

### 9.2.1 Defining a Dialog Box

Dialog boxes may be of two basic kinds

- *Modal* dialog boxes must be closed before the user can do anything else in this application.
- *Modeless* dialog boxes may remain open until closed.

The dialog box associated with Options > Preferences and the **simple.f** example are modal dialogs; SwiftForth's Watch window is a modeless dialog.

To define a dialog box, use the form:

```
DIALOG <name>
   [<kind> " <title> " <x> <y> <cx> <cy> <options> ]
   [<control> ... ]            (one for each control in your dialog box)
END-DIALOG
```

...where:

- *kind* is either **MODAL** or **MODELESS**, so you would use the form:

  ```
  [MODAL    ...   ]or
  [MODELESS ...   ]
  ```

- *name* is its Forth name
- *title* is the text that will be displayed in the title bar
- *x* and *y* are the position of the upper-left corner, in *dialog box units* (about 1/4 an average character width in *x* and 1/8 an average character height in *y*, for the system font)
- *cx* and *cy* are the horizontal and vertical sizes, in dialog box units
- *options* may include font or style modifiers. A font modifier begins with **(FONT** followed by the size parameter and font name, and ends with a **)**. If the specified font isn't available on a particular computer, a system default will be substituted. Style modifiers are described in Section 9.2.2.

The definition is terminated by **END-DIALOG**. The word **DIALOG** creates a data structure whose address is returned by the use of *name*. It may be used as a parameter to **DialogBoxIndirectParam**, which will instantiate it, as we shall see shortly.

The dialog box in **simple.f** is defined like this:

```
DIALOG (SIMPLE)
   [MODAL  " Press counter"      10   10  160   40
      (FONT 8, MS Sans Serif) ]

\  [control        " default text"   id   xpos ypos xsiz ysiz ]
   [DEFPUSHBUTTON  " OK"             IDOK   105   20   45   15 ]
   [PUSHBUTTON     " Clear"          103     05   20   45   15 ]
   [PUSHBUTTON     " Throw"          104     55   20   45   15 ]
   [RTEXT                            101     05   05   18   10 ]
   [LTEXT          " Total errors"   102     25   05   50   10 ]

END-DIALOG
```

Its control specifications are described in Section 9.2.3.

*Glossary*

**DIALOG** <name> *( — )*
Start building the data structure for a dialog box, which will be terminated by **END-DIALOG**. This must be followed by a specification for the dialog box using **[MODAL** or **[MODELESS**, and any controls featured in the dialog box. Use of *name* will return the data structure's address.

**[MODAL** *( — )*
Begins the specification of parameters for a modal dialog box, which include the title, location, and size parameters, as well as optional font and style specifiers. The parameter list is terminated with **]**.

**[MODELESS** *( — )*
Begins the specification of parameters for a modeless dialog box, which include the title, location, and size parameters, as well as optional font and style specifiers. The parameter list is terminated with **]**.

**(FONT** *( — )*
Begins the font specification for a dialog box, followed by the size and font name. The specification is terminated with **)**.

### 9.2.2 Dialog Box Styles

When you define a dialog box as described in Section 9.2.1, it uses a template based on certain defaults. Every dialog box template specifies a combination of style values that define the appearance and features of the dialog box. The style values can be window styles, such as **WS_POPUP** and **WS_SYSMENU**, and dialog box styles, such as **DS_MODALFRAME**. The number and type of styles for a template depends on the type and purpose of the dialog box.

The default styles set by SwiftForth for modal and modeless dialogs are:

- Modal: **WS_POPUP**, **WS_CAPTION**, **DS_MODALFRAME**, **WS_VISIBLE**
- Modeless: **WS_POPUP**, **WS_SYSMENU**, **WS_CAPTION**, **WS_BORDER**, **WS_VISIBLE**

You may specify styles for your dialog box using **(STYLE** followed by a list of style specifier constants terminated by **)**. You may also add or remove style specifiers using the words **(+STYLE** and **(-STYLE**. Each is followed by one or more constants specifying styles; the list is terminated by a **)**. For example, the sample tabbed dialog box in **tabbed.f** begins with:

```
DIALOG (TAB1)
   [MODELESS 4 21 172 95
   (STYLE WS_CHILD WS_VISIBLE WS_BORDER)
   (FONT 8, MS Sans Serif) ]
```

Style specifiers may also be used with the controls inside a dialog box. For example, this tabbed dialog box has an edit text box control specified this way:

```
[EDITTEXT IDC_EDIT1   20 17  110 12 (+STYLE ES_AUTOHSCROLL) ]
```

…which automatically scrolls text to the right by 10 characters when the user types a character at the end of the line.

*Glossary*

**(STYLE** *( — )*

Begins a list of style parameters that will apply to the dialog box or control currently being built. Parameters are specified by Windows constants that will be ORed together. The parameter list is terminated with **)**.

**(+STYLE** *( — )*

Begins a list of style parameters that will be added to the dialog box or control currently being built. Parameters are specified by Windows constants that will be ORed together. The parameter list is terminated with **)**.

**(-STYLE** *( — )*

Begins a list of style parameters that will be deleted from the dialog box or control currently being built. Parameters are specified by Windows constants that will be ORed together. The parameter list is terminated with **)**.

### 9.2.3 Dialog Box Controls

Most dialog boxes have *controls*, which are objects such as pushbuttons, checkboxes, edit boxes, etc., for user interaction. You may define as many controls as you like (up to Windows' limit of 255) for your dialog box. The definition of a control is similar to that for a dialog box, except it is identified by number rather than by name. The form is:

```
<defining word> " <text> " <id> <x> <y> <xc> <yc> ]
```

…where:

- *defining word* is one of those in Table 22.
- *text* will be displayed in the control (if there is none, this may be omitted).
- *id* is the numeric ID of the control; it may be a value predefined in Windows, or may be programmer-assigned uniquely for this dialog box. (To avoid conflict with IDs assigned by Windows, we recommend values starting with 100.)
- *x* and *y* are the position of the upper-left corner, in dialog box units.
- *cx* and *cy* are the horizontal and vertical sizes, in dialog box units.

The dialog box in **simple.f** has these controls:

**Table 22: Dialog box controls**

| Defining word | Description |
|---|---|
| **[AUTO3STATE** | Checkbox with on/off/gray states. |
| **[AUTOCHECKBOX** | Checkbox with on/off states. |

**Table 22: Dialog box controls *(continued)***

| Defining word | Description |
|---|---|
| **[AUTORADIOBUTTON** | A radio button in a group, of which only one can be on. |
| **[CHECKBOX** | A box that can be checked. |
| **[COMBOBOX** | Combination of a single-line edit control and list box. |
| **[CTEXT** | Centered text. |
| **[DEFPUSHBUTTON** | Default pushbutton (Enter key selects it). |
| **[DRAWNBUTTON** | Owner-drawn pushbutton. |
| **[EDITBOX** | Box (usually white) in which you can type multiple lines. |
| **[EDITTEXT** | One-line edit control. |
| **[GROUPBOX** | A thin line around multiple controls to make a visual grouping. |
| **[HSCROLLBAR** | Horizontal scroll bar. |
| **[ICON** | An icon image (e.g., in upper-left corner) |
| **[LISTBOX** | Drop-down list. |
| **[LTEXT** | Left-justified text. |
| **[PROGRESS** | A window that an application can use to indicate the progress of a lengthy operation. See Section 9.3. |
| **[PUSHBUTTON** | Non-default pushbutton. |
| **[RADIOBUTTON** | Radio (round) button. |
| **[RICHBOX** | A *rich text* edit box; similar to an edit box, but with more features. |
| **[RTEXT** | Right-justified text. |
| **[STATE3** | Checkbox without pre-defined behavior. |
| **[STATIC** | Provides the user with text and graphics that typically require no response. |
| **[TEXT1BOX** | Single-line, read-only text box. |
| **[TEXTBOX** | Multi-line, read-only text box. |
| **[TRACKBAR** | A window that contains a slider and optional tick marks. When the user moves the slider, using either the mouse or the direction keys, the trackbar sends notification messages to indicate the change. |
| **[UPDOWN** | The little up and down arrows often attached to a number in a single-line edit box. |
| **[VSCROLLBAR** | Vertical scroll bar. |

```
\  [control        " default text"   id   xpos ypos xsiz ysiz ]
   [DEFPUSHBUTTON  " OK"             IDOK   105   20   45   15 ]
   [PUSHBUTTON     " Clear"          103     05   20   45   15 ]
   [PUSHBUTTON     " Throw"          104     55   20   45   15 ]
   [RTEXT                            101     05   05   18   10 ]
   [LTEXT          " Total errors"   102     25   05   50   10 ]
```

These provide a *default pushbutton* (the one "armed" to respond if the user presses Enter), right-justified text which is blank (in this example, it is where the number of presses will be displayed), left-justified text, and two more pushbuttons. The OK pushbutton is an example of a standard Windows button; **IDOK** is a Windows constant.

The controls provided by SwiftForth are described in Table 22.

### 9.2.4 Dialog Box Events

When the user does something to your dialog box, you need definitions that will respond to these events. Here are examples from **simple.f**:

```
: SIMPLE-CLOSE ( -- res )        \ Close the dialog box
   HWND 0 EndDialog ;

VARIABLE PRESSES

: .PRESSES ( -- )                \ Display presses counter
   HWND 101 PRESSES @ 0 SetDlgItemInt DROP ;
```

In these definitions, note that **HWND** is the handle of the dialog box, set by the callback that indicates a user action in the box; in **.PRESSES**, 101 is the numeric ID of the text control where the number will be displayed.

When an event (such as a user mouse press) occurs in the box, Windows will send it a message, which will be processed by the switches below. The switch **SIMPLE-MESSAGES** handles the messages, which close the box, start it up, or represent a command, respectively. If the message is a command, it is masked and sent to the switch **SIMPLE-COMMANDS**, which can close it or (in the last case) increment the presses counter.

```
[SWITCH SIMPLE-COMMANDS ZERO  ( -- res )
   IDOK     RUN: SIMPLE-CLOSE ;
   IDCANCEL RUN: SIMPLE-CLOSE ;
   103      RUN: PRESSES OFF  .PRESSES 0 ;
   104      RUN: STUPID 0 ;
SWITCH]

[SWITCH SIMPLE-MESSAGES ZERO
   WM_CLOSE      RUNS SIMPLE-CLOSE
   WM_INITDIALOG RUN: ( -- res )   0 PRESSES !  .PRESSES  -1 ;
   WM_COMMAND    RUN: ( -- res )   WPARAM LOWORD SIMPLE-COMMANDS ;
SWITCH]
```

The next step is to define the actual callback routine for the dialog box:

```
:NONAME ( -- res ) MSG  LOWORD SIMPLE-MESSAGES ; 4 CB: RUNSIMPLE
```

The code takes the message (returned by **MSG**), masks it, and sends it to the switches. This callback takes four parameters, and is named **RUNSIMPLE**.

Finally, we have the definition that instantiates the dialog box, the word you type in the command window to launch it:

```
: SIMPLE
   HINST (SIMPLE) HWND RUNSIMPLE
   0 DialogBoxIndirectParam DROP ;
```

Note that *this* use of **HWND** refers to the command window, since that's where you type **SIMPLE**! This establishes the command window as the "owner" of this dialog box.

## 9.3 Progress Bars

SwiftForth provides a simple way of generating and managing *progress bars* that show the progress of an operation that may take several seconds or longer. A progress bar is a specific form of dialog box. Progress bar support may be found in the file **SwiftForth\lib\options\win32\progress.f**.

To launch a progress bar, use the form:

```
Z" <title text>" +PROGRESS
```

...where *title text* will be displayed on the title bar. When you are ready to close it, you may do so with **-PROGRESS**. SwiftForth supports only one progress bar at a time using these words.

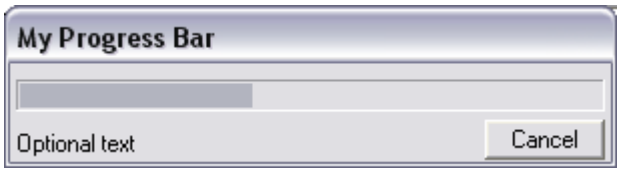


**Figure 24. Progress bar**

In the progress bar are two pre-defined areas in which you can display information. The most important is the horizontal area where you show progress toward completion of an operation. You may write to this area using the form:

```
<n> .PROGRESS
```

...where *n* is a number in the range 0–100, representing the percentage of completion, which will be represented as a filled percentage of the area.

The second writable region is a text area just below the progress area. You may put a message there using the form:

```
Z" <message text>" PROGRESS-TEXT
```

This will display *message text*.

The progress bar in Figure 24 was constructed using the following sequence:

```
Z" My Progress Bar" +PROGRESS
40 .PROGRESS
Z" Optional text" PROGRESS-TEXT
```

You may wish to display progress data with your text message. The easiest way to do this is to concatenate your data with the text portion of your message in an ASCIIZ string that you then pass to **PROGRESS-TEXT**. Here's an example:

```
99 (.) PAD ZPLACE
S"  seconds remaining" PAD ZAPPEND
PAD PROGRESS-TEXT
```

Here we used **(.)** to convert the integer to a string, leaving its address and length. The string was placed at **PAD**, and the text message appended to it. Note the extra space at the beginning of the text string, to provide one space after the number and before the text.

If the user presses Cancel while the operation is in progress, SwiftForth issues an **IOR_BREAK THROW**. To process this (and other possible **THROW**s) in your application code, execute the code containing the progress bar from within a **CATCH** (see Section 4.7). Otherwise, SwiftForth will simply abort the current operation, which is equivalent to pressing the Break button on the toolbar.

*Glossary*

**+PROGRESS** *( addr — )*
Launch a progress bar, with the ASCIIZ string whose address is given displayed in the title bar.

**-PROGRESS** *( — )*
Close a current progress bar, if there is one. Does nothing otherwise.

**.PROGRESS** *( n — )*
Display gray bars in the horizontal progress area reflecting *n* percent (0–100) completion.

**PROGRESS-NAME** *( addr — )*
Display the ASCIIZ string, whose address is given, in the title bar (replacing any previous title).

**PROGRESS-TEXT** *( addr — )*
Display the ASCIIZ string, whose address is given, in the text field below the progress area.

**(.)** *( n — addr u )*
Convert *n* to characters, without punctuation, as for **.** (dot), returning the address and length of the resulting string.

*References*
Output number conversions, *Forth Programmer's Handbook*
String operations in SwiftForth, Section 4.5.1
Exception handling and **THROW** codes, Section 4.7

## 9.4 SwiftForth's Status Bar

SwiftForth's status bar, described in Section 2.3.1, contains six parts, of which three

are used (for the stack display, current number conversion base, and editing mode). The stack display region resizes dynamically when the command window is resized; the others are of constant size.

Each part can display a string, and can respond to left or right mouse clicks in its area. You may change the behavior of any of these parts, using integers 0 through 5 to select a part (numbered from left to right). For example, to display the message "System Ready" to the right of the number base, you could use:

```
Z" System Ready" 2 .SPART
```

To set a mouse-click behavior, store the *xt* of a word to be executed in the appropriate status bar control cell. You can get the correct address using:

```
<part#> SBLEFT  or  <part#> SBRIGHT
```

For example:

```
: LEFT   Z" Left button" 2 .SPART ;
: RIGHT  Z" Right button" 2 .SPART ;
: PART2  ['] LEFT 2 SBLEFT !  ['] RIGHT 2 SBRIGHT ! ;
```

To test, execute **PART2** and click in the region to the right of the stack display.

*Glossary*

**.SPART** *( addr n — )*

Display, in part *n* (0 through 5) of the status bar, the zero-terminated string located at *addr*.

**SBLEFT** *( n — addr )*

Return the address of the cell containing the *xt* of the behavior for a left mouse click in status bar part *n*.

**SBRIGHT** *( n — addr )*

Return the address of the cell containing the *xt* of the behavior for a right mouse click in status bar part *n*.

# Section 10: 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.

## 10.1 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.

### 10.1.1 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 five 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 10.1.4**.**
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 10.1.6.
5. **Messages** are un-named responses to arbitrary numeric message values (such as Windows messages), and are discussed in Section 10.1.7.

*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.

### 10.1.2 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 acts 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 10.3.1)—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 context 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>

### 10.1.3 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. This is particularly useful in Windows programming, as these objects can be used in Windows callback routines.

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**.

**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*

### 10.1.4 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.

### 10.1.5 Information Hiding

Thus far, all named members of 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 10.1.1, 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.

### 10.1.6 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 10.1.1).

**SUPREME** *( — hclass )*

Return the handle of the ultimate superclass in SWOOP. All classes defined by **CLASS** are subclasses of **SUPREME**.

### 10.1.7 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*. This strategy is particularly useful for handling the vast number of Windows message constants and other numeric identifiers.

The form of a numeric message definition is:

```
<value> MESSAGE: <words to be executed> ;
```

For example, the class **SIMPLE-TEST** in the mouse-tracking example program **SwiftForth\unsupported\sfc\samples\mousecap.f** defines the following:

```
WM_CREATE  MESSAGE: ( -- res )   LPARAM
   DROP  TITLED  TRACKING OFF  0 ;
```

When the window is created, this code will set the title and initialize tracking.

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 exe-

cuted 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.

### 10.1.8 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 10.1.7), which sends an arbitrary message ID to an arbitrary object. This strategy makes it possible, for example, to send Windows message constants to a window object for processing.

Of the two strategies for using dynamic objects discussed in Section 10.1.3, **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**.

## 10.2 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.

### 10.2.1 Classes

The data representation of a class is shown in Figure 25. 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.

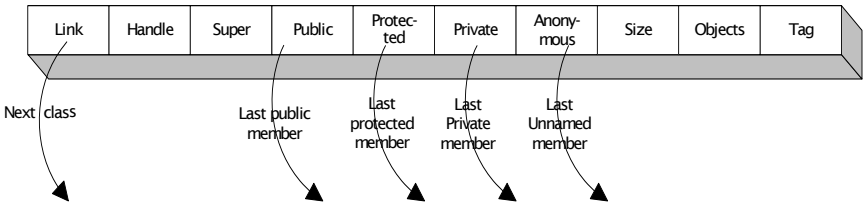


**Figure 25. Structure of a class**

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 10.1.5). 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 10.1.7), 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 10.3.1).

All links in this system are relative, and all handles are execution tokens (*xt*). This means that objects created in the interactive system at a given address will work when saved as a DLL, which is loaded at an arbitrary address by Windows.

### 10.2.2 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 26; the specific format of some member types is shown in Figure 27.

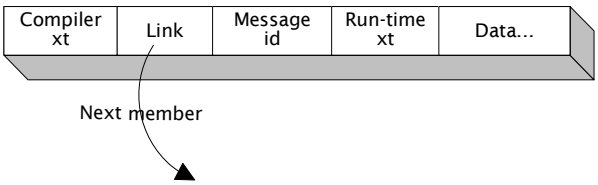


**Figure 26. Basic structure of a member**

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 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 *runtime-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.

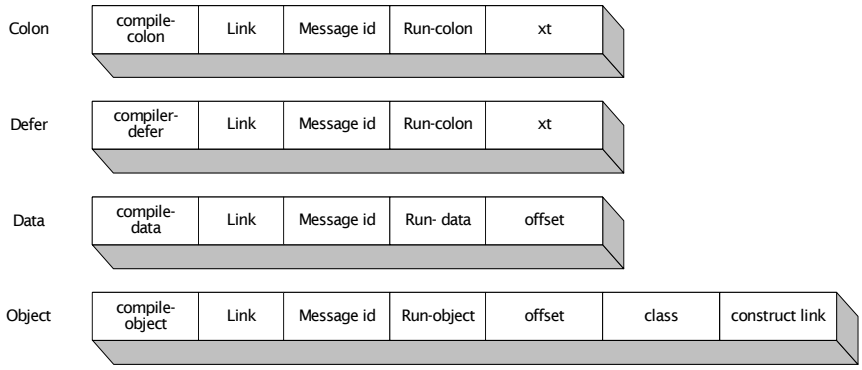


**Figure 27. Data structures for various member types**

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.

### 10.2.3 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**.

## 10.3 Implementation Strategies

Having discussed the basic syntax and data structures involved in SWOOP, we can now consider the underlying mechanisms in the system.

### 10.3.1 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 10.2.1.)

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 25. 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 reentrant.

*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, Section 7.2.1

### 10.3.2 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**).

### 10.3.3 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.

## 10.4 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 11: Windows Objects

SwiftForth includes a number of pre-defined object classes. Although some of these are for internal SwiftForth use only, many represent classes discussed in the Windows API and functions that are potentially useful to SwiftForth programmers. These are discussed in this section.

## 11.1 Standard Windows Data Structures

SwiftForth provides a number of classes consisting of standard Windows data structures described in the Win32API and other Windows reference materials. We have not attempted to implement an exhaustive set of such classes, but believe we have provided enough to serve as good examples. Those provided are listed in Table 23. Further information regarding the content and use of these structures may be found in Windows documentation.

The Forth names for the members of these data structures differ from the names in Windows documentation, although their sizes and use conform exactly. To see the details of these structures, use the command:

```
EDIT <classname>
```

**Table 23: SwiftForth classes for Windows data structures**

| Class Name | Description |
|---|---|
| **BITMAPHEADER** | **BITMAPFILEHEADER** plus **BITMAPINFOHEADER**. The **BITMAPFILEHEADER** structure contains information about the type, size, and layout of bitmap; the **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a bitmap file. |
| **CHARFORMAT** | Contains information about character formatting in a rich edit control. |
| **CHARRANGE** | Specifies a range of characters in a rich edit control. |
| **CHOOSE-FONT** | Contains information that the **ChooseFont** function uses to initialize the Font common dialog box. After the user closes the dialog box, the system returns information about the user's selection in this structure. Note that this is renamed from Windows' **CHOOSE-FONT** to avoid conflict when case-insensitive. |
| **DEVMODE** | Contains information about the device initialization and environment of a printer. |
| **DOCINFO** | Contains the input and output filenames and other information used by the **StartDoc** function. |
| **EDITSTREAM** | Contains information about a data stream used with a rich edit control. |
| **FILETIME** | A 64-bit value representing the number of 100-nanosecond intervals since January 1, 1601. |

**Table 23: SwiftForth classes for Windows data structures *(continued)***

| Class Name | Description |
|---|---|
| **FORMATRANGE** | Contains information that a rich edit control uses to format its output for a particular device. This structure is used with the **EM_FORMATRANGE** message. |
| **LOGICAL-FONT** | Defines the attributes of a font. Note that this is renamed from Windows' **LOGFONT** to avoid conflict when case-insensitive. |
| **NMHDR** | Contains information about a notification message. The pointer to this structure is specified as the **lParam** member of the **WM_NOTIFY** message. |
| **OPENFILENAME** | Contains information that the **GetOpenFileName** and **GetSaveFileName** functions use to initialize an Open or Save As… dialog box. After the user closes the dialog box, the system returns information about the user's selection in this structure. |
| **PAGESETUPDIALOG** | Contains information the **PageSetupDlg** function uses to initialize the Page Setup common dialog box. After the user closes the dialog box, the system returns information about the user-defined page parameters in this structure. Note that this is renamed from Windows' **PAGESETUPDLG** to avoid conflict when case-insensitive. |
| **PARAFORMAT** | Contains information about paragraph formatting attributes in a rich edit control. This structure is used with the **EM_GETPARAFORMAT** and **EM_SETPARAFORMAT** messages. |
| **POINT** | Defines the x- and y- coordinates of a point. |
| **PRINTDIALOG** | Contains information used by the **PrintDlg** function to initialize the Print common dialog box. After the user closes the dialog box, the system returns information about the user-defined print selections in this structure. Note that this is renamed from Windows' **PRINTDLG** to avoid conflict when case-insensitive. |
| **RECT** | Defines the coordinates of the upper-left and lower-right corners of a rectangle. |
| **REQRESIZE** | Contains the requested size of a rich edit control. A rich edit control sends this structure to its parent window as part of an **EN_REQUESTRESIZE** notification message. |
| **RGBQUAD** | Describes a color consisting of relative intensities of red, green, and blue. |
| **TEXTMETRIC** | Contains basic information about a physical font. All sizes are given in logical units; that is, they depend on the current mapping mode of the display context. |
| **WIN32_FIND_DATA** | Describes a file found by the **FindFirstFile** or **FindNextFile** function. |
| **WNDCLASS** | Contains the window class attributes registered by the **RegisterClass** function. |

## 11.2 Example: File-Handling Dialogs

An excellent example of the use of SWOOP objects to support Windows functions may be found in SwiftForth's support for file-handling functions, **INCLUDE**, **EDIT**, etc. All of these feature a need to let the user "browse" to a particular file or directory path, a function for which Windows provides a basic customizable dialog box.

The class hierarchy used to implement these functions is shown in Figure 28.

The basic factoring of the code to support these file functions includes the following layers:

1. The Windows data structure **OPENFILENAME**.
2. The sub-class **FILENAME-DIALOG** that provides all the common support functionality required for performing the set of functions needed by SwiftForth based on the **OPENFILENAME** data structure. The features that differ (the actual Windows function to be invoked, various options specified by flags, the names of the dialog boxes) are factored into the two **DEFER** members, **CUSTOM** and **ACTION**. This class also provides the critical word **CHOOSE** that manages the overall operation of the dialog box, presenting the dialog box configured by **CUSTOM** for the user's choice, and calling **ACTION** to implement that choice.
3. Two subclasses of **FILENAME-DIALOG** that differ only in that they provide alternative definitions for **ACTION**, consisting of the two possible Windows calls.
4. Subclasses of each of these two classes that specify individual names for the dialog box and flags denoting each one's specific functions.
5. Program interface words, that can be called in response to user commands, menu selections, etc., that locally instantiate and use each of the five lowest-level subclasses in the hierarchy.

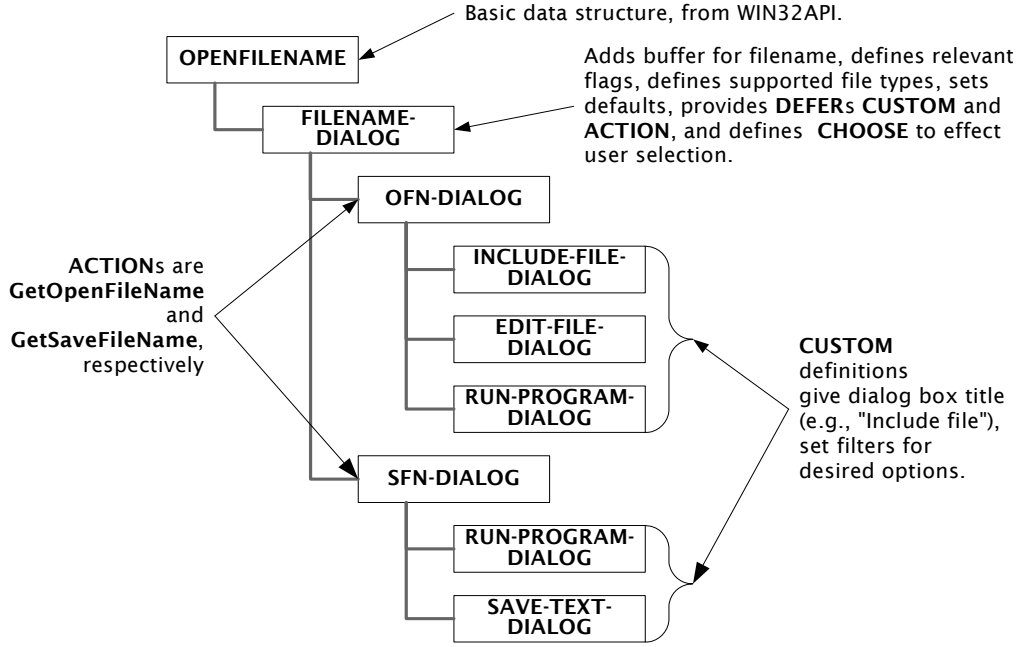


**Figure 28. Class hierarchy supporting file-handling features**

The program interface words vary, but all follow a general pattern of which the simplest example is **CHOOSE-EDIT-FILE** (part of the response to the File > Edit menu selection or toolbar button):

```
: CHOOSE-EDIT-FILE ( -- )
   [OBJECTS
        EDIT-FILE-DIALOG MAKES EFD
   OBJECTS]
   EFD CHOOSE IF
      0 EFD FILENAME ZCOUNT EDIT-FILE
   THEN ;
```

In this definition, we see a local instantiation of an **EDIT-FILE-DIALOG** object, which is used to obtain a filename to be passed to **EDIT-FILE**, in order to proceed with issuing the message to your linked editor to open the file.

## 11.3 Color Management

SwiftForth supports color selection for various Windows features. These include names for the 20 standard Windows colors, and examples showing how SwiftForth uses them to manage colored items like the screen background and text, highlighted text, etc.

The color names and general support words may be found in the file **SwiftForth\src\ide\win32\colors.f**. The dialog box used to select colors for SwiftForth is in **SwiftForth\src\ide\win32\pkcolor.f**. SwiftForth uses five named color styles for text, described in Table 24.

**Table 24: SwiftForth color attributes**

| | | Defaults | |
|---|---|---|---|
| **Name** | **Description** | **Background** | **Foreground** |
| **NORMAL** | Normal text display. | White | Black |
| **INVERSE** | Selected items | Black | White |
| **BOLD** | Highlighted items | White | Red |
| **BRIGHT** | Highlighted items | White | Blue |
| **INVISIBLE** | Invisible text | White | White |

These may be used before any output word (such as **.**, **."**, or **TYPE**) to specify the way the next output of text will be displayed. Each of these modes will stay set until changed. **NORMAL** is the default, and need not be specified unless you've selected another mode. For example, the word **EMPHASIZED** will display a string in **BOLD** style:

```
: EMPHASIZED ( addr len -- )
   BOLD TYPE NORMAL ;
```

To preserve **NORMAL** as the default, always restore the style to **NORMAL** when you change it temporarily!

## 11.4 Rich Edit Controls

SwiftForth includes support for rich edit controls in the file **SwiftForth\src\ide\win32\richedit.f**. Primary documentation for rich edit controls may be found in the Win32API. This section provides a brief description of SwiftForth's support for these features.

Rich edit controls support almost all of the messages and notification messages used with multi-line edit controls. Applications that already use edit controls can be easily changed to use rich edit controls. An application can send messages to a rich edit control to perform such operations as formatting text, printing, and saving. In addition, an application can process notifications to filter keyboard and mouse input, to permit or deny changes to protected text, or to resize the control as needed to fit its content. Rich edit controls support most of the window styles used with edit controls as well as additional styles, including font selection, font styles (e.g., bold, italic), and many other features.

The two classes defined in **richedit.f** are:

- **PARAFORMAT** manages paragraph formatting issues including indentation, margin offsets, tabs, alignment, and bullets or numbering.
- **CHARFORMAT** manages character formatting issues such as font, size, color, style (bold, italic, etc.).

The various specifications used by these classes are controlled by bit masks defined as constants, using the names given in the Win32API. These attributes may be combined to provide masks that describe the current settings for paragraphs and characters, respectively. Both classes include methods **GET** and **SET** which issue the Windows calls to fetch and set current masks, as well as methods for adding and subtracting various bit masks representing features.

All sizes used in rich edit controls are specified internally in units known as *twips*. A twip is a unit of measurement equal to 1/20th of a printer's point. There are 1440 twips to an inch, 567 twips to a centimeter.

The words **INCHES**, **POINTS**, and **CMS** are provided to scale numbers in more convenient units to twips by providing the appropriate conversion factors. These words will automatically scale the input number if it has any decimal places. For example, the phrase:

```
10 POINTS HIGH
```

…will set the height of the current font, while the phrase:

```
.25 INCHES INDENT-LEFT
```

…will set the left margin of paragraphs to 1/4 in.

For details on the commands provided, refer to the file **richedit.f**.

## 11.5 Other Available Resources

In addition to the predefined data structures discussed in Section 11.1, SwiftForth includes a number of other pre-defined classes that may be helpful to you. Some of these are described in Table 25.

**Table 25: More useful pre-defined classes**

| Class Name | Description |
|---|---|
| **I-TO-Z** | A class supporting output formatting of integers to zero-terminated strings without using global system resources such as the standard Forth number-conversion buffer. |
| **FILENAME-BUFFER** | A buffer whose size is **MAX_PATH**, used to contain filenames with path information. |
| **FONT-PICKER** | A subclass of the data structure **CHOOSE-FONT** that provides the logic for handling the common dialog box for selecting a font. |

Here's an example showing the use of the **FONT-PICKER** class to specify a font for an arbitrary window with text in it:

1. Load and start a sample application. For example:
   ```
   REQUIRES CLICKS
   START DROP
   ```
   (the **DROP** discards an unneeded application handle)

2. Create a font data structure:
   ```
   FONT-PICKER BUILDS CLICKS-FONT
   ```
3. Create a font description for the application window:
   ```
   HAPP CLICKS-FONT SELECT ( -- flag ) DROP
   ```
   This sends the application handle **HAPP** to the **CLICKS-FONT** member of **FONT-PICKER**, which returns a result flag.

4. Create the actual font by passing the address of the data structure to Windows, which returns a handle:
   ```
   CLICKS-FONT 'FONT CreateFontIndirect ( -- hfont )
   ```
   **'FONT** is another member of **FONT-PICKER** that returns the font size.

5. Make the application use the new font:
   ```
   HAPP GetDC SWAP SelectObject DROP
   ```
   This passes the font handle from Step 4 and device context to **SelectObject**, which installs the font. The window is now using the selected font.

# Section 12: Floating-Point Math Library

SwiftForth's floating-point math library is a system option providing support for the Intel Architecture Floating-Point Unit (FPU). See Section 2.3.5 for instructions for configuring SwiftForth to use this option.

## 12.1 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.

## 12.2 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 the Options > Optional Packages > System Options dialog box. If this file is included, the FPU will be initialized as part of the system initialization. If you will be using this package regularly, you may find it convenient to make a turnkey program containing it, as described in Section 2.3.5.

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 dis-

cussed 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.

### 12.2.1 Configuring the Floating-Point Options

The dialog box shown in Figure 29 is automatically displayed the first time you load the FPMath option, and may be recalled thereafter by using the Options > FPMath menu selection. It allows you to configure the settings described below.

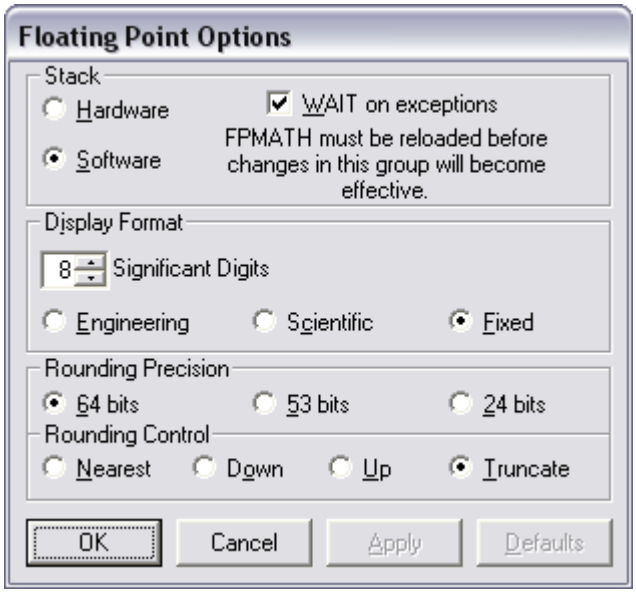


**Figure 29. Configure floating-point options**

*Stack:* The FPU has an on-board stack with a theoretical capacity of eight floating-point numbers. Of these eight positions, only seven are practically useful, and any output routine will consume at least one, leaving six for algorithmic use. If your algorithm can work within this limitation (and most can), your code will run significantly faster if you select the hardware stack. The software stack has a depth of 32 items.

Note: the FPU does not generate an exception on hardware stack overflow or underflow, which means these conditions cannot be automatically detected. SwiftForth provides the word **?FSTACK**, which **THROW**s on overflow or underflow. When the FPMath option is loaded, this is included in SwiftForth's normal stack check in the text interpreter, and also in the floating output routines **FS.**, **FE.**, etc. You may wish to add it to your application code in routines that may require more than a few stack positions.

*WAIT on exceptions,* when checked, causes an **FWAIT** instruction to be executed following any FPU instructions, in order to ensure that any FPU exception will be detected before any non-FPU instructions are executed. This option is useful during

debugging, but will slow the code slightly.

Note: any change in the above two options will not take effect until the floating-point option is re-loaded.

*Display Format:* This section configures the output word **N.** (Section 12.2.3) by selecting the number of significant digits to be displayed and the format. The Engineering option selects **FE.**, Scientific selects **FS.**, and Fixed selects **F.** (all discussed in the *Forth Programmers' Handbook*). Specification of output format has no effect on the internal representation of floating-point numbers.

*Rounding Precision:* The internal format maintains a 64-bit mantissa and 16-bit exponent (80 bits total). If you convert to a single or double integer format, rounding will occur. This option specifies the size of the mantissa: 64 bits is full precision, 53 bits corresponds to a double integer (64 bits total), and 24 bits corresponds to a single integer (32 bits total).

*Rounding Control:* The SwiftForth default is to truncate. The configurable alternatives are:

- Round-to-nearest, and at the boundary round to even.
- Round down, or floor (1.5 rounds to 1 and -1.5 rounds to -2).
- Round up (-1.5 would round to -1).

Rounding operations are always done with the default precision and exception masking. These can be changed by adjusting the values in **FP-ROUND**. Note that, per ANS Forth requirements, the word **F>D** (page 190) will *always* truncate, as will the non-ANS Forth word **S>D**.

### 12.2.2 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

### 12.2.3 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*.

### 12.2.4 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.

### 12.2.5 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.

### 12.2.6 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"

#### 12.2.6.1 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.

#### 12.2.6.2 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.

## 12.3 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 12.3.3.3 for further references.

*References* i386 assembler, Section Section 6:

### 12.3.1 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.

### 12.3.2 CPU Synchronization

The FPU assembler functions as part of the SwiftForth i386 assembler. FPU instructions are assembled in the same manner and format.

For maximum throughput, FPU instructions are not synchronized with the CPU 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**.

### 12.3.3 Addressing Modes

The FPU employs two types of addressing modes:

- memory reference
- hardware stack top relative

Memory reference modes use the i386 operand formats, whereas the hardware stack top relative mode is unique to the FPU.

#### 12.3.3.1 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 26.

**Table 26: 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
```

#### 12.3.3.2 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 27.

**Table 27: 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**. |

#### 12.3.3.3 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 13: Recompiling SwiftForth

There are two binary program files included in the SwiftForth distribution; all three are installed in the SwiftForth\bin directory:

- `sfk.exe` — The SwiftForth kernel with no extensions loaded. This is a "bare-bones" Windows console application.
- `sf.exe` — The extended SwiftForth interactive development environment. Includes many features beyond the basic kernel: debug tools, Windows API interface, SWOOP, turnkey program generator, and DLL exports, just to name a few. This is a "turnkey" Windows GUI application.

The complete source code for all of these components is supplied with the SwiftForth distribution. The following sections detail how to recompile part or all of the SwiftForth binaries.

## 13.1 Recompiling the SwiftForth Turnkey

Launch the SwiftForth kernel console window, **SwiftForth\bin\sfk.exe**. 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\win32**.

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 SwiftForth.

After loading the electives, save a new turnkey with the **PROGRAM** command as detailed in Section 4.1.2.

The batch file **SwiftForth\bin\turnkey.bat** is supplied as an example of how to recompile the **sf.exe** turnkey program.

## 13.2 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.exe is in **SwiftForth\xcomp**. To recompile the kernel, launch the full sf.exe turnkey (sfk.exe by itself does not have enough extensions loaded to support the target compiler) with the working directory set to **SwiftForth\src\kernel\win32**, then include the "make" file:

```
INCLUDE MAKE
```

The output will be saved in **SwiftForth\bin\sfk.exe** 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.

The batch file **SwiftForth\bin\make.bat** recompiles the kernel and then uses the new kernel to recompile the sf.exe turnkey program. Before recompiling the kernel, it pops up a Notepad editor window in which the release version number can be edited. If you're making your own customize kernel, append something (like "-L1") to the end of the version string to indicate that this is a custom "local" version. We recommend that you do not change the version number itself.

# Appendix A: Block File Support

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, most modern Forth implementation run under general-purpose, file-oriented operating systems. In order to maintain portability between native and OS-based Forths, the concept of block-oriented disk access was preserved, but blocks reside 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 block-handling features, use the Tools > Optional Packages > Generic Options dialog box. Select BLOCKEDIT to load the full block-support package including the block editor. Select BLOCKS to load block support, but not the block editor. (You can start with BLOCKS and add BLOCKEDIT later with no penalty.)

From a source file, you can load the optional block and block editor support like this:

**REQUIRES BLOCKEDIT**
...to load the full block-support package, including the block editor.

**REQUIRES BLOCKS**
...to load block support without the editor.

In SwiftForth, Forth blocks are held in 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 Windows files is called the *blockmap* and is maintained in an array named **PARTS**. For each file, the blockmap contains:

- a reference to the filename
- the file handle used for OS calls
- 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 relationship. 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 28. Each item occupies one cell.

**Table 28: 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** | Fail access mode (R/O, R/W, etc.) |
| **#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 Windows 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 Windows 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 modi-

fying. If a filename extension is given, it will be used; if it is omitted, the extension **.src** will be added. For example:

```
60 NEWFILE My
```

creates a new file called **My.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.

**?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 29 summarizes buffer usage.

**Table 29: 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 30: 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 2.3.5). 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 30: 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.<br>**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, in a dialog box and the command window. |
| 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<br>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 |

**Table 30: Implementation-defined options in SwiftForth *(continued)***

| Option | SwiftForth Support |
|---|---|
| Size of one character in address units (**3.1.2 Character types**) | 1 |
| 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 2.3.5). 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 31: 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 Windows. |
| 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. |

**Table 31: SwiftForth action on ambiguous conditions *(continued)***

| Condition | Action |
|---|---|
| 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 |
| 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** |

**Table 31: SwiftForth action on ambiguous conditions *(continued)***

| Condition | Action |
|---|---|
| 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) |
| 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 32: 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 and Section 2.4.8 |

**Table 32: Other system documentation *(continued)***

| Requirement | SwiftForth information |
|---|---|
| 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 |
| System dictionary space required, in address units | About 300K bytes |

## B.2 Block Wordset Documentation

**Table 33: 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 34: 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 35: 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 Section 7.2.2). |
| The number of blocks available for source text and data | User specified; see Section A.1. |

## B.3 Double Number Wordset Documentation

**Table 36: 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 37: 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 the source file: **SwiftForth\src\ide\errmessages.f** |

## B.5 Facility Wordset Documentation

**Table 38: 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, depending on what else is<br>running under Windows. |

**Table 39: 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 40: 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"<br>-37 " File I/O exception"<br>-38 " Non-existent file"<br>-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**) | Windows 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**)<br>-192 Can't rename file (**RENAME-FILE**)<br>-193 Can't resize file (**RESIZE-FILE**)<br>-194 Can't flush file (**FLUSH-FILE**)<br>-195 Can't read file (**READ-FILE**, **READ-LINE**)<br>-196 Can't write file (**WRITE-FILE**)<br>-197 Can't close file (**CLOSE-FILE**)<br>-198 Can't create file (**CREATE-FILE**)<br>-199 Can't open file (**OPEN-FILE**, **INCLUDE-FILE**)<br><br>These always return 0 (success, or ignore and continue)<br>**FILE-POSITION**<br>**REPOSITION-FILE**<br>SEEK-FILE<br>**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 40: 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 41: 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 42: 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 12.1. |
| Results of **12.6.1.2143 REPRESENT** when *float* is out of range | FPU trap |

**Table 42: 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 12.2. |
| 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 43: 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 43: 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 44: 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 45: 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 46: 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 47: 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 48: 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 49: 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 50: 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: Glossary of Windows Terms

- *API* — generally speaking, a set of function calls through which a system provides services to applications. For example, Forth provides a serial API consisting of **TYPE**, **EMIT**, **KEY**, **ACCEPT**, etc. In this context, however, we're usually referring to a very specific API called Win32. This is a vast collection of functions (over 3,500) provided with current Windows versions.
- *Callback* — an entry point associated with a window capable of processing a message from the OS. A pointer to a callback is often passed to Windows as part of a call from SwiftForth, so that Windows can respond directly, for example, with requested information. Some callbacks handle a single function; more commonly, a callback must determine which of several possible messages has been received and respond accordingly.
- *Control* — a special type of window[1] supporting a specific user interaction. Examples include pushbuttons, scroll bars, edit boxes, checkboxes, and radio buttons.
- *Device context* — a data structure associated with Windows' "Graphics Device Interface" (GDI, a subset of the Win32API). A device context (sometimes abbreviated DC) controls a particular display device, such as a printer, plotter, or a window. Values in a device context are called *attributes*, and define how various GDI drawing functions will behave for this particular device.
- *Dialog box* — a type of window that supports interaction between the program and the user, typically via controls.
- *DLL* — a "dynamically linked library," a binary image of code in a format that can be linked to your program, temporarily or permanently, to perform certain services. A DLL's services are made available through its *exports* or entry points, which must be defined in your program as *imports*. Windows provides hundreds of DLLs, and most programs or libraries come with still more.
- *Focus* — The place in an active window or dialog box where the next action will be recorded. For example, if there are several boxes where you can type, the one with focus is where the next keystroke will go. Similarly, of several windows that might be open at one time, the one with focus is the active one.
- *Handle* — a number used to reference an object. Windows, files, and other objects all have handles assigned by the OS when the object is opened or created. Handles are important message parameters. A window handle is called **HWND**, and a device context handle is called **HDC**.
- *Message* — in addition to the specific meaning this has in SWOOP and in object-oriented programming in general, this is a communication from the OS to the application, usually reporting some event such as a keypress or mouse click. Windows calls an entry point you provide in your program, sending information about the event. A message is analogous to an interrupt in non-Windows environments.
- *Message loop* — code in your program that retrieves messages from the message queue and dispatch them to the appropriate callback routine. Just as a message is analogous to an interrupt, the message loop is analogous to an interrupt-polling routine which routes interrupts to the appropriate service code.

1.This term potentially is confusing because the technical definition of "window" is "a dataset with a callback," therefore controls qualify as a type of window.

- *Message queue* — a path set up by the OS when your program begins execution, by which your program can receive messages. Certain classes of messages are sent directly to the appropriate callback, but most go into the queue, awaiting processing by your program's message loop.
- *Process* — a program launched by Windows. It has various resources, including program memory, stacks, the ability to open files, etc. A process has at least one execution thread and usually (though not always) a message loop; it may launch other threads, which will share its program memory and open files, but which may have private stacks and message loops. If Windows launches multiple instances of a program, each of them is a process.
- *Thread* — normally a subordinate component of a process, with some resources of its own but usually executing some of its parent process' code, and executing concurrently with other threads or programs. A process has at least one thread, but may also launch additional threads that all share resources (such as program memory and open files) with their parent process. A thread may have one or more windows, in which case it may have its own message loop (although it may also continue to depend on its parent process' message loop). A thread with no windows (e.g., doing background number crunching) doesn't need a message loop. A thread has its own stacks, may have local variables, and has its own processor state that is saved and restored during thread switches.
- *Window* — used in many contexts with a bewildering set of meanings, it specifically refers to a rectangular area on the screen maintained by and communicating with the OS on behalf of a running application. The term is also applied to functioning elements (more properly referred to as "child windows") within a main window, such as menus, toolbar, buttons, etc.
- *Window class* — a style or type of window, defined by parameters that control the appearance and behavior of windows belonging to that class. Thus, windows of a particular class will share common characteristics; a sub-class may be created to augment or alter those characteristics. To make a window, you first define its class, register the class with the OS, and then construct one or more instances of the class (which will actually appear and begin to function).
- *Window procedure* — a procedure associated with a window, often abbreviated as "WinProc." When Windows sends a message to your program regarding a window, it is calling its relevant procedure, which could be in your program itself, or in a DLL.

# Appendix D: 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 table does not include Windows messages, constants, or procedures, although these can be referred to in SwiftForth as though they were defined Forth words. They are documented in Windows documentation or the Windows API Help system.

The stack-argument notation is described in Table 51. Where several arguments are of the same type, and clarity demands that they be distinguished, numeric subscripts are used.

**Table 51: 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).<br>On 32-bit machines, the range is from -2**63 through +2**63-1. |
| *flag* | A single-precision Boolean truth flag<br>(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 12.2.2. |
| *u* | A single-precision, unsigned number with a range from<br>0 through 2**32-1 on 32-bit machines. |

**Table 51: Notation used for data types of stack arguments *(continued)***

| Notation | Description |
|---|---|
| *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 52: Index of Forth Words

**Table 52: Index of Forth Words**

**Table 52: Index of Forth Words**

**Table 52: Index of Forth Words**

**Table 52: Index of Forth Words**

**Table 52: Index of Forth Words**

**Table 52: Index of Forth Words**

**Table 52: Index of Forth Words**

| Word | Stack | Page |
|---|---|---|
| **NEWFILE** <name> | *( n — )* | *198* |
| **NEXT-WORD** | *( — addr u )* | *76* |
| **NH** | *( — addr )* | *62* |
| **NOT** | *( $cc_1$ — $cc_2$ )* | *115* |
| **NOW** | *( ud — )* | *65* |
| **NTH_PARAM** <name> | *( n — )* | *129* |
| **NUMBER** | *( addr u — n \| d )* | *61* |
| **NUMBER?** | *( addr u — 0 \| n 1 \| d 2 )* | *61* |
| **OBJECTS]** | *( — )* | *163* |
| **OCTAL** | *( — )* | *61* |
| **OFF** | *( addr — )* | *58* |
| **ON** | *( addr — )* | *58* |
| **OPEN-PERSONALITY** | *( addr — )* | *95* |
| **OPTIONAL** <name> <description> | *( — )* | *47* |
| **ORIGIN** | *( — addr )* | *86* |
| **OV** | *( — cc )* | *114* |
| **PACKAGE** <name> | *( — tag )* | *91* |
| **PAD** | *( — addr )* | *68* |
| **PART** | *( n — )* | *197* |
| **PAUSE** | *( — )* | *125* |
| **PE** | *( — cc )* | *114* |
| **PLACE** | *( $addr_1$ u $addr_2$ — )* | *70* |
| **PO** | *( — cc )* | *114* |
| **POPPATH** | *( — )* | *48* |
| **POPUP** <label> | *( — )* | *149* |
| **PRIVATE** | *( tag — tag )* | *91* |
| **PRIVATE** | *( — )* | *165* |
| **PROGRAM** <filename>[.<ext>] [<icon>] | *( — )* | *56* |
| **PROGRAM-SEALED** <filename>[.<ext>] [<icon>] | *( — )* | *137* |
| **PROGRESS-NAME** | *( addr — )* | *156* |
| **PROGRESS-TEXT** | *( addr — )* | *156* |
| **PROTECTED** | *( — )* | *165* |
| **PUBLIC** | *( tag — tag )* | *91* |
| **PUBLIC** | *( — )* | *165* |
| **PUSHPATH** | *( — )* | *48* |
| **READONLY** | *( n — )* | *198* |
| **RELEASE** | *( addr — )* | *123* |

**Table 52: Index of Forth Words**

**Table 52: Index of Forth Words**

| Word | Stack | Page |
|---|---|---|
| **TERMINATE** | *( — )* | *125* |
| **THEN** | *( addr — )* | *113* |
| **TIME** | *( — )* | *64* |
| **TIME&DATE** | *( — $u_1$ $u_2$ $u_3$ $u_4$ $u_5$ $u_6$ )* | *64* |
| **TIMER** | *( u — )* | *66* |
| **TO** <name> | *( x — )* | *75* |
| **TOPIC** <string> | *( — )* | *144* |
| **U<** | *( — cc )* | *114* |
| **U<=** | *( — cc )* | *114* |
| **U>** | *( — cc )* | *114* |
| **U>=** | *( — cc )* | *114* |
| **UBETWEEN** | *( $u_1$ $u_2$ $u_3$ — flag )* | *76* |
| **uCOUNTER** | *( — d )* | *66* |
| **UDUMP** | *( addr u — )* | *39* |
| **UNCALLED** | *( — )* | *37* |
| **UNMAP** | *( n — )* | *197* |
| **UNMAPPED** | *( — n)* | *198* |
| **UNMAPS** | *( $n_1$ $n_2$ — )* | *197* |
| **UNTIL** | *( addr cc — )* | *113* |
| **UNUSED** | *( — n )* | *86* |
| **USING** <classname> | *( addr — addr )* | *163* |
| **USING** <name> | *( — n )* | *197* |
| **uTIMER** | *( d — )* | *66* |
| **VERBOSE** | *( — )* | *49* |
| **WARNING** | *( — addr )* | *57* |
| **WATCH** | *( addr — )* | *42* |
| **WHERE** <name>, **WH** <name> | *( — )* | *37* |
| **WHILE** | *( addr$_1$ cc — addr$_2$ addr$_1$ )* | *113* |
| **WINERROR** | *( — addr u )* | *79* |
| **WORDLIST** | *( — wid )* | *90* |
| **WPARAM** | *( — n )* | *129* |
| **Z"** <string> **"** | *( — addr )* | *69* |
| **Z\"** <string> **"** | *( — addr )* | *69* |
| **ZAPPEND** | *( addr$_1$ u addr$_2$ — )* | *70* |
| **ZERO** | *( x — 0 )* | *76* |
| **ZPLACE** | *( addr$_1$ u addr$_2$ — )* | *70* |

# Index

For a comprehensive index of SwiftForth commands, see Appendix D.