



Getting Results Faster...

SwiftX

Cross-Development Software

Reference Manual

FORTH, Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. FORTH, Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

SwiftX is a trademark of FORTH, Inc. All other brand and product names are trademarks or registered trademarks of their respective companies.

Copyright © 1998, 2000 by FORTH, Inc. All rights reserved.

First edition, January 1998

Second edition, January 2000

Printed 1/28/00

This document contains information proprietary to FORTH, Inc. Any reproduction, disclosure, or unauthorized use of this document, either in whole or in part, is expressly forbidden without prior permission in writing from:

FORTH, Inc.

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

CONTENTS

Welcome! xi

What is SwiftX? xi
Scope of This Book xi
Audience xii
How to Proceed xii
Support xii

1. Getting Started 1

1.1 Components of SwiftX 1

1.2 SwiftX System Requirements 2

1.3 Installation Instructions 3

1.3.1 Installing the Host Software 3
1.3.2 Linking to a Text Editor 4

1.4 Development Procedures 4

1.4.1 Starting a Development Session 5
1.4.2 Using SwiftX With Other Hardware 6
1.4.3 Running the Demo Application 6
1.4.4 Changing the SwiftX Kernel 7

2. Introduction to SwiftX 9

2.1 SwiftX Programming 9

2.2 System Organization 10

2.3 IDE Quick Tour 12

2.3.1 The Command Window 12

2.3.2 File Menu 16

2.3.3 Edit Menu 17

2.3.4 View Menu 17

2.3.5 Options Menu 18

2.3.6 Tools Menu 20

2.3.7 Project Menu 21

2.3.8 Help Menu 21

2.4 Interactive Programming Aids 22

2.4.1 Interacting With Program Source 22

2.4.2 Cross-references 25

2.4.3 Disassembler 25

2.4.4 Memory Dumps 26

2.4.5 Single-Step Debugger 27

2.4.6 Managing the Command Window 28

2.5 Host-Target Interaction 29

3. Programming with SwiftX 31

3.1 Source-File Management 31

3.1.1 Interpreting Source Files 31

3.1.2 Extended Comments 33

3.1.3 File Load Order 34

3.2 Program Development Strategy 34

3.2.1 Interactive Development 35

3.2.2 Building an Application for ROM 36

4. Cross-compiler Principles 39

4.1 Cross-compiler Control 39

4.1.1 Input Number Conversions 39

4.1.2 Conditional Compilation 40

4.2 Target System Configuration 41

4.3 Memory Allocation 42

4.4 Defining Words 46

4.5 Memory Access 48

4.5.1 Basic Memory Access Commands 48

4.5.2 String Initialization and Management 49

4.6 Compiler and Interpreter 50

4.6.1 Basics of Compilation 50

4.6.2 Compiler Scoping 52

4.6.3 Effects of Scoping on Colon Definitions 55

4.6.4 Effects of Scoping on Data Object Defining Words 57

4.7 Custom Data Objects 57

4.8 Saving a Compiled Object Image 61

4.9 The Cross-Target Link 62

4.9.1 Target/Image Interactions 63

4.9.2 XTL Protocol 66

5. The SwiftOS Multitasking Executive 71

5.1 Forth Re-entrancy and Multitasking 72

5.2 Principles of Operation 73

- 5.2.1 Task-Scheduling Algorithm 73
- 5.2.2 Interrupts and Tasks 76
- 5.2.3 User Variables 77
- 5.2.4 Sharing Resources 84

5.3 Background Tasks 86

- 5.3.1 Defining a Background Task 86
- 5.3.2 Initializing a Background Task 88
- 5.3.3 Controlling a Background Task 90

5.4 Terminal Tasks 93

- 5.4.1 Defining a Terminal Task 93
- 5.4.2 Initializing a Terminal Task 95
- 5.4.3 Controlling a Terminal Task 96

5.5 A Multitasking Demo 97**5.6 Comparing Background Tasks and Terminal Tasks 98****6. Target Libraries 99****6.1 Enhanced Number Conversion 99****6.2 Timing Functions 103**

- 6.2.1 Date and Time of Day Functions 103
- 6.2.2 Interval Timing 106
- 6.2.3 Benchmarks 108

6.3 Arithmetic Library 109

- 6.3.1 Double-Precision Arithmetic 109
- 6.3.2 Fixed-Point Fractions 110
- 6.3.3 Angles 111
- 6.3.4 Transcendentals 112
 - 6.3.4.1 Square Root 112
 - 6.3.4.2 Trigonometric Functions 112

7. Interpreter/Compiler Option 115

7.1 Configuring SwiftX for the Interpreter Option 115

- 7.1.1 Configuring the Target Dictionary 116
- 7.1.2 Configuring a Terminal Task for the Interpreter 117
- 7.1.3 Additional User Variables 120

7.2 Using the Interpreter 121

7.3 Using the Compiler 122

Appendix A: Bibliography 123

Appendix B: Programming Examples 125

B.1 The Washing Machine 125

- B.1.1 Top-down Program Design 126
- B.1.2 Hardware Control 127
- B.1.3 Code for the Washing Machine 129

B.2 The Conical Pile Calculator 130

- B.2.1 Algorithm Description 130
- B.2.2 Material Selection 136
- B.2.3 Testing the Calculator 137

Appendix C: Index to Forth Words 139

General Index 147

List of Figures

1. SwiftX directory structure 2
2. Interaction between host and target components 11
3. Code layers in a SwiftX system 12
4. The SwiftX command window 13
5. Toolbar items 14
6. Status bar indicators 14
7. Select Editor dialog box 18
8. Preferences dialog box 19
9. Right-click to launch editor or revisit **LOCATED** words 23
10. Debug window 27
11. The Forth virtual machine in a SwiftX target 42
12. A defining word and an instance, in host and target 59
13. XTL communication 66
14. SwiftOS round-robin multitasking loop 74
15. User variables apply an offset to a task area 79
16. Memory allotted for a background task 87
17. Two tasks are involved in a word containing **ACTIVATE** 91
18. Terminal task use of memory 119
19. Angles of repose 131
20. Conical pile calculator 132

List of Tables

1. Command window keyboard controls 15
2. File menu options 16
3. Edit menu options 17
4. View menu options 17
5. Options menu options 18
6. Examples of editor parameter sequences 19
7. Tools menu options 20
8. Project menu options 21
9. Help menu options 21
10. Number-conversion prefixes 40
11. Memory section types 42
12. Scope selectors 53
13. Added search-order commands for extending the cross-compiler 54
14. Scopes in which colon definitions are accessible 56
15. Object-file format selection 62
16. Memory access words affected by target connection 63
17. Host-to-target commands 67
18. Target-to-host responses 68
19. Vectored terminal-specific words 94
20. Functions applied to background and terminal tasks 98
21. Parameters governing the space allocation for a terminal task 118
22. Comparison of results using 16-bit and 32-bit arithmetic 136
23. Notation used for data types of stack arguments 139
24. Index to Forth words 140

Welcome!

What is SwiftX?

SwiftX is FORTH, Inc.'s interactive cross-compiler, the fastest and most powerful way of developing software for embedded microprocessors and microcontrollers. SwiftX is based on the Forth programming language, which for 25 years has been the language of choice for engineers developing software for challenging embedded and real-time control systems. SwiftX uses the power and convenience of Windows to provide you with the most intimate, interactive relationship possible with your target system, to speed the software development process, and to help ensure thoroughly tested, bug-free code. It also provides a fast, multitasking kernel and libraries to give you a big head start in developing your target application.

This book describes the basic principles and features of the SwiftX cross-compiler product line. It is accompanied by additional material documenting the specific CPU and platform you have purchased.

Scope of This Book

The purpose of this book is to help you learn SwiftX and use it effectively. It includes the basic principles of the cross-compiler, SwiftOS multitasking operating system, libraries, development tools, and recommended programming strategies.

This book does not attempt to teach Forth. If you are learning Forth for the first time, install this system and then turn to the *Forth Programmer's Handbook*, which accompanies this system.

Audience

This manual is intended for engineers developing software for processors in embedded systems. It assumes general knowledge of embedded system programming, and some familiarity with the Forth programming language (which you can get by following the suggestion above).

How to Proceed

If you are not familiar with Forth, start by reading the first two sections of the *Forth Programmer's Handbook*. Then experiment with this system by downloading simple definitions to your target and testing them before proceeding. Additional references about the Forth language are provided in Appendix A, "Bibliography."

After you have installed and tested SwiftX on the PC and on the target board supplied with this system, all SwiftX functions will be available to you.

Support

A new SwiftX purchase includes a Support Contract. While this contract is in effect, you may obtain technical assistance for this product from:

FORTH, Inc.
111 N. Sepulveda Boulevard, Suite 300
Manhattan Beach, California USA 90266-6847
310.372.8493 or 800.55.FORTH
forthhelp@forth.com

1. GETTING STARTED

This section provides a general overview of SwiftX, including information necessary to help you install the system and to become familiar with its principal features. Please refer to the platform-specific documentation that comes with your SwiftX system for details about the hardware connections.

1.1 COMPONENTS OF SWIFTX

SwiftX consists of the following components:

- Executable image of the SwiftX development system, including the Interactive Development Environment (IDE); cross-compiler; host support for the SwiftX Cross-Target Link (XTL, discussed in Section 4.9); and programming aids including debugger, disassembler, and other tools.
- Forth language source files for your specific SwiftX kernel.
- Pre-compiled image of the SwiftX kernel that will run on your target board. This kernel contains a representative set of functions that will support most embedded applications. Using the source files provided with this system, you may regenerate this image, modify it, or compile and test additional software to run on this kernel.
- A demo target board (for most targets), on which you may run the kernel provided with this system in order to test your software and learn about Forth and SwiftX.

The directory structure for the files provided with the SwiftX is shown in Figure 1. Source code is stored on disk in ASCII text files, and can be accessed and edited with a text editor of your choice.

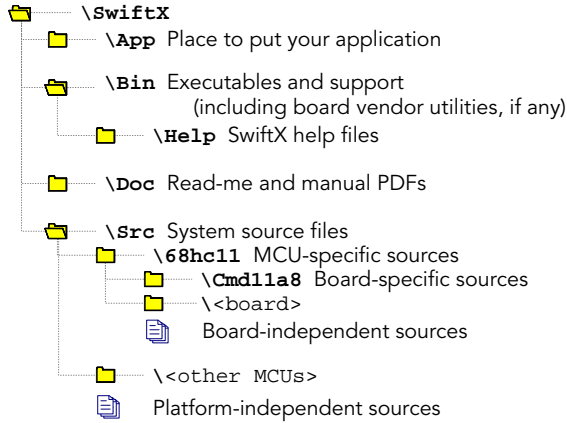


Figure 1. SwiftX directory structure

1.2 SWIFTX SYSTEM REQUIREMENTS

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

- PC running Windows 95/98, 2000 or NT. At least 16 Mb of RAM is recommended. The package will occupy approximately 13 Mb of disk space.
- A programmer's editor of your choice. Provision is made for linking interactive features of SwiftX with most standard editors.
- A serial or parallel printer port to connect to the target board; see your product-specific documentation for details.
- If you wish to burn PROMs, you will need suitable PROM-burning hardware and software.

1.3 INSTALLATION INSTRUCTIONS

This section describes how to install your SwiftX cross-compiler and connect it to the target board supplied with this system. It also describes basic procedures for downloading and testing a kernel and your programs.

1.3.1 Installing the Host Software

Follow this procedure to install your SwiftX software:

1. Turn on your computer and start Windows.
2. Start the install program by running **Setup.exe** on your installation disk. The install procedure may prompt you for additional information or choices.

An interpreter option (described in Section 7) is available in the SwiftX install package. If that option is selected during installation, the installer will ask for a password, which you may obtain from your SwiftX sales representative. The interpreter option adds several files to SwiftX, in both the **Src** and **Src\<CPU>** directories. It also provides a modified version of the **Kernel.f** file, which is the main load file for your SwiftX kernel. This includes the interpreter files at the appropriate points in the load sequence.

The default main directory is **SwiftX**. If you prefer another directory name, you will be given a chance to change it during the installation. However, this manual will use that name.

If you have previously installed SwiftX, you will be given the option to move older copies of files being installed to a backup directory. If you already have a SwiftX directory containing files that you prefer to retain, you may wish to rename it (or the new one).

The installation procedure creates a SwiftX program group on the Windows Start > Programs menu, from which you may launch the main program for your target. You may wish to add a shortcut to SwiftX in the directory **SwiftX\Src\<CPU>\<target-board>**. When SwiftX is launched, you may compile a target image of a SwiftX kernel plus libraries and other routines, depending upon how you have configured your system. There are two options for doing this, both available from SwiftX's project menu:

- `Project > Debug` compiles the target image, and establishes communication with the Cross-Target Link (XTL) on the target board. If no target is connected, this command will fail and will generate the message “No XTL. Try again? (Y/N)”. If the target system runs from PROM or Flash RAM, it will compare the newly compiled image to the image that is in the target; they must be identical for the XTL to work properly. With some targets, a new image can simply be downloaded. If these steps are completed successfully, additional debugging facilities are then downloaded to support interactive testing and debugging.
- `Project > Build` compiles the target image in a file on the PC suitable for burning into a PROM.

The normal development approach is to use Debug during development, and Build only when you are ready for PROM burning.

Other icons in the SwiftX program group provide access to documentation files.

To complete the installation, launch SwiftX and select other options as described in the sections below

1.3.2 Linking to a Text Editor

The SwiftX Interactive Development Environment 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 SwiftX to use an editor of your choice by specifying certain command parameters as described in Section 2.3.5.

1.4 DEVELOPMENT PROCEDURES

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

- Run the development configuration unchanged.
- Change the kernel in some way and debug the new kernel.
- Write and test application routines.
- Run the final debugged system.

Simple guidelines for doing these things are given in the following sections, and will prepare you to interact with your target board by using the software as delivered. Further details about the SwiftX compiler and interactive development aids are given in Sections 2.4 and 4.

1.4.1 Starting a Development Session



Launch SwiftX as described in Section 1.3.1. When SwiftX is running and your target board connected, you may change any options you wish and, when you are ready, you may compile your kernel by selecting Project > Debug from the menu or toolbar. This completely compiles the kernel and activates the Cross-Target Link (XTL).

If the board is not connected properly, you will get an error message, No XTL . Try again? (Y/N). This means the kernel was fully compiled, but the XTL connection was not established. Check your connections and select Project > Debug again.

If the download is successful, the target will greet you by displaying the system and target IDs. At this point, you may directly execute words on the target, examine its memory or I/O ports, and define additional words which will be downloaded automatically for testing. To try it, type:

```
2 6 + .
```

The numbers you typed will be transmitted to the target, followed by the commands + and . (the command “dot,” which types a number). These commands will be executed on the target board, which will add the numbers and display the sum. The sum will be displayed on your screen, because your PC is providing the target’s keyboard and display functions.

Whether or not you are connected to a target, you may use **LOCATE**, **DUMP**, and many other debugging commands, described in Section 2.4.

1.4.2 Using SwiftX With Other Hardware

If you are using a target system other than the test board supplied with this SwiftX product for your final application, you may need to make some changes to the parameters of your kernel. The configuration procedure described in Section 4.2 will help you to do this.



If your target will execute out of PROM, you will need to burn a PROM. The menu selection Project > Build compiles a target image and writes it in a file suitable for burning into a PROM. See Section 4.8 for the available object file formats.

1.4.3 Running the Demo Application

As shipped, the Debug option is configured to run a demo application based on the Conical Pile Calculator described in Appendix B.2. The source for this program is in the file **SwiftX\Src\Conical.f**; we recommend that you review it, because it exhibits many features found in SwiftX programming.

The demo program requires that the target be connected and communicating with the host. It may use either the host keyboard and screen via the XTL (the default configuration), or you may connect the target board's serial port to a COM port on your host and set up a separate task in the target to talk to a standard terminal emulator utility in the target, as described in Section 5.5.

To run this program, launch SwiftX as described in Section 1.4.1 and select Project > Debug to bring up the target program. Type **CALCULATE** to start the application; it will issue the prompt:

```
Enter material, feet, inches: nM nF nI =
```

followed by a list of materials.

Select a material by typing, for example, **1M**. The target will echo, in that example, `Loose gravel`. Similarly, enter a number of feet and inches. Finally, type **=** and the target will display the answer.

This demo program is configured to run indefinitely; to exit, press Esc on the

host and reset the target by typing **RESET**.

To explore further, try typing **LOCATE DENSITY**. This will display the source where this word is defined. If you have linked an editor (as described in Section 1.3.2 and Section 2.3.5) you may press your right mouse button and select Edit This to launch your editor, opening the **Conical.f** file.

You may examine the contents of any of the variables in this program by typing, for example:

```
DENSITY ?
```

You may also exercise the individual words in the demo by providing suitable stack arguments (look at the comment following the name of each definition); for example:

```
1 SELECT
```

This displays `Loose gravel` and stores the parameters for that material in **DENSITY**, **THETA**, and **MATTER**.

1.4.4 Changing the SwiftX Kernel

When you have successfully compiled the kernel and exercised the Cross-Target Link, you can begin trying changes to the kernel.

The simplest (and safest) changes involve adding new words. You can simply type new definitions at the keyboard, and they will be cross-compiled and downloaded automatically, available for immediate testing. For example, you could type:

```
: Hi      ." HELLO, WORLD! " CR ;
```

This defines a word which, when executed, displays the message between quotation marks, followed by a new-line function (**CR**). Try it by typing **Hi**.



Edit

You can create a new file—e.g., via the File > Edit menu item or toolbar button or by typing **EDIT filename**—for your source code, for example, **Extras.f**. With the board in interactive mode (after doing a Project > Debug), you may compile the words in your new file for interactive testing by using the menu



item or toolbar button File > Include or by typing:

INCLUDE EXTRAS

For the typed command, the **.f** extension is assumed.

You may do this repeatedly, until the board runs out of code space. If necessary, you may start over with a new download by repeating Project > Debug, which re-initializes the target.

When the new functions are tested to your satisfaction, you may add them to the kernel by inserting **INCLUDE EXTRAS** before the last lines of the main load file **Kernel.f** and before the startup code.

The startup code for a SwiftX system is usually factored into two files, both called **Start.f**, but at different directory levels: one is in the directory **Swiftx\Src\<CPU>**, and one is in the target-specific subdirectory. These must always be the last files to be loaded, with the target-specific one (containing the actual power-up code) last.

Then rebuild and re-install the kernel as described above (using Project > Build), and test. Instructions for installing a new kernel are provided in your platform-specific documentation.



Changes to the Forth primitives (such as those in the file **Core.f**) should only be made with the greatest of care. The SwiftX kernel and the Cross-Target Link rely on these primitives for their operation.



Changes to the model of the underlying Forth system (such as cell size or stack allocation) should only be attempted by an experienced Forth programmer.

2. INTRODUCTION TO SWIFTX

SwiftX is a development system used to create software for microcontrollers in various types of embedded systems.

To debug embedded software, interaction with the target hardware is needed. In a conventional C or assembler programming environment, this usually is achieved by using in-circuit emulators, simulators, and debuggers. These tend to be expensive and complex, and provide only limited interactivity, at best.

SwiftX supports interactive development and testing of software, on even the smallest microcontrollers, without expensive additional hardware and software tools. This is achieved by using a Windows-based host computer to handle a continuous communications link between the PC and the target.

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

2.1 SWIFTX PROGRAMMING

The SwiftX cross-compiler is based on the Forth language, and its target source code is written in Forth. Therefore, gaining familiarity with the essentials of Forth is strongly recommended before tackling SwiftX programming.

If you are a Forth beginner, read the *Forth Programmer's Handbook*. Review the demo application ("Conical Pile Calculator") supplied with your system. Find out what software is available by looking through the source code supplied with SwiftX. Finally, don't hesitate to contact the FORTH, Inc. Hotline Support Service with any questions or problems (see page xii). Forth programming courses are available at FORTH, Inc. and can help shorten the learning process.

SwiftX is a very powerful and flexible system, supporting software development for virtually any embedded system configuration. Although the internal principles of SwiftX 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

The SwiftX system has two basic components: a *host* system (the PC side of the development system) which supports the SwiftX cross-compiler and interactive programming tools, and a *target* system which executes on the target board and supports interactive debugging with the host.

The host software is based on a 32-bit Forth system running under Windows. This host is designed to support a special cross-compiler for the target processor, along with a cross-assembler and the source code for the target system's kernel and application.

The separation between host and target is manifested in several areas:

- **Memory and address space** There are a number of different address spaces: the host's local memory, which is not directly accessible in the compiler; the various regions of target memory (code space plus initialized and uninitialized data space); and a private region where the host maintains pointers into the target's memory image. When you are programming in SwiftX, you usually are working with target data space.
- **Command set** SwiftX compilers are written in Forth. The host provides locally executable versions of many Forth words that are also present in the target. In order to distinguish between the host and target versions of these words, they are maintained in separate, searchable *word lists*. Commands are provided to select among these word lists. The context in which a word may be accessed (either for execution or for compiling a reference to it) is called its *scope*. The default arrangement is that one accesses the target versions of these words. Scopes are discussed further in Section 4.6.
- **Compiler words and directives** There actually are two compilers in SwiftX. The most visible compiler is the one used to construct the target program, and

contains defining words (such as **:**, **CONSTANT**, **VARIABLE**, etc.), flow-of-control words (such as **IF**, **THEN**, **BEGIN**, **DO**, **LOOP**, etc.), words for managing data space (such as **,** and **ALLOT**, etc.), and other compiler words. However, an underlying compiler on the host was used to build the cross-compiler. This compiler may be extended in order to provide special compiling or defining capabilities. For this reason, it is accessible via special *scope selectors* (described in Section 4.6.2), as are the host versions of common Forth commands.

SwiftX contains an *assembler* which provides direct access to the native instruction set of the CPU. The assembler for your target processor is described in a separate document accompanying your SwiftX system.

SwiftX operates in one of two states: *interpreting* or *compiling*. The programmer may add new words that act in either of these states. New words added for the interpreting state might, for example, support custom target data structures; new words added to the compiling state might support custom conditional or loop structures in target definitions.

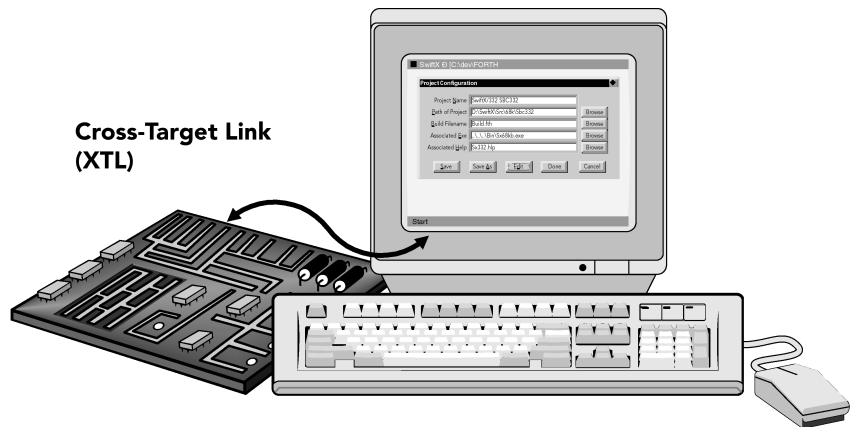


Figure 2. Interaction between host and target components

SwiftX also supports a third state, called *interacting*. This means that words typed on the command line will be executed on the target. The Cross-Target Link (XTL, described in Section 4.9) connects the host and target systems in this interactive mode. Two programs communicate via this link. The *host* XTL issues commands to the target and responds to output from the target. The *target* XTL receives and executes commands from the host, and sends display

commands to the host. The XTL allows communication at moderate speed over a serial or parallel line.

The target software may be thought of as having several layers, illustrated in Figure 3. Included with your SwiftX system are the core, drivers (including the XTL protocol on systems using a serial host-target link), SwiftOS multitasking executive (see Section 5), and libraries (supplied in source form). You may select which libraries to add to the system (see Section 6). You may also add custom drivers, other libraries, and application code easily. This makes your SwiftX system very adaptable, in both content and size, to your special requirements.

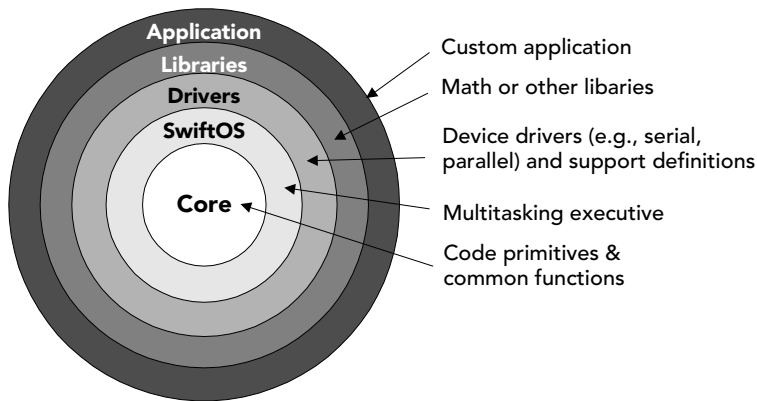


Figure 3. Code layers in a SwiftX system

2.3 IDE QUICK TOUR

The SwiftX Interactive Development Environment (IDE) presents a user interface that may be managed from a command line or with pull-down menus. This section summarizes its principal features.

2.3.1 The Command Window

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

which will be executed by SwiftX or routed to the target for execution, as described in Section 2.5.

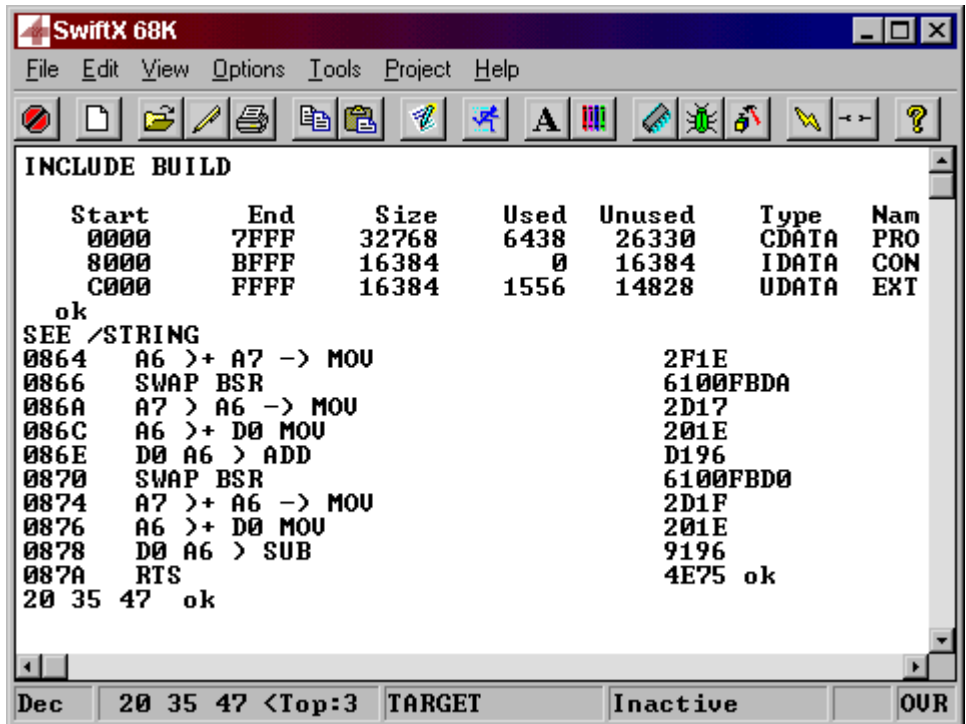


Figure 4. The SwiftX command window

All information displayed in the command window (including commands you type, and system responses or displays) is kept in a large circular buffer while the IDE is running; to see previous parts of the session, you may scroll through this buffer by using the scroll bar or the PageUp and PageDown keys. You may also print or save the entire buffer, or any portion of it 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 5).

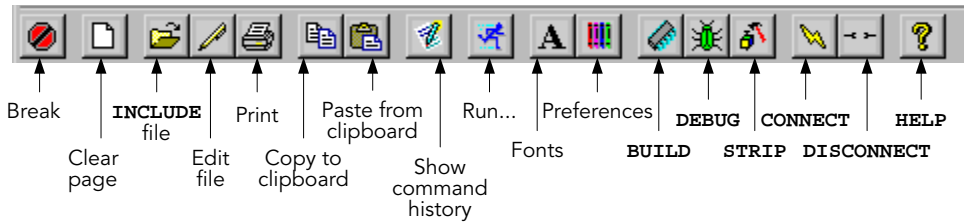


Figure 5. Toolbar items

The status bar at the bottom of the command window shows the current number base (the default is decimal), the stack depth with the actual values of the top several items, the current scope (see Section 4.6.2), and other useful information (see Figure 6).

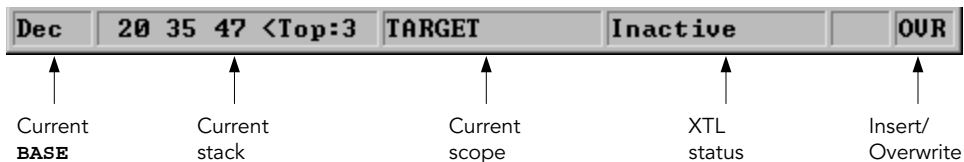


Figure 6. Status bar indicators

In addition to the buffer containing the general history of the command window's contents, SwiftX 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 (indicated at the right end of the status bar). You can execute the entire line by pressing Enter; or leave the line without executing it, by pressing Esc.

You may display the source for any word defined in your current scope by double-clicking the word. (Also see Section 2.4.1 for details about displaying and interacting with source code.) If the source is not present, you will get an error message. For example, double-clicking on a word defined in the cross-compiler or assembler returns the error alert "Can't open file"; attempting to locate the source of a word defined interactively in the command window returns the message "Can't locate keyboard definition". This feature is discussed further in Section 2.4.1.

After you have displayed source for a word in the command window, pressing Alt-PageUp and Alt-PageDown will display adjacent portions of that source file.

Table 1: Command window keyboard controls

Key	Action
PageUp PageDown	Scroll through the history of the current session.
↑ ↓	Retrieve commands you have typed.
← →	Move cursor on command line.
Insert	Toggle the insert/overwrite mode of typing.
Enter	Execute the command line the cursor is on.
Ctrl-C Ctrl-V	Copy from, or paste text into, the window. (See Section 2.3.3.)
Double-click on a word	Display source code for the word (if available). Equivalent to LOCATE or L .
Alt-PageUp Alt-PageDown	Scroll through the source file containing the word whose source is currently displayed.
Right-click	Display pop-up menu options: PageUp, PageDown, Edit This (which launches your editor with the cursor positioned at the source line targeted by your previous LOCATE action; see Sections 2.3.5 and 2.4.1); and with shortcuts to previously LOCATED words. (See Figure 9 on page 23.)

The following sections describe the menu options that are available from the command window. 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 command (if any), the Alt version, and a description of the item.

2.3.2 File Menu

The File menu offers the selections described in Table 2.

Table 2: File menu options

Item	Command	Action
<u>I</u> nclude	INCLUDE <filename>	Interpret a file (load it and any files it loads). Displays a file-selection dialog box; INCLUDE processes the file <i>filename</i> .
<u>E</u> dit		Launch a linked editor, allowing you to select a source file.
<u>P</u> rint		Print the command window. In the print dialog, you may choose to print the entire contents or a selected portion.
<u>P</u> rint Setup		Select and configure a printer.
Save <u>C</u> ommand Window		Record the current contents of the command window in a text file.
Save Keyboard <u>H</u> istory		Record all the commands you’ve typed in this session in a text file.
Session <u>L</u> og		Start recording all actions for this session in a text file.
<u>B</u> reak		Force the main console task to abort; used for error recovery.
<u>E</u> xit	bye	Exit SwiftX.



Edit

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



Include

Both the File > Include and File > Edit menu items and their corresponding toolbar buttons bring up a “Browse” dialog box through which you can find your file. Both will reset SwiftX’s path to the one for the file you select. However, the command **INCLUDE** invoked from the keyboard will *not* change SwiftX’s current-path information.

Save Command Window, Save Keyboard History, and Session Log are discussed further in Section 2.4.6.

2.3.3 Edit Menu

Most editing in SwiftX is done with your associated editor (see Section 1.3.2 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



Copy



Paste

Item	Keystroke	Action
<u>C</u> opy	Ctrl-C	Copy selected text to your clipboard.
<u>P</u> aste	Ctrl-V	Paste the current clipboard contents on a new command line and interpret its contents.
<u>W</u> ipe 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. Each feature will toggle when you select it. The choices are described in Table 4.

Table 4: View menu options

Item	Action
<u>S</u> tatus line	If checked, displays the status line at the bottom of the screen.
<u>T</u> oolbar	If checked, displays the toolbar at the top of the screen.

2.3.5 Options Menu

The Options menu provides ways to customize SwiftX. Its selections are summarized in Table 5.

Table 5: Options menu options

Item	Action
<u>F</u> ont	Select a font for the command window. Only fixed-width (i.e., non-proportional) fonts are listed.
<u>E</u> ditor	Select and set parameters for your editor.
<u>P</u> references	Set text and background colors for normal and highlighted displays, select case sensitivity, and other options
<u>S</u> ave Options	Save all current settings.

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

Next you must specify, on the Editor Options line, how SwiftX is to pass line-number and filename parameters to your editor. The specification format is:

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

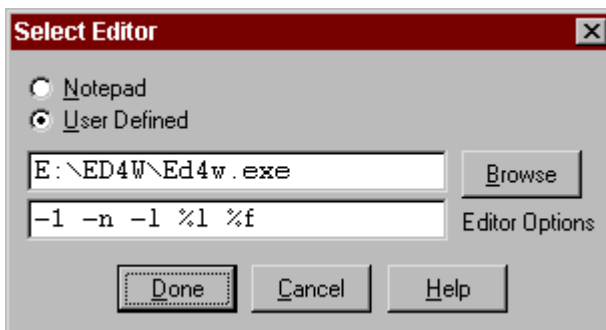


Figure 7. Select Editor dialog box

When SwiftX calls the editor, it provides the line number at the place in this string that has a %l (lower-case L), and provides the filename at the place that has a %f. Example parameter strings for some editors are shown in Table 6.

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" (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"
UltraEdit	%f/%l"
WinEdit	"%f" /#:%l

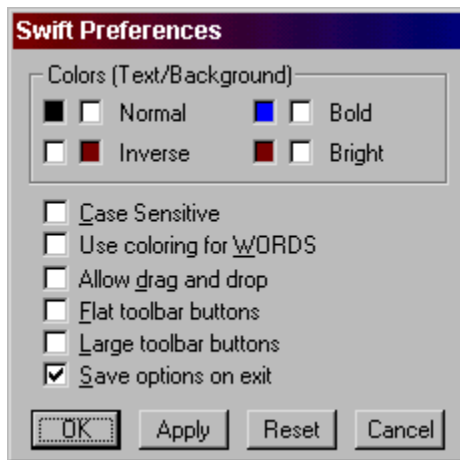
When your editor's pathname and parameter string are correct, click Done. This information will be saved when you exit SwiftX.



Prefs

SwiftForth's Options > Preferences dialog (or its equivalent Toolbar button) lets you specify a number of configuration items, shown in Figure 8.

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

**Figure 8. Preferences dialog box**

"Use coloring for WORDS" vectors the command **WORDS** (described in Section 2.3.6). If this box is checked, typing **WORDS** in the command window produces a color-coded display in the command window; otherwise, it launches the Words browser window. The behavior of the Words browser when launched from the toolbar is not affected by this option.

"Allow drag and drop" lets you drag files (e.g., from a directory window or Windows Explorer) onto SwiftForth, which will cause the dropped file to be **INCLUDED**.

“Save options on exit” means your selections will be recorded so 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.




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

References **WORDS** command, Section 2.3.6
 INCLUDE, Section 3.1

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
Run	<u>W</u> ords	WORDS	Display the words available in the current scope.
	<u>R</u> un	RUN	Run an auxiliary program (e.g., a PROM programmer).
History	<u>H</u> istory		Open command history window, and start recording
	Connect	CONNECT TARGET	Start the XTL communicating with the target.
Connect	Disconnect	DISCONNECT TARGET	Disable the XTL connection.



Disconnect

The Connect and Disconnect commands let you temporarily change the status of your communication with the target. This is sometimes useful for comparing host and target initialized data spaces, for example.

References XTL communication, Section 2.5, Section 4.9

2.3.7 Project Menu

A *project* is a link between the SwiftX cross-compiler for the kernel's CPU and a source file set up to build your system. It also incorporates configuration information, such as the port and baud rate used for the XTL.

Table 8: Project menu options



Debug



Build



Strip

Item	Command	Action
<u>D</u> ebug	DEBUG	Compile the target system and activate the XTL for interactive debugging.
<u>B</u> uild	BUILD	Compile the target system, leaving its image in a file.
<u>S</u> trip	STRIP	Recompile the target (as for Build), omitting any words that are never called.

Section 1.4 discusses the uses of Debug and Build; Debug is the option most heavily used during development.

2.3.8 Help Menu

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

Table 9: Help menu options



Help

Item	Content
<u>F</u> orth	A glossary of common Forth words.
<u>S</u> wiftX	Information relating to the use of SwiftX.
<u>M</u> CU	Target-specific information, including assembler documentation, configuration issues, etc.
<u>H</u> andbook	<i>Forth Programmer's Handbook</i> , in PDF format.
<u>R</u> eference Manual	This manual, in PDF format.
<u>T</u> arget Manual	Documentation for the target CPU and boards.
G <u>O</u> <u>O</u> ne	Connect to the FORTH, Inc. web site.
<u>A</u> bout	Product release date and related information.

2.4 INTERACTIVE PROGRAMMING AIDS

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

2.4.1 Interacting With Program Source

The command:

```
LOCATE <name>
```

is equivalent to double-clicking on a word in the command window (discussed in Section 2.3.1). If *name* is defined in the current scope, this will display several lines of the source file from which *name* was compiled, with *name* highlighted. This will work for all code compiled from source files; source is not available for:

- code typed directly into the SwiftX command window
- source code that was pasted into the command window
- words in the SwiftX cross-compiler and assembler

LOCATE may also fail if the source file has been altered after the last time it was compiled, because the host version of each compiled definition contains a pointer to the position in the file that produced it.

For example, the command:

```
LOCATE /STRING
```

(or double-clicking on **/STRING**) opens the correct source file, and displays:

```
Line   23 D:\SWIFTX\SRC\STRING.F
: MOVE ( c-addr1 c-addr2 u)
    >R 2DUP SWAP DUP R@ + WITHIN IF
    R> CMOVE> EXIT THEN R> CMOVE ;

: ERASE ( c-addr u -- ) 0 FILL ;
: BLANK ( c-addr u -- ) BL FILL ;
```

```

: /STRING ( c-addr1 u1 u -- c-addr2 u2)
>R SWAP R@ + SWAP R> - ;

: SEARCH ( c-addr1 u1 c-addr2 u2 -- c-addr3 u3 flag )
2>R 2DUP BEGIN DUP R@ U< NOT WHILE
    OVER R@ 2R@ COMPARE WHILE 1 /STRING REPEAT
2SWAP 2DROP 2R> 2DROP -1 EXIT THEN
2DROP 2R> 2DROP 0 ;

```

SwiftX keeps a history of the last few words you displayed in this fashion. If you right-click the mouse, you will see this list in a pop-up menu, shown in Figure 9. You can re-display any of them by selecting its name from this menu.

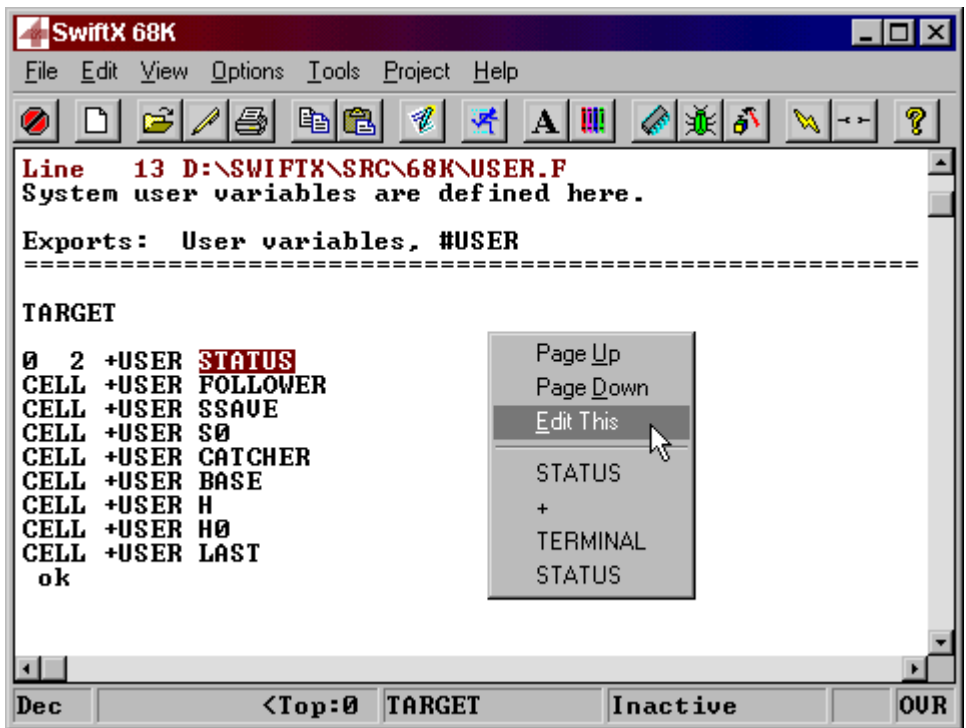


Figure 9. Right-click to launch editor or revisit LOCATED words

Near the top of this menu, you will see the option Edit This. Selecting that

command, or typing **EDIT**, 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 you last located. This feature lets you immediately edit the source, if you wish, and examine the rest of the file. (Using the Options menu to link to an editor other than the default Notepad is described in Section 2.3.5.)

You may also use the **EDIT** command to open an arbitrary file for editing, by typing **EDIT** *wordname* or **EDIT** *filename*. (Also see the File > Edit menu option, discussed in Section 2.3.2). If **EDIT** sees a string, it will attempt to look it up in the dictionary, in the current scope. If it finds it, it will use the editor to open the source file in which *wordname* is defined, positioned at the definition. If the text cannot be found in the dictionary, **EDIT** will attempt to treat it as a filename and open that file in the editor. To avoid confusion, we recommend that you always append the **.f** extension when you use **EDIT** with a filename.

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, SwiftX 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 SwiftX command window. *name* must be in the current scope.
- L** (—)
 After a compiler error, display the source line at which the error occurred, along with the source path and line number, in the SwiftX command window.
- EDIT** <text> (—)
 Launch or switch to a linked editor, passing appropriate commands to open and position a source file. The *text* following **EDIT** is optional. If it is present, **EDIT** will attempt to find it as a defined word and open the file in which it is defined, positioned at its definition. If *text* cannot be found in the dictionary, **EDIT** will attempt to treat it as a filename and open the file. If no text is provided, **EDIT** will select the source most recently displayed by **LOCATE**, **L**, or a double-click.

2.4.2 Cross-references

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

WHERE <name>

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

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

WH <name>

does the same thing.

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

Glossary

WH <name> (—)

WHERE <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 synonymous.

2.4.3 Disassembler

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

The single command **SEE** *name* disassembles **CODE** commands and colon definitions defined for the target scope. The results depend somewhat upon the implementation; see your target-specific documentation. Subroutine-threaded systems and systems that compile actual machine code may be unable to

reconstruct a high-level definition, and may instead show the assembler code that was generated. In such a case, use **LOCATE** to display the source.

An alternative is to disassemble or decompile from a specific address. This is useful for decompiling headless code, such as code preceded only by a **LABEL**. The command to disassemble a **CODE** definition, given an address *addr*, is:

```
<addr> DASM
```

Glossary

SEE <name> (—)

Disassemble or decompile *name*, where *name* must be a target definition. The display format will depend on the CPU and the implementation strategy (see your target-specific documentation).

DASM (*addr* —)

Disassemble code beginning at *addr*. The display format is platform dependent.

References Using **LOCATE** to display source, Section 2.4.1
CODE and **LABEL**, Section 4.4

2.4.4 Memory Dumps

Sequences from the host image or target may be dumped via these commands. If you are connected with an active XTL, all sections of actual target memory will be displayed; if not, you will see the host image, and uData may not be dumped.

Glossary

DUMP (*addr u* —)

Display *u* bytes of hex characters, starting at *addr*, in the current section, which may be either code or data. If you are not connected to your target, you can only display code space or initialized data space.

DUMPC (*addr u* —)

Display *u* bytes of hex characters, starting at *addr*, in the current code-space section.

References **CONNECT** and **DISCONNECT**, Section 4.9.1
CODE and **LABEL**, Section 4.4
Memory sections, Section 4.3

2.4.5 Single-Step Debugger

SwiftForth's single-step debugger allows you to step through source compiled from a file. A simple example is the sample program **sstest.f**, shown below:

```
REQUIRES SINGLESTEP
[DEBUG
: 2X ( n -- n*2 )    DUP + ;
: 3X ( n -- n*3 )    DUP 2X + ;
: 4X ( n -- n*4 )    DUP 2X SWAP 2X + ;
: 5X ( n -- n*5 )    DUP 3X SWAP 2X + ;
DEBUG]
```

Assuming this source has been compiled from the file **sstest.f**, you may type **4 DEBUG 5X** to cause a debug window to appear, as shown in Figure 10.

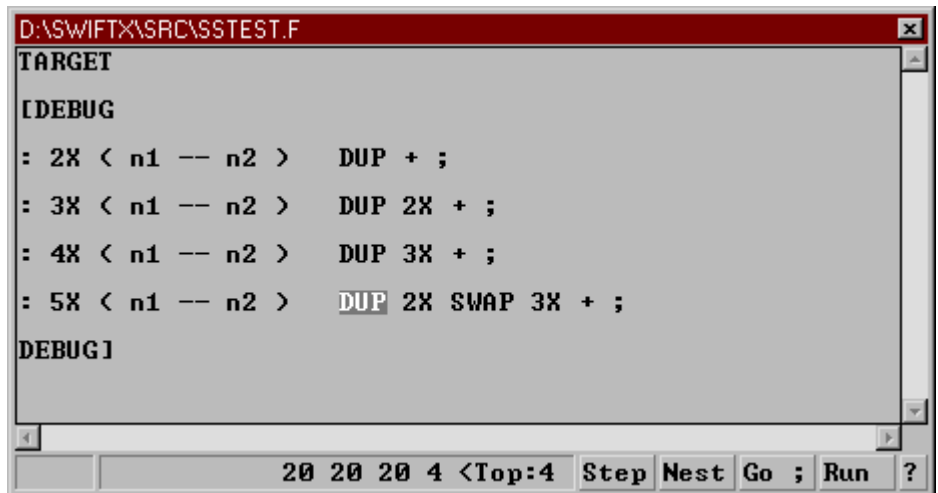


Figure 10. Debug window

The current stack is displayed at the bottom of the debug window. While this debug window is active, you may control it using the buttons at the bottom, which work as follows:

- Step* Execute the next word, with no nesting.
- Nest* Execute the next word, nesting if it is a call.
- Go ;* Execute to the end of the current definition without stopping.
- Run* Stop debugging and continue running at (near) full speed.

2.4.6 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.



History

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.

2.5 HOST-TARGET INTERACTION

SwiftX supports interactive development via a debug interface known as the XTL (Cross-Target Link). This provides access to the target through the terminal's on-board serial port. XTL functions include reading and writing memory, as well as controlling execution.

The XTL interface supports the following debug functions:

- Display and modify target memory
- Download data to target memory
- Interactively test target words
- Examine the data stack using the `.s` command



Most of the programming aids discussed in this book that facilitate interactive testing and debugging of the target kernel use the XTL. The XTL is discussed in more detail in Section 4.9.

3. PROGRAMMING WITH SWIFTX

3.1 SOURCE-FILE MANAGEMENT

The primary vehicle for SwiftX 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.

3.1.1 Interpreting Source Files



Include

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 and menu options display a browse window in which you can select the file, and they change SwiftX's current path *to that of the selected file*, whereas the typed command handles a specified file (including relative path information) and *doesn't affect the 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.

2. **Relative path to subdirectories** The name does not begin with a `\` or `..\` and the file is located in the current directory or in 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 directory level from the current subdirectory. After you have raised the level sufficiently, you can use the technique in #2 to go down subdirectory levels.

The **CD** (Change Directory) command works 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.

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

```
\Swiftx\Src\<<CPU>\<Platform>\Kernel.f
```

To see a log of the names and target addresses of all definitions, you may set:

```
LOGGING ON
```

This causes any **INCLUDE** operation to display in the command window a log showing the locations of definitions compiled, which may be printed or saved using the File menu items described in Section 2.3.2.

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. Leaves **BASE** set to decimal. **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 the current path.

References File-based disk access, *Forth Programmer's Handbook*, Section 3.4

3.1.2 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 accommodate this, SwiftX defines braces as functionally equivalent to parentheses except for taking a terminating brace instead of the right parenthesis. 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, SwiftX 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

<code>{</code>	(—)
Begin a comment, which may extend over multiple lines, until a terminating right brace <code>}</code> is encountered.	
<code>\\</code>	(—)
During an INCLUDE operation, treat anything following this word as a comment; i.e., anything that follows <code>\\</code> in a source file will not be compiled.	

3.1.3 File Load Order

The main load file for your system is **Kernel.f**. Its contents are organized into groups identified by a comment at the beginning of each:

- **Nucleus** contains functions that, collectively, represent most of the run-time words in the ANS Forth Core wordset (the compiler and interpreter words are available on the host). Most are required, but the two files called **Double.f** (at different directory levels) may be omitted if you do not need 64-bit and mixed 32-bit/64-bit arithmetic. The last file in this group, **Methods.f**, is also optional; it supports the ANS Forth word **VALUE**, which you may or may not use.
- **Extensions** contains the multitasker and also **Tools.f** (containing **?**, **.S**, and **DUMP**, which are extremely useful in debugging but are rarely used in applications). The multitasker is quite small, and we recommend keeping it even if you have only one task, because the interaction it provides between interrupts and task-level processing is very convenient.
- **Drivers** supports hardware specific to your SwiftX system. These files are required for the system as shipped, but you may need to modify or replace them if your ultimate target hardware is different from the demo target board.
- **Electives** contains functions specific to your application. This is the recommended place to load SwiftX library files, as well.
- **Initialization** contains the startup functions, and must be last.

Files containing new code that is being tested can be loaded interactively using `File > Include` (described in Section 1.4.4) and the procedures described in Section 3.2.1. When you are confident that it works, you may add it to the Electives group in the **Kernel.f** load file; however, if you are working on a substantial application, we recommend that you develop a separate load file to control your application files, and **INCLUDE** that file in the Electives group. Near the end of **Kernel.f**, a file **App.f** is loaded. This is intended to serve as the main load file for your application.

3.2 PROGRAM DEVELOPMENT STRATEGY

SwiftX is shipped with a pre-compiled kernel configured to contain a representative set of functions, and a simple demo application (described in Section B.2).

This section will describe the overall procedure for developing and testing your application.

Application development with SwiftX generally follows a pattern of incremental compiling and testing of increasingly complex application functions. Providing your target allows you to download new code into its memory, this is extremely easy.

The overall process is as follows:

1. Write a related set of functions.
2. Test them interactively.
3. Add them to your kernel.
4. Repeat steps 1–3 as necessary until you’re finished.
5. Move your highest-level functions and start-up code into the kernel, ready to run from PROM as a standalone program.

3.2.1 Interactive Development

The interactive development phase should occupy most of your time. It depends on using a modified version of the file **Debug.f** supplied with your system, and requires writable code space (preferably RAM, but flash or EEPROM can be used with some inconvenience) for the code under test.

The basic procedure is to replace the demo files loaded between the commands **SET-DOWNLOAD** and **DOWNLOAD-ALL** in **Debug.f** with the files being tested, as described in Section 4.9.1. You may list the files individually or, especially as your application becomes more complex, use an **INCLUDE** file such as the file **App.f**, as described in Section 3.1.3. Since SwiftX is temporarily disconnected from the target between **SET-DOWNLOAD** and **DOWNLOAD-ALL**, however, you cannot do such things as initializing uData or interrupt vectors at this time.

In general, we strongly recommend that you centralize all target initialization functions, because it will be easier to manage them during development and when bringing up your standalone system when you’re finished. Depending on the extent of required initialization, you can group such activities in a definition or a few definitions in a file. During interactive development, you can do these things when the target is re-connected following the **DOWNLOAD-ALL**.

If your application is large or your RAM code space is limited, you may wish to incrementally test related groups of functions and, after they are tested, add them to the kernel by moving their **INCLUDEs** to **App.f**.

3.2.2 Building an Application for ROM

When all your application has been tested interactively, it is time to configure your final application. This generally is done by moving the loading of your application files to the file **App.f** (which is loaded by the main file **Kernel.f**). If your application is very complex, you may wish to group major components into their own load files, which are then loaded by **App.f**.

Two major steps remain before your application is completely finished:

1. Remove un-needed code from your kernel.
2. Perform all application-related hardware and software initialization functions.

The first may be done in stages. If memory is in short supply even during development, look hard at the list of functions loaded in **Kernel.f** and discard those you know you won't need (e.g., fraction arithmetic). As your familiarity with both SwiftX and your application grows, you may be able to discard more.



When your application is complete and installed in the **App.f** file loaded by **Kernel.f**, you can use the ultimate weapon: the stripper. This feature (Project > Strip) repeatedly does a Build, noting each time which (if any) words were never called. On the next pass, the compiler will skip those words. When words have been omitted, any additional words *they* called may be freed on the next pass. When two successive compiles give the same target size, the process has completed—your application has everything it needs and nothing more. (The source files are not affected.)

We don't recommend using Strip until your application is essentially complete and tested, as you may strip from your kernel functions you will need later.

Power-up initialization is performed in two files named **Start.f**. One of these is in the directory for your MCU, and the other is in the directory for your board. If you have custom hardware, you may have to modify the board-level version.

Basic hardware initialization is performed by the SwiftX kernel in the **POWER-**

UP routine in the board-level **Start.f**, and handles these general requirements:

1. Low-level hardware initialization (e.g., disable interrupts until interrupt vectors are set, configure memory, etc.).
2. Forth virtual machine initialization, consisting of setting the data and return stack pointer registers and the multitasker user pointer.
3. Set up interrupt vectors and enable interrupts. If you have specified interrupts in your drivers, they will already have been installed in a table copied to the “live” interrupt vectors by this code; see your MCU-specific manual for details.
4. Jump to the high-level startup routine.

The address of **POWER-UP** is placed in the device’s power-up vector location.

Most initialization is performed by the high-level startup routine named **START**, usually found at the end of the board-level **Start.f** file. Here’s a representative example (from the 68HC12 CMD12A4 board):

```
| : START ( -- )                \ High-level initialization
  OPERATOR CELL+ STATUS DUP |U| ERASE    \ Set up OPERATOR task
  |OPERATOR| CMOVE
  /IDATA /RTI /CLOCK /SCI0 /SCI1        \ iData, interrupts, etc.
  GO ;                                \ Application initialization
```

The default contents of the file **App.f** (which is intended to be used to load your application) is a single definition for **GO**, as follows:

```
: GO ( -- )    DEBUG-LOOP ;
```

This simply starts the XTL running. In order to launch your application from power-up (rather than from Debug) you need to replace this with a definition that performs any additional initialization required by the application and that finally calls the highest-level word in the application that makes it all run.

4. CROSS-COMPILER PRINCIPLES

This section covers the requirements of the cross-compiler used to develop the target program, including methods you may use in source files that will generate target code—such as techniques for mapping memory, controlling the compiler in various ways, and accessing the code and initialized data space images in the host and target.

The cross-compiler uses the same words as a resident Forth system to construct definitions, and to define and manage data objects. The discussion of these words in this section focuses on special issues related to cross-compilation; please refer to the *Forth Programmer's Handbook* for basic descriptions and examples of usage.

4.1 CROSS-COMPILER CONTROL

This section describes how you can control the SwiftX cross-compiler using typed commands and (more commonly) program source. In most respects, the words typed at the command-line in the SwiftX command window is treated identically to program source—anything you do in source may be done interactively.

4.1.1 Input Number Conversions

When the SwiftX text interpreter encounters numbers in the input stream, it converts them to binary. If the system is in compile mode (i.e., between a `:` and a `;`), it will compile a reference to the number as a literal and, when the word being compiled is executed later, 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 are controlled by the user variable **BASE**. The host system's **BASE** controls all input number conversions on the host; there is also a **BASE** in the target that controls number conversions performed by the target system. The words described in this section are used to control both the host and target versions of **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; see Section 6.1.

In addition, input number conversion may be directed to convert a single number using the base specified by a prefix character from Table 10. 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 10: Number-conversion prefixes

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

References Enhanced number conversion, Section 6.1

4.1.2 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 (see **Config.f**):

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

then in **Kernel.f** 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 a word has not been defined. For example, in **Strings.f** we find:

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

[THEN]
```

[UNDEFINED] <word> will return a *true* flag 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.

References Conditionals, *Forth Programmer's Handbook*, Section 2.5.3

4.2 TARGET SYSTEM CONFIGURATION

System configuration occurs in the file

```
Swiftx\Src\<CPU>\<Target>\Config.f
```

which specifies the memory organization of the target system. Platform-specific configuration issues are discussed in your target-specific documentation.

SwiftX divides target memory into two logical regions, as shown in Figure 11: code space (which may or may not be in ROM) and data space (directly accessible RAM). Data space is further divided into *initialized* and *uninitialized* regions. Initialized data space may be pre-set to specific values at compile time; these values will be automatically copied into initialized RAM at power-

up in the target. You may further define multiple sub-regions of each type, as discussed in Section 4.3.

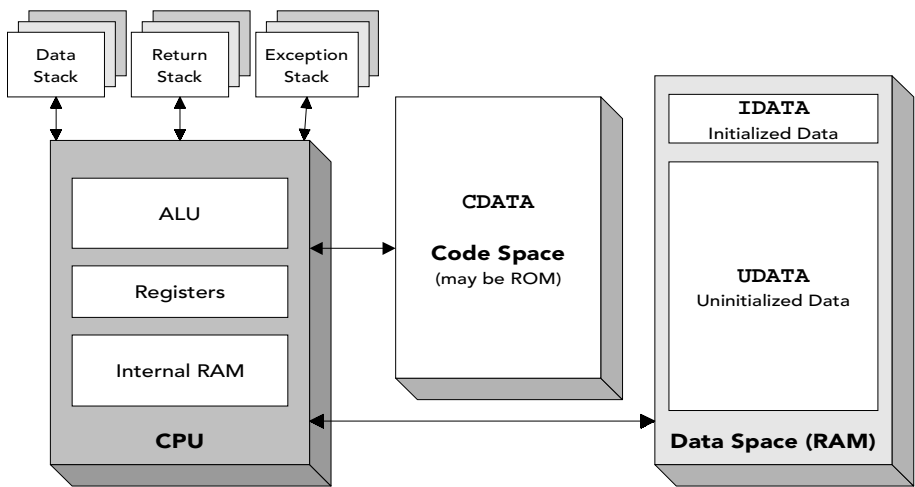


Figure 11. The Forth virtual machine in a SwiftX target

4.3 MEMORY ALLOCATION

Target memory space can be divided into multiple *sections* of three types, shown in Table 11. Managing these spaces separately provides an extra measure of flexibility and control, even when the target processor does not distinguish code space from data space.

Table 11: Memory section types

Type	Description
CDATA	Code space; includes all code plus initialization tables. May be in PROM.
IDATA	Initialized data space; contains preset values specified at compile time and instantiated in the target as part of power-up initialization.
UDATA	Uninitialized data space, allocated at compile time. Its contents are unspecified.

The words in Table 11 select a *current section type*. The current section type controls data defining words, memory allocation, and memory accesses of various types, both during compilation and during interactive testing.

At least one *instance* of each section must be defined, with upper and lower address boundaries, before it is used. Address ranges for instances of the same section type may not overlap. The syntax for defining a memory section is:

```
<low address> <high address> <type> SECTION <name>
```

An instance becomes the *current section* of its type when its name is invoked. The compiler will work with that section as long as it is current, maintaining a set of allocation pointers for each section of each type. Only one section of each type is current at any time.

The words used to allocate and access memory (the *vectored words* in the glossaries below) are vectored to operate on the current section of the current type. Some words control only cData, and are available any time; their use does not affect the selection of the current section or type. Use of one of the section type selectors **CDATA**, **IDATA**, or **UDATA**, sets the vectors for the vectored words. If you only have one section of each type, the section names are rarely used.

As an example, consider the 68HC12/CMD-A4, which runs from PROM. Its **Config.f** file defines the following sections:

```
INTERPRETER  HEX
0800 08FF IDATA SECTION IRAM          \ Initialized data
0900 0BFF UDATA SECTION URAM          \ Uninitialized data
8000 FFFF CDATA SECTION PROGRAM       \ Program in external ROM
```

Note that the **PROGRAM** section is in ROM. This is necessary for the system, and will probably be fine when the application is finished, but it will make it hard to interactively compile and test code during development. So the file loaded by Project > Debug, whose purpose is to set up your interactive development environment, defines these additional sections:

```
$2000 $2FFF IDATA SECTION EXT-IRAM
$3000 $3FFF UDATA SECTION EXT-URAM
$4000 $6FFF CDATA SECTION EXT-PRAM

EXT-PRAM  IDATA
```

This defines additional, larger, RAM spaces for data, and also defines a new **CDATA** section to accept new code downloaded via the XTL. The last line selects this RAM code space and specifies **IDATA** vectoring for the data-compiling word **C**, and similar words. (**IDATA** is the normal default section type.)

The current state of all section definitions, including the current section type and the current section of each section type—but not the contents of the sections—is called the *section context*. It may be saved and restored by **SAVE-SECTIONS** and **RESTORE-SECTIONS**.

Glossary
Section Type Selectors

CDATA	(—)
Select cData as the current section type.	
IDATA	(—)
Select iData as the current section type.	
UDATA	(—)
Select uData as the current section type.	
SAVE-SECTIONS	(— $n \times n$)
Save the entire current section context, consisting of n cells.	
RESTORE-SECTIONS	($n \times n$ —)
Restore the entire section context previously saved by SAVE-SECTIONS .	

Vectored Words

ORG	(<i>addr</i> —)
Set the address of the next available location in the current section of the current section type.	
HERE	(— <i>addr</i>)
Return the address of the next available location in the current section of the current section type.	
ALLOT	(n —)
Allocate n bytes at the next available location in the current section of the current section type.	

ALIGN	(—)	Force the space allocation pointer for the current section of the current section type to be cell-aligned.
ALIGNED	(<i>addr</i> — <i>a-addr</i>)	Force <i>addr</i> to be cell-aligned.
C,	(<i>b</i> —)	Compile <i>b</i> at next available location (cData and iData only).
W,	(<i>x</i> —)	Compile a word (the low-order 16-bits of <i>x</i>) at the next available location (cData and iData only). Available on 32-bit targets only.
,	(<i>x</i> —)	Compile a cell at the next available location (cData and iData only).

cData-specific Words

THERE	(— <i>addr</i>)	Return the address of the next available location in the current cData section.
GAP	(<i>n</i> —)	Allocate <i>n</i> bytes at the next available location in the current cData section.
C,C	(<i>b</i> —)	Compile <i>b</i> at the next available location in the current cData section.
W,C	(<i>x</i> —)	Compile a word (low-order 16-bits of <i>x</i>) at the next available location in the current cData section. Available on 32-bit targets only.
,C	(<i>x</i> —)	Compile a cell at the next available location in the current cData section.

References Compiling words and literals, *Forth Programmer's Handbook*, Section 2.9

4.4 DEFINING WORDS

Most defining words in SwiftX are used to create executable target definitions. Some—such as those created by **:** (colon) and **CODE**—are executable functions; others are data objects of various kinds. However, data objects may be thought of as having an executable component (or characteristic behavior), as well as a data component (or value). For example, a **CONSTANT**'s behavior is to return its value, whereas a **VARIABLE**'s behavior is to return the address of its data.

One defining word, **EQU**, does not exist in the target. An **EQU** is a constant for use by the cross-compiler, and has neither an executable component nor data space in the target. **EQU**s are typically used to name addresses, flags, sizes, etc., that the cross-compiler will use in preparing the target system. When used in a target colon definition, an **EQU** will be compiled as a literal.

Target defining words place their executable components in code space. Data-defining words such as **CREATE**—and custom defining words based on **CREATE**—make definitions that reference the section that is current when **CREATE** is executed.

Because uData is only *allocated* at compile time, there is no compiler access to it. uData is allocated by the defining words themselves; a summary of defining words is given below. At power-up, uData is uninitialized.

Space allocated by a single **BUFFER:** definition or by a single call to **RESERVE** is guaranteed to be contiguous, but there is no guarantee that space allocated by successive uses of these words will be contiguous with earlier allocations.

More details on the use of defining words, and how to make custom defining words, may be found in the *Forth Programmer's Handbook*.

Glossary

EQU <name> (*x* —)
 Define a one-cell constant in the host only, whose value is *x*. If an **EQU** is referenced inside a target colon definition, its value will be compiled as a literal. Execution of *name* returns *x*.

CREATE <name>	(—)
Define a named reference to the next available location in the current target section. Does not allocate any data space.	
CONSTANT <name>	(x —)
Define a one-cell constant whose value is x . Execution of <i>name</i> returns x . You cannot change the value of a CONSTANT .	
2CONSTANT <name>	(x_1 x_2 —)
Define a two-cell constant whose values are x_1 x_2 . Execution of <i>name</i> returns x_1 x_2 . You cannot change the value of a 2CONSTANT .	
VALUE <name>	(x —)
Define a one-cell, named value in iData, initialized to x . When executed, <i>name</i> will return its value like a CONSTANT . To change the value, use: <new- x > TO <name>	
TO <name>	(x —)
Store x in <i>name</i> 's data space. <i>name</i> must have been defined by VALUE .	
CVARIABLE <name>	(—)
Define a one-byte named location in uData. Execution of <i>name</i> returns the address of its data space.	
WVARIABLE <name>	(—)
Define a word-length (16-bit) named location in uData. Execution of <i>name</i> returns the address of its data space. Available on 32-bit targets only.	
VARIABLE <name>	(—)
Define a one-cell, named location in uData. Execution of <i>name</i> returns the address of its data space.	
BUFFER: <name>	(n —)
Define a named array of length n bytes in uData. Execution of <i>name</i> returns the address of the start of its data space.	
RESERVE	(n — <i>addr</i>)
Allocate n bytes of uData, starting at <i>addr</i> .	

References Defining words, *Forth Programmer's Handbook*, Section 2.7
Custom data objects, Section 4.7

4.5 MEMORY ACCESS

Words that access code and data space are described in this section. The behavior of these words depends critically on the context in which they are used. When they are used inside target definitions, they behave in a straightforward manner as described in the *Forth Programmer's Handbook*, referencing target data at given addresses.



However, if you use these words interpretively (e.g., typing them during debugging, outside of definitions in a source file, or in the compile-time part of a defining or compiling word) they have a special cross-compiler behavior that depends on whether you are connected to a running target system using the XTL (see Section 4.9.1). *Unless you are connected to a running target, access using the words described in this section is restricted to cData and iData.*

This section specifically describes the cross-compiler (interpreting or compiling) behavior of these words.

4.5.1 Basic Memory Access Commands

The following words are used to access individual bytes, cells, and words. As with the allocation and initialization words in Section 4.3, the primary words are vectored according to the current section type; but some convenient cData-specific equivalents do not affect the selection of the current section type.

Glossary

Vectored Words

C@	(<i>addr</i> — <i>b</i>)
Fetch a byte from <i>addr</i> in the current section of the current section type.	
w@	(<i>addr</i> — <i>x</i>)
Fetch a word (16-bits) from <i>addr</i> in the current section of the current section type. Available on 32-bit targets only.	
@	(<i>addr</i> — <i>x</i>)
Fetch a cell from <i>addr</i> in the current section of the current section type.	

C!	(<i>b addr</i> —)	Store a byte at <i>addr</i> in the current section of the current section type.
W!	(<i>x addr</i> —)	Store a word (the low-order 16-bits of <i>x</i>) at <i>addr</i> in the current section of the current section type. Available on 32-bit targets only.
!	(<i>x addr</i> —)	Store a cell at <i>addr</i> in the current section of the current section type.

cData-specific Words

C@C	(<i>addr</i> — <i>b</i>)	Fetch a byte from <i>addr</i> in the current cData section.
W@C	(<i>addr</i> — <i>x</i>)	Fetch a word (16-bits) from <i>addr</i> in the current cData section. Available on 32-bit targets only.
@C	(<i>addr</i> — <i>x</i>)	Fetch a cell from <i>addr</i> in the current cData section.
C!C	(<i>b addr</i> —)	Store a byte at <i>addr</i> in the current cData section.
W!C	(<i>x addr</i> —)	Store a word (low-order 16-bits of <i>x</i>) at <i>addr</i> in the current cData section. Available on 32-bit targets only.
!C	(<i>x addr</i> —)	Store a cell at <i>addr</i> in the current cData section.

References Memory allocation, Section 4.3

4.5.2 String Initialization and Management

The following words are used to initialize and manage strings. All are vectored according to the current section and section type. See Section 4.5 for the restrictions that apply to these words.

Glossary

BLANK		(<i>c-addr len</i> —)
	Fill area with spaces.	
ERASE		(<i>c-addr len</i> —)
	Fill area with zeros.	
FILL		(<i>c-addr len b</i> —)
	Fill area with <i>len</i> copies of <i>b</i> .	
MOVE		(<i>c-addr₁ c-addr₂ len</i> —)
	Copy <i>len</i> bytes from <i>c-addr₁</i> to <i>c-addr₂</i> .	
S" <text>"		(— <i>c-addr len</i>)
	Build a string, which must be terminated by " , at HERE and return its address and length.	

References String operations, *Forth Programmer's Handbook*, Section 2.3

4.6 COMPILER AND INTERPRETER

This section describes SwiftX tools used to create and test target commands. Forth's compiler and interpreter work together to process source text for this purpose. Strictly speaking, a Forth compiler is only operating between the colon that begins a definition and the semi-colon that ends it. The interpreter is processing the text, finding words and either executing them or delivering them to the compiler for processing. However, the consequence of executing most of the words encountered in source text causes definitions and data objects to be constructed and manipulated in the target image, both inside and outside of colon definitions, an activity that falls within the broad, commonly understood meaning of *compilation*. It is this broader definition that we will discuss in this section.

4.6.1 Basics of Compilation

A Forth compiler works by interpreting source text—e.g., by executing com-

mands that create data structures, commands that initialize data structures, and commands that begin compilation or that assemble machine instructions. Other commands may control various aspects of the compiler itself.

The glossary below lists examples of defining words used to begin compilation or when creating defining words. Data-space defining words are described in Section 4.4; commands that allocate or fill memory are listed in Section 4.3.

Glossary

CODE <name>	(—)
Begin assembling a word that will be executed when referenced. Target CODE words may not be referenced outside a colon definition unless the host is connected to the target with an active XTL.	
LABEL <name>	(—)
Begin assembling a word that will return the address of its code when referenced. A LABEL may be referenced at any time, inside or outside a definition.	
END-CODE	(—)
End a code word that was started with CODE or LABEL .	
: <name>	(—)
Begin compiling a Forth definition. Target definitions may not be referenced outside a colon definition unless the host is connected to the target with an active XTL.	
DOES>	(—)
Begin the run-time action of a new defining word written in high-level Forth. Used with CREATE (see the <i>Forth Programmer's Handbook</i> and Section 4.7).	
;CODE	(—)
Similar to DOES> , but the run-time action is written in assembler code.	
;	(—)
End a Forth definition that was started with : (colon).	

References Defining words, *Forth Programmer's Handbook*, Section 2.7

4.6.2 Compiler Scoping

Many details of this advanced subject are outside the domain of this document; they are covered more thoroughly in the *Forth Programmer's Handbook*. However, when developing a target application, sometimes it can be helpful to extend the supplied cross-compiler. To do so, it is necessary to understand that a Forth system implements multiple (e.g., host and target) versions of some words and selects among them by managing the way in which its dictionary is searched.

For example, a Forth cross-compiler includes an integral Forth compiler and assembler; they are used to build the target compiler and assembler which, in turn, are used to construct target code. But some words—from simple primitives to higher-level compiler words like **:** (colon) and **IF**—are defined in more than one of these places, and are implemented differently in each to provide specific functionality in each context.

To enable the programmer to access specific versions of compiler and assembler commands, Forth supports a *scoping* method based on named word lists and on the order in which the dictionary is searched. SwiftX's support for scoping is based on the optional Search-Order Wordset in ANS Forth. These facilities are used to define several compiler directives for managing scope in the cross-compiler.

The scopes defined in SwiftX are listed in Table 12. These words specify the scope to which new words will be added, as well as the words that are available to the compiler. Each of these remains in effect until explicitly changed.

By default, new commands belong to the **TARGET** scope; i.e., they are compiled onto the target. But after the **ASSEMBLER** command, new words will be added to the assembler and will be found while assembling machine instructions into the target. Likewise, after the **INTERPRETER** command, new words are added to the host that will be found when the host is interpreting on behalf of the target; and so on.

If you use any of these *scope selectors* (or *compiler directives*) to change the default scope, you must later use **TARGET** before commands can again be compiled to the target. SwiftX's status line (at the bottom of the command window) indicates the current scope, so you can easily verify, for example, that the current scope is **TARGET** when you attempt to interactively test a target word.

Table 12: Scope selectors

Command	Type of word to be compiled	Examples
ASSEMBLER	Words executed on the host while compiling assembly language definitions for the target. Typically, they provide machine code instructions and addressing modes.	MOV PUL JMP
COMPILER	Words used on the host while compiling target commands. Typically, they aid the target compilation process.	IF CASE ; LITERAL
HOST	Words that provide support for the cross-compiler and debugging.	LOGGING VERBOSE
INTERPRETER	Words executed on the host while interpreting on behalf of the target. Typically, they set up data structures, allocate memory, or start compilation.	CONSTANT ALLOT :
TARGET	Words compiled onto the target, available to target programs. (The default scope.)	DUP + !

The compiler directive in force *at the time you create* a new colon definition is the scope in which the new word will be found. As a trivial example:

```

TARGET ok
: Test1 1 . ; ok
Test1 1 ok

INTERPRETER ok
Test1
Error 0 TEST1 is undefined
ok

```

On rare occasions, while defining a new word, you might need to specify words from scopes other than the one currently in effect. Therefore, the specifiers in Table 13 are provided for use inside colon definitions being built in the host. They add the search orders **COMPILER**, **ASSEMBLER**, etc., to the beginning of the current search order, without otherwise changing the search order. These additions normally are for some limited, specific purpose, so the command [**PREVIOUS**] may be used to remove the most recent addition.

Table 13: Added search-order commands for extending the cross-compiler

Command	Search Order	Typical use
[+ASSEMBLER]	ASSEMBLER	Build assembler macros.
[+INTERPRETER]	INTERPRETER	Access the INTERPRETER behavior of a defining word.
[+HOST]	HOST	Access the HOST behavior of words defined for HOST and TARGET .
[+TARGET]	TARGET	Access TARGET words in an INTERPRETER or HOST definition.
[PREVIOUS]	Remove the most recently added search order.	Restore previous search order after completing the action enabled by one of the words above.

Glossary

ASSEMBLER (—)
Select **ASSEMBLER** scope. This provides access to the native instruction set of the target CPU. Selecting this scope lets you customize your assembler, for example, by adding macros.

COMPILER (—)
Select **COMPILER** scope. Words defined in this scope will be available for execution only while compiling target colon definitions; selecting this scope lets you customize your compiler.

Examples of **COMPILER** words include **IF**, **ELSE**, **THEN**, **BEGIN**, **UNTIL**, **DO**, **LOOP**, and similar words.

HOST (—)
Select **HOST** scope. **HOST** words are used on rare occasions to extend the host system to provide specialized compiler capabilities that will be used in scopes other than **TARGET**, or to execute the host versions of words that have been redefined in other scopes.

INTERPRETER (—)
Select **INTERPRETER** scope. Add the following definitions to the host envi-

ronment’s interpreter. **INTERPRETER** words are used to create target data objects. Examples include **CREATE**, **VARIABLE**, etc.

TARGET (—)

Select **TARGET** scope. This is the default compiler state. If you select another scope, you must re-assert **TARGET** before producing any more target definitions.

Target colon definitions are available for interactive execution and testing only when you are connected to a target via an interactive XTL. Target data objects are executable when interpreting, according to the guidelines in Section 4.6.4.

[**+ASSEMBLER**] (—)

Add **ASSEMBLER** scope to the current scope inside a colon definition. This gives access to mnemonics, addressing modes, etc., to define assembler macros.

[**+INTERPRETER**] (—)

Add **INTERPRETER** scope to the current scope inside a colon definition. This lets you access the compile-time behavior of a defining word.

[**+HOST**] (—)

Add **HOST** scope to the current scope inside a colon definition. This lets you access the underlying Forth system’s behavior of a word that is defined in both **HOST** and **TARGET** (e.g., @).

[**+TARGET**] (—)

Add **TARGET** scope to the current scope inside a colon definition. This lets you access target words from within an **INTERPRETER** or **HOST** definition.

[**PREVIOUS**] (—)

Remove the most recently added scope from the search order. This is normally used following the actions enabled by words such as [**+ASSEMBLER**] above, to return to the previous scope.

References Word lists and search order, *Forth Programmer’s Handbook*, Section 3.6

4.6.3 Effects of Scoping on Colon Definitions

The behavior of words defined in each of these scopes is different for colon definitions than for data objects, and also depends upon the state (compiling

or interpreting) in which they are invoked.

The default behavior of colon definitions defined in **HOST**, **INTERPRETER**, or **TARGET** when they are invoked in compiling state is to compile a reference to the word. The default behavior of words defined in **COMPILER** is to be executed, the usual consequence of which is to modify a colon definition that is being constructed.

The default behavior of colon definitions when they are invoked in interpreting state is to be executed. However, **COMPILER** words may *not* be invoked in interpreting state, and **TARGET** words may *only* be invoked when you are connected to a target via an active XTL.

Table 14 shows the accessibility of colon definitions defined in various scopes when they are invoked in interpreting and compiling states.

Table 14: Scopes in which colon definitions are accessible

If defined in:	Available in these scopes while	
	interpreting:	compiling:
ASSEMBLER	ASSEMBLER	ASSEMBLER (used to define macros)
COMPILER	Not allowed	TARGET
HOST	HOST, INTERPRETER, COMPILER	HOST, INTERPRETER, COMPILER
INTERPRETER	TARGET	INTERPRETER
TARGET	Not allowed unless connected	TARGET

Special state-dependent behaviors, if any, of words described in this manual will be described where they differ from the *default behavior* displayed when interpreting (for **HOST** and **INTERPRETER** words), compiling (for **COMPILER** words), or when the target is executing the word (for **TARGET** words).

4.6.4 Effects of Scoping on Data Object Defining Words

Defining words other than **:** (colon) are used to build data structures with characteristic behaviors. Many such words are included in SwiftX, and it is possible to construct new ones, if you wish.

Normally, a SwiftX programmer is primarily concerned with building data structures for the target system; therefore, the dominant use of defining words is in the **TARGET** scope while in interpreting state. You may also build data objects in **HOST** that may be used in all scopes except **TARGET**; such objects might, for example, be used to control the compiling process. Data objects fall into three classes:

- *iData objects* in initialized data memory—e.g., words defined by **CREATE**, **VALUE**, etc., including most user-defined words made with **CREATE ... DOES>**.
- *uData objects* in uninitialized data memory—e.g., words defined by the use of **VARIABLE**, **BUFFER:**, etc.
- *Constants*—words defined by **CONSTANT**, **2CONSTANT**, and **USER**.

Unlike target colon definitions, target data objects may be invoked in interpreting state. However, they may not exhibit their defined target behavior, because that is available only in the target (or in interacting state). Constants will always return their value; other words will return the address of their target data space address. *iData* objects may be given compiled, initial values with **,** (comma) and **C,** (c-comma), and you may also use **@** and **!** with them regardless of whether you're connected to a target with an active XTL. However, there is no way to initialize *uData* objects at compile time, and you may only access their data space when the XTL is connected and active.

4.7 CUSTOM DATA OBJECTS

One of the most powerful features of Forth is the ability to construct custom data-object defining words. The basic principles of custom defining words in Forth are discussed in the *Forth Programmer's Handbook*.

However, some special issues arise when creating custom data objects in a cross-compiled environment: defining words are executed on the *host*, to create

new definitions that can be executed on the *target*. Therefore, you must be in the **INTERPRETER** scope (see Section 4.6.2) when you create a custom defining word, and you must be aware of what data space you are accessing (see Section 4.3) in the new data object.

Consider this example:

```

INTERPRETER
\ PRINTS defines words that display their values.

: PRINTS ( n -- )
  CREATE ,          \ New definition with value n.
  DOES> ( -- )      \ Execution behavior.
    @ . ;          \ Fetch value and display it.

TARGET

1 PRINTS ONE
2 PRINTS TWO

```

ONE and **TWO** are target definitions, *instances* constructed by the defining word **PRINTS**. Each instance has its own value, but all objects defined by **PRINTS** share the run-time behavior (@ and .) associated with **PRINTS**.

You must specify **INTERPRETER** before you make the new defining word, and then return to **TARGET** to use this word to add definitions to the target. The **INTERPRETER** version of **DOES>** allows you to reference **TARGET** words in the execution behavior of the word, since that will be executed only on the target.

When **CREATE** (as well as other memory allocation words listed in Section 4.3) is executed to create the new data object, it uses the *current section type*. The default in SwiftX is **iData**. Defining words that explicitly use **uData** (**VARIABLE**, **BUFFER:**, etc.) do not affect the current section type. If you wish to force a different section type, you may do so by invoking one of the selector words (**CDATA**, **IDATA**, or **UDATA**) inside the defining portion or before the defining word is used. If you do this, however, you must assume responsibility for re-asserting the default section. You may choose to use **SAVE-SECTIONS** and **RESTORE-SECTIONS** (described in Section 4.3).

You can control where individual instances of **PRINTS** definitions go, like this:

```

CDATA
1 PRINTS ONE

IDATA
2 PRINTS TWO

```

In this case, the data space for **ONE** is in code space, but the data space for **TWO** is in initialized data space. (Not all processors support data objects in code spaces, so **TWO** is portable and **ONE** is not.)

Alternatively, assuming your processor permits it, you could define **PRINTS** to explicitly assert cData:

```

: PRINTS ( n -- )
  CDATA          \ Select code section.
  CREATE ,       \ New definition with value n.
  IDATA          \ Restore default iData section.
  DOES> ( -- )   \ Target execution behavior.
    @ . ;       \ Fetch value and display it.

```

In this case, both the **CREATE** and the **,** (comma) will use cData.

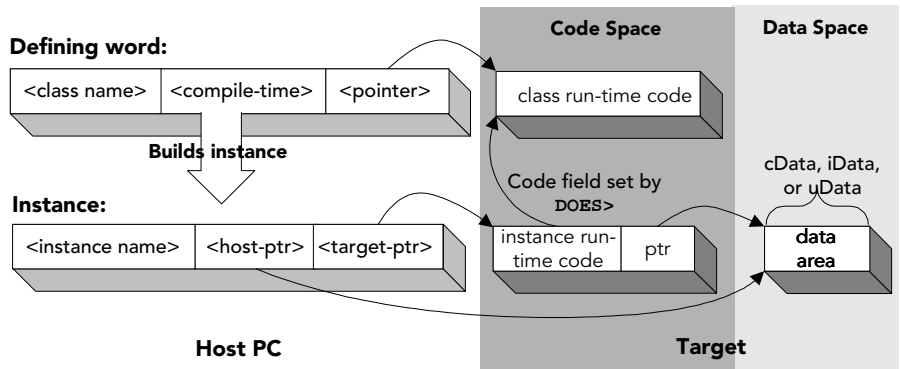


Figure 12. A defining word and an instance, in host and target

Figure 12 shows the action of a defining word. The run-time behavior of all its instances exists at one place in the target cData. The host part of the defining word can compile instances of its class, each of which has a definition in the host, a run-time behavior in the target, and a data space in the target. The host

portion of an instance contains two pointers: the *host pointer* is to the data area in target address space, while the *target pointer* is to the instance's run-time code in the target. The instance's run-time code will call the general run-time code for the class, providing the address of the instance's data area. The data area may be in cData, iData, or uData, but is usually in iData.

The demo program `SwiftX\Src\Conical.f` (see Section 1.4.3) contains a good example of a custom defining word, **MATERIAL**, used to define materials. When executed, a word that has been defined by using **MATERIAL** (an *instance* of **MATERIAL**) stores its parameters in the variables used in the demo program.

If you invoke an instance's name interpretively on the host, its behavior depends on whether you're connected to the target with an active XTL. If you're connected, the target pointer will be used to execute the target's run-time behavior. If not, the host pointer will be used to return the address of its data area; if that is in cData or iData, you may read or write that location in the host's target image.

It is possible to define custom host behaviors that emulate target behaviors (assuming, of course, that it's possible and doesn't require access to target-only devices or other features). The technology used to do this is called *twins*, because it involves making twin dictionary entries in the Interpreter and Target scopes. The two key words are **TWIN** (which makes a duplicate dictionary entry with one data cell, like a **VARIABLE**) and **2TWIN** (which makes a duplicate entry with two data cells, like a **2VARIABLE**). These words are already built into pre-existing defining words that return values (e.g., **CREATE**, **VARIABLE**, **CONSTANT**), but are also available for defining custom host behaviors.

Consider this example:

```
HOST
VARIABLE >FCB                \ Current field, host copy
: (FCB:) ( x -- ) TWIN      \ Host behavior
  DOES> ( i -- addr )    @ >FCB @ + ;

TARGET
VARIABLE >FCB                \ Current field, target copy

INTERPRETER
```



```

: FCB: ( n1 -- n2 )      \ Defining word definition
  DUP (FCB:)            \ Make host twin
  CREATE DUP , CELL+    \ Make target twin
  DOES> @ >FCB @ + ;    \ Target behavior

```

TARGET

```

0 FCB: ORG      FCB: LIM      FCB: R#      DROP

```

In this example, **FCB:** defines fields within a file control block in data space. The host twin word returns the field in the structure whose address is contained in the host variable **>FCB**. The target word returns the field pointed to by the target variable **>FCB**. The host twins are useful for defining precompiled structures in iData.

TWIN and **2TWIN** should always be used in conjunction with a defining word (**CREATE** in the above example) that creates target definitions. The built-in **TWIN** in words like **CREATE** and **CONSTANT** is automatically bypassed if you use an explicit **TWIN**, as in the example.

Glossary

- TWIN** <name> (x —)
 Make a duplicate dictionary entry for *name* in the current scope, leaving the input stream unchanged (i.e., *name* still visible), and give it the initial value x .
- 2TWIN** <name> (x_1 x_2 —)
 Make a duplicate dictionary entry for *name* in the current scope, leaving the input stream unchanged (i.e., *name* still visible), and give it the initial values x_1 and x_2 .

4.8 SAVING A COMPILED OBJECT IMAGE

SwiftX can generate the three types of object file formats shown in Table 15. Two commands, shown in the glossary below, are available to create object files suitable for PROM programmers and for downloading. In each case, the first letter of the filename extension determines the format that will be used.

Table 15: Object-file format selection

First letter of extension	Format	Example
H	Intel Hex format	Target.hex
S	Motorola S-record format	Target.s19
any other	Binary data	Target.bin

If you have multiple cData or iData sections, each must be saved separately. For example, to save two cData sections named **ProgA** and **ProgB**, you might use:

```
ProgA SAVE-CODE PROGA.H
ProgB SAVE-CODE PROGB.H
```

For an example, see the end of `\Swiftx\Src\<CPU>\<terminal>\Build.f` for the commands that construct the target image.

Glossary

SAVE-CODE <name.ext> (—)
Records the current cData section in the file *name.ext*, where the filename extension *ext* indicates the format to be used, per Table 15.

SAVE-DATA <name.ext> (—)
Records the current iData section in the file *name.ext*, where the filename extension *ext* indicates the format to be used, per Table 15.

4.9 THE CROSS-TARGET LINK

The Cross-Target Link, or XTL, is a key feature of SwiftX. It provides an extraordinary level of interactivity and debugging power, facilitating thorough testing of code and interactive testing of the hardware. Major features of the XTL are discussed in this section.

4.9.1 Target/Image Interactions

The cross-compiler can operate while *connected* (i.e., interacting with a terminal using the XTL) or *disconnected* (not connected, or not using the XTL). Building a program image for later downloading (the Project > Build menu option) may be done in either mode, because the cross-compiler keeps an image in host memory of each iData and cData section that has been defined (described in Section 4.3).

When you are compiling a program image for later downloading, the words in Table 16 access a *host image* of the current section being constructed or maintained. In such circumstances, you may only access cData and iData.

When interacting with actual target memory via the XTL, you may access *any* type of space, including uData. New definitions (including colon definitions and new data structures) will modify the host image. On systems that support interactive compilation, such changes will also be downloaded to the target, immediately available for interactive testing. See your platform-specific documentation.

The **CONNECT** command establishes an XTL connection. Following this command, memory fetches are made from the remote device, and memory stores go to both the target and the host image. The **DISCONNECT** command ends the connection; thereafter, memory fetches and stores are to the host image only. Memory commands that are directly affected are shown in Table 16; they are described further in Section 4.5.1.

Table 16: Memory access words affected by target connection

Words		Descriptions
C@	C!	Byte (character) fetch and store in the current section
W@	W!	Word (16-bit) fetch and store in the current section (available only on 32-bit targets)
@	!	Cell fetch and store in the current section
C@C	C!C	Byte (character) fetch and store to code space
W@C	W!C	Word (16-bit) fetch and store to code space (available only on 32-bit targets)
@C	!C	Cell fetch and store to code space

The XTL is used in one of two modes:

- **Fully interactive**, in which you can examine target memory and execute target definitions and, also, if you type a colon or code definition it will be immediately downloaded to the target, ready for testing.
- **Batch**, in which the target is temporarily disconnected and an image of your code is compiled in the host memory for later downloading. This mode is much faster if you're compiling one or more files of source, as opposed to typing single definitions.

These modes are controlled by **SET-DOWNLOAD** and **DOWNLOAD-ALL**. You can see how these are used in the file **Debug.f**, where you'll find a sequence like:

```
SET-DOWNLOAD
```

```
INCLUDE ..\..\CONICAL      \ Conical piles demo
INCLUDE ..\..\DUMB        \ Dumb terminal
```

```
256 TERMINAL CONSOLE
```

```
: DEMO    CONSOLE ACTIVATE  DUMB SCI-TERMINAL
      BEGIN  CALCULATE  AGAIN ;
DOWNLOAD-ALL
```

SET-DOWNLOAD is called before the files are **INCLUDED**, the task defined, and the startup word **DEMO** defined. These actions occur in batch mode. When they are complete, **DOWNLOAD-ALL** downloads the compiled code and reconnects the XTL in full interactive mode. To test your application interactively, all you have to do is replace the demo files and definitions with your application.

To support interactive debugging, the image of the target dictionary and data space in the host must be an exact match for the executing one in the target. To ensure this, SwiftX places a checksum of a compiled kernel in the kernel image. Near the beginning of **Debug.f**, the word **SYNC-CORE** is executed. This word compares the current host image's checksum with the one in the target. If they don't match, it will abort with the error message, "Mismatch". Should this occur, you must install a matching kernel in your target by following the instructions in your target-specific manual.

Glossary

- CONNECT** (—)
Start the XTL communicating with the target. **CONNECT** will abort if target communication cannot be established (e.g., it is not connected or not responding). When communication is established, the host can execute words on the target, examine its memory, etc.
- DISCONNECT** (—)
Shut down the XTL link. When this is done, you may no longer execute target words. You may, however, examine the host image of cData and iData, and perform host functions such as **LOCATE**, **WH**, etc.
- SET-DOWNLOAD** (—)
Disconnect the XTL, save target code and data space pointers, and enter batch mode. All subsequent compilation affects the host's image of target cData and iData only.
- DOWNLOAD-ALL** (—)
Re-connect to the target. Compare the saved code and data space pointers with the current ones, and download all new cData and iData. Leave the target in interactive mode, ready to test the new code.
- SYNC-CORE** (—)
Ensure that the host and target images of code space are synchronized by comparing their checksums. If the comparison fails, interactive testing will not be possible until a matching kernel is installed in the target.

4.9.2 XTL Protocol

Figure 13 is an overview of the logical relationship between the host and target using a standard serial XTL. Implementations using Motorola's Background Debugging Mode (BDM) may differ; see your platform-specific documentation.

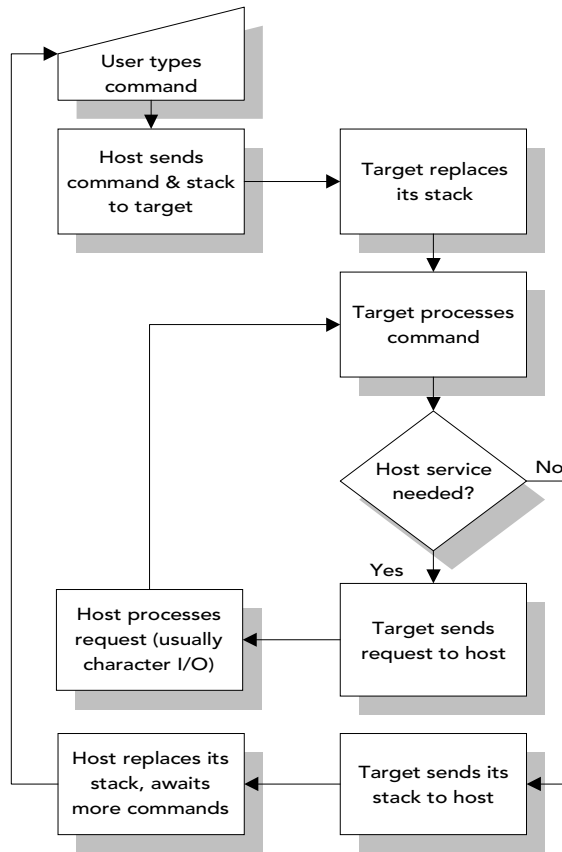


Figure 13. XTL communication

Table 17 lists each command from the host to the target, parameters sent to the target, and parameters received from the target in response to the command.

Table 17: Host-to-target commands

Fn	Description	Parameters to target	Parameters from target
0	Execute target word	$n(1), data(n*4)$	$\{cmd(1)\}n(1), data(n*4), ack(1)$
1	Send back register contents		CPU dependent
2	Fetch byte from data space	$addr(4)$	$data(1), ack(1)$
3	Store byte to data space	$addr(4), data(1)$	$ack(1)$
4	Fetch half-cell (16-bits) from data space	$addr(4)$	$data(2), ack(1)$
5	Store half-cell to data space	$addr(4), data(2)$	$ack(1)$
6	Fetch cell from data space	$addr(4)$	$data(4), ack(1)$
7	Store cell to data space	$addr(4), data(4)$	$ack(1)$
8	Download to data space	$n(1), addr(4), data(n)$	$ack(1)$
9	Fetch byte from code space	$addr(4)$	$data(1), ack(1)$
10	Store byte to code space	$addr(4), data(1)$	$ack(1)$
11	Fetch half-cell from code space	$addr(4)$	$data(2), ack(1)$
12	Store half-cell to code space	$addr(4), data(2)$	$ack(1)$
13	Fetch cell from code space	$addr(4)$	$data(4), ack(1)$
14	Store cell to code space	$addr(4), data(4)$	$ack(1)$
15	Download to code space	$n(1), addr(4), data(n)$	$ack(1)$

Commands from host to target consist of a single byte, followed by optional parameters; the size of each parameter, in bytes, is noted in parentheses. The target executes the command, returns its optional parameters, and ends with a positive acknowledge (ack) code (see Table 18). Multi-byte parameters are sent most-significant byte first. The half-cell operations (Fn 4, 5, 11, and 12) are available on 32-bit targets only.

All data transmitted, in both directions, are binary 8-bit characters. One exception is the Esc character ($1B_H$), which has special meaning when transmitted to

the target. Esc followed by any other character tells the target to abort the current command in progress and to return an “Announce reset” (255) response. Two Esc characters in a row indicate a single data byte whose value is $1B_H$. The Esc character has no special meaning when transmitted from the target to the host.

The *data* indicated in Table 17 for function 0 (execute target word)—for both host and target—is the parameter stack. The stack is passed bottom item first. When sent from the host to the target, the top two stack items are the execution address for the word and the host’s **BASE**. (If the target implements **CATCH** and **THROW** exception handling, it executes the word using **CATCH**.) When returned by the target to the host, the top two stack items are the host’s **BASE** and the function’s **THROW** code (0 indicates successful completion or that **CATCH/THROW** isn’t implemented on the target). Thus, the value *n* (number of stack items) is always at least two, in both directions. The *cmd* parameter in the “from target” field can be any of the responses listed in Table 18. The host will not begin receiving the target stack until it receives the Ack response, indicating that the target has finished executing its word.

Table 18: Target-to-host responses

Fn	Description	Parameters from target	Parameters to target
255	Announce reset		
254	Ack; command completed successfully		
253	Nak; command aborted ¹		
252	KEY (input key and send it back)		<i>char</i> (1)
251	KEY? (test for keypress, send status back)		<i>flag</i> (1)
250	Display address	<i>addr</i> (4)	
249	ACCEPT (input string, return actual length)	<i>addr</i> (4), <i>length</i> (1)	<i>length</i> (1)
248	AT-XY (cursor position)	<i>row</i> (1), <i>col</i> (1)	
13	CR (new-line function)		
12	PAGE (clear-screen function)		

1. Target issues function 253 on start-up and on restart.

Table 18 lists the possible *cmd* responses from the target; they are only valid while the host awaits the initial response to function 0 (execute target word). These response codes enable the host to provide virtual terminal services to the target. The host will acknowledge any of the terminal output functions by sending an unspecified character to the target, to pace output.

If the program under test never returns control to the host, pressing the Esc key will display the message `ESCAPE` and abort out of the wait loop. This is useful when testing a routine whose behavior is an infinite loop, or when a program behaves unexpectedly.

Except for the codes listed in Table 18, all characters sent from the target to the host will be displayed by the host in the command window.

Targets using the Motorola Background Debugging Mode (BDM) may have a slightly different command sequence; if you have one of these systems, check your platform-specific documentation for details.

5. THE SWIFTOs MULTITASKING EXECUTIVE

Multitasking allows a computer to appear to be doing many things at once. In particular, the SwiftOS multitasker provides service to multiple programs operating without any fixed timing relationship (i.e., asynchronously). This section explains how tasks are constructed, how they are controlled, and how the CPU is shared between them.

SwiftOS supports two types of tasks: *background tasks* and *terminal tasks*. These are fundamentally identical, but a terminal task can be thought of as a more elaborate background task tailored to service a serial-type device. You can easily prune unnecessary resources from a terminal task, or add more to either kind of task.

Each task has a private area of uData space containing its data and return stacks and a *user area* for task-specific variables.

A task, in SwiftOS, 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 run a hydraulic lift might be named **LIFTER**.

There are three aspects to task definition and control:

1. **Task definition** takes place at compile time. Tasks are given fixed allocations of memory in which to operate, including stack space and user area.
2. **Task initialization** takes place after power-up in the target system. At this time, the task’s RAM areas are initialized and it is linked into the running multitasker loop.
3. **Task activation** may take place at one or more points in the running application. This involves giving the task words to execute. It may be a temporary

assignment (execute this, then stop) or a permanent one (start running an infinite loop).

Although the definition and physical structure of a task is static, its job assignments may change according to the needs of the application.

5.1 FORTH RE-ENTRANCY AND MULTITASKING

When more than one task can share a piece of code, that code can be called *re-entrant*. Re-entrancy is valuable, because memory is conserved when tasks share code.

Routines that are not re-entrant are those containing elements that are subject to change while the program runs. Thus, self-modifying code is not re-entrant. Routines with *private variables* are not re-entrant, but re-entrant routines can have *private constants* (because a constant's value does not change). Re-entrant routines can always be programmed into ROM.

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

Since re-entrancy is easily achieved, tasks may share routines in a single program space. This conserves large amounts of memory. In most applications, all system and application routines can be shared (with the minor exception of the I/O instructions on certain processors). Terminal tasks (such as printer spooling) can operate with only a few hundred bytes. The minimum size of a useful task is about 512 bytes, devoted to the task's stacks and user variables. Some applications (PBXs, process control, and some communications systems) naturally use large arrays of small tasks, with each task running a simple shared program.

References User variables, Section 5.2.3

5.2 PRINCIPLES OF OPERATION

The SwiftOS multitasker is designed to fulfill several objectives:

1. Concurrent, asynchronous execution of code.
2. Convenient to use
3. Fast in execution
4. Simple to understand
5. Minimal memory requirements
6. Independent of hardware configuration (for example, a timer or memory manager is unnecessary)

The SwiftOS multitasker satisfies 2, 3, and 4 above by consisting of only about 13 words. Number 5 is a consequence of Forth's inherently re-entrant structure (see Section 5.1). Numbers 1 and 6 are ensured by the way SwiftOS schedules tasks—SwiftOS services tasks when an executing task stops to await I/O.

Simplicity and good performance are ensured because task-switching only happens at known, programmer-controllable points, and between Forth words. This greatly simplifies the context-switching operation (thus reducing overhead) and the job of writing routines for a multi-user environment.

5.2.1 Task-Scheduling Algorithm

This section provides a detailed discussion of the scheduling algorithm, its associated words, and some useful techniques. Processor-specific details of the SwiftOS implementation are described in a separate document accompanying your SwiftX product.

The SwiftOS multitasker may be said to be *I/O driven*. A *round-robin* algorithm (see Figure 14) schedules processor time. Each task has control until it executes the high-level **PAUSE** or **STOP**, or the assembler code ending **WAIT**. Most words performing asynchronous hardware operations (e.g., **TYPE**, **ACCEPT**) contain a **WAIT** or a jump to **PAUSE** so, while a task is waiting for an I/O operation to be completed, other tasks can use the CPU. Since Forth is naturally very fast, tasks tend to spend much of their time awaiting I/O. Tasks that perform extensive computations may be prevented from impacting overall system performance

by incorporating **PAUSE** into a few regularly executed or CPU-intensive words.

The round robin is sometimes called the **PAUSE loop**, the *idle loop*, or the *multi-tasking loop*. Where possible, it is implemented as an endless loop of jump instructions (see Figure 14). Each task has its own jump instruction, which transfers control to the jump instruction of the next task.

When a task is being scheduled to awaken, the task's jump instruction is replaced by an instruction to transfer control to the machine code that awakens the task. This special instruction is usually called **WAKE**, and is usually a trap or subroutine call. The address of the first byte of a task's jump instruction is pushed on the stack by the high-level Forth word **STATUS**, a user variable (see Section 5.2.3) for the current task. In assembly code, the address of the current task's **STATUS** is available in register **U**. The address used by the jump in **STATUS** immediately follows it.

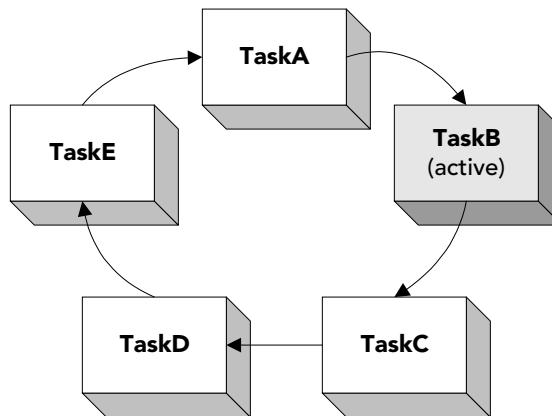


Figure 14. SwiftOS round-robin multitasking loop

The most general case of changing tasks in the round robin occurs when CPU control is relinquished from high-level Forth, with arrangements for the task to awaken and resume execution on its next turn. **PAUSE** performs this most general case. **PAUSE** can be embedded in complex calculations which perform no I/O and which might otherwise cause a particular task to control the CPU for undesirably long periods. Consider the following collision-orbit calculation:

```

: POSITION    X STEP  Y STEP ;
: ?COLLIDE   30000 0 DO  POSITION HIT
    PAUSE LOOP ;

```

In this example, the word **STEP** is assumed to have been defined to perform the calculations for integrating the next step in the target X or Y coordinate. **HIT** expects the coordinate of the target (computed by **POSITION**) on the stack and performs appropriate course corrections. Because these computations are time consuming, and because other functions must be running concurrently, it is desirable to give up the CPU for one turn around the round robin for each step in the integration. Inserting the **PAUSE** in the loop accomplishes this.

The related words **STOP** and **WAIT** (see the glossary below) share much of the code for **PAUSE**. These are the steps **PAUSE** performs:

1. The **WAKE** instruction is stored into **STATUS**, replacing the round-robin jump. This ensures that the task executing **PAUSE** will awaken on its next turn. On many machines, **WAKE** is equivalent to a subroutine call to the code for step 5, below.
2. The system pointers **I** and **R** are saved by being pushed onto the current task's data stack. This portion of the code is the entry point for **STOP** and **WAIT**, which by skipping step 1 do not automatically resume execution of the current task on the next turn. (On some implementations, the address interpreter pointer **I** is not needed; see your platform-specific SwiftX documentation for details.)
3. The data stack pointer (**S**, in assembler) is saved in a reserved location in the user area. At the completion of this step, all non-recoverable unshared task data for the address interpreter has been saved. Tasks only relinquish the CPU between Forth words, so other registers do not have to be saved.
4. The CPU jumps to the location whose address follows **STATUS** (i.e., the next task's **STATUS**), and proceeds to jump through the circular round-robin loop until a **WAKE** instruction is encountered. The **WAKE** transfers control to step 5.
5. The address of the new task's **STATUS** is stored in **U**. The address of **STATUS** can be obtained from the address left by **WAKE**.
6. Using **U** to find the task, the data-stack pointer is restored, then **I** and **R** are restored from the new task's data stack.
7. The **SLEEP** instruction (a jump to the next task) is stored into the current task's **STATUS** to make the current task's state "don't awaken."
8. Finally, **PAUSE** is exited and the task will execute its next word, based on the

contents of **I** and **R**.

The words in the glossary below control task use of the CPU.

Glossary

PAUSE	(—)
Suspend the task that calls PAUSE to allow all other tasks one turn in control of the CPU.	
STOP	(—)
Put the task that calls STOP to sleep until that task is awakened by an interrupt routine or by some other task.	
WAIT	(—)
An assembler code ending on some systems that is equivalent to STOP .	
WAKE	(— <i>x</i>)
Return <i>x</i> , the machine instruction (usually a trap instruction or subroutine call) that may be stored in a task's STATUS to cause the task to be awakened at its next turn in the multitasker round robin. See your platform-specific documentation for implementation details.	
SLEEP	(— <i>x</i>)
Return <i>x</i> , the machine instruction (a jump to the task whose address follows the current task's STATUS) that may be stored in a task's STATUS to cause the task to remain inactive (skip its turn in the round robin). See your platform-specific documentation for implementation details.	

<i>References</i>	Forth re-entrancy and multitasking, Section 5.1
	User variables, Section 5.2.3
	TYPE , ACCEPT , serial I/O in general, <i>Forth Programmer's Handbook</i> , Section 3.8

5.2.2 Interrupts and Tasks

Interrupt routines are often used to awaken tasks. A common way to perform complex, non-critical interrupt servicing is to have the interrupt routine perform all time-critical operations, and then store **WAKE** into a task's **STATUS**. For

example, if a hypothetical data acquisition word **ACQUIRE** solicits data from a serial device, it could send a request to the device and then **STOP**. Input from the device could be buffered by interrupt code; when the interrupt code sees a carriage return, it would awaken the terminal task associated with the interrupting device. When the round robin gets around to the task, the task resumes execution at the word immediately following **STOP**, and runs until it executes another **STOP** or **PAUSE**, or **WAITS** for an I/O operation.

Tasks typically perform asynchronous operations such as data reduction and logging. An example of a data reduction loop being run by a dedicated task might be:

```

: COLLECT    BEGIN ACQUIRE DATA REDUCE
      ARCHIVE STOP AGAIN ;

```

The interrupt routine that services the data source for this example will awaken the task by storing **WAKE** in the task's **STATUS** when data is ready to be accepted.

5.2.3 User Variables

In SwiftOS, tasks can share code for the text interpreter, I/O drivers, etc., but each task will have different data for these facilities. The fact that all users have private copies of variable data for such shared functions enables them to run concurrently without conflicts. For example, number conversion in one task needs to control its **BASE** value without affecting that of other tasks.

Such private variables are referred to as *user variables*. User variables are not shared by tasks; each task has its own set, kept in the task's *user area*.

- Executing the name of a user variable returns the address of that particular variable within the task that executes it.
- Invoking <task-name> @ returns the address of the first user variable in *task-name's* user area, which is generally named **STATUS**.

Some user variables are defined by the system for its use. You may add more in your application, if you need to in order to preserve re-entrancy, to provide private copies of the application-specific data a task might need.

User variables are defined by the defining word **+USER**, which expects on the

stack an offset into the user area plus a size (in bytes) of the new user variable being defined. A copy of the offset will be compiled into the definition of the new word, and the size will be added to it and left on the stack for the next use of **+USER**. Thus, when specifying a series of user variables, all you have to do is start with an initial offset and then specify the sizes. When you are finished defining **+USER** variables, you may save the current offset to facilitate adding more later. The conventional way to do this is by using **EQU** (which makes a host-only constant and, conveniently, removes the last offset from the stack).

The minimum SwiftOS user area begins something like this (details vary, depending upon your target CPU):

```
0 2 +USER STATUS
CELL +USER FOLLOWER
CELL +USER SSAVE
CELL +USER S0
CELL +USER CATCHER
EQU #USER
```

Here, the task's status area starts at offset 0 relative to the beginning of the user area, and provides two bytes for a jump to the next task. Next, one cell is provided for the address of the next task in the loop, followed by a cell for the "stack-pointer save" location, and then **S0**, etc.

Additional user variables might be assigned as follows:

```
\ Terminal task user variables
#USER CELL +USER 'EMIT
      CELL +USER 'TYPE
      CELL +USER 'CR
      CELL +USER 'PAGE
      CELL +USER 'ATXY
      CELL +USER 'KEY
      CELL +USER 'KEY?
      CELL +USER 'ACCEPT
      CELL +USER DEVICE
EQU #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 all over your source.

A user variable is defined as an offset into the user area, where a zero offset usually corresponds to **STATUS**. The offset is the number passed on the stack during a sequence of **+USER** definitions; the current value of the offset is compiled into each user variable definition, and the offset left on the stack is incremented by the number of bytes to be reserved (one cell, in most of the examples above). Eventually, 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 Figure 15; also task definition, Section 5.3.1 and Section 5.4.1).

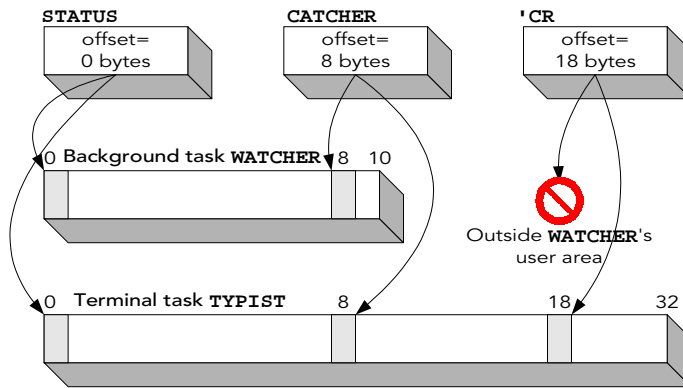


Figure 15. User variables apply an offset to a task area*

The system does not test an address returned by invoking a user variable to see if it is within the space allotted to the current task's user variables. It is your responsibility not to let a task reference a user variable outside its user area!

The *minimum* size for a background task's user area would be about five cells for the example on the previous page, for **STATUS** through **CATCHER**. Terminal tasks generally have a larger user area to accommodate I/O, task management, dictionary management (in programmable systems—see Section 7.1.3), text interpretation, and other functions.

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**

* Example is for a 16-bit target

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 terminal task named **CHUCK** to 2 (binary). In this example, **HIS** takes the address of the executing task's **BASE** and subtracts the **STATUS** address of the executing task from it to get the offset, then adds the offset to the **STATUS** address of the desired task, supplied by **CHUCK**.

The user variable definitions provided for **COLLECT** in the sample data logger developed below are examples of application-specific user variables. Note how **HIS** is used to transfer a value from the stack of the task that starts the definition **RECORD** to the user variable **#SAMPLES** belonging to the task **SCRIBE**, which will execute the words **COLLECT** and **STOP** following **ACTIVATE**.

This example assumes that **A/D** will request a sample from an analog-to-digital converter, **PAUSE** while the conversion takes place, and signal via an interrupt when a sample is ready. The interrupt should read the sample, put it in a temporary place, then set the task to awaken. The rest of **A/D** will get the value and put it on the stack. This scenario is typical of the relationship between interrupt code and task-level code.

```
#USER CELL +USER #SAMPLES          \ Current # of samples
10000 CONSTANT MAX-SAMPLES          \ Maximum # of samples
MAX-SAMPLES CELLS BUFFER: SAMPLES    \ Sample buffer

: COLLECT ( -- )                    \ Record #SAMPLES samples
  #SAMPLES @ 0 DO                  \ Loop setup
    A/D                            \ Read one sample
    SAMPLES I CELLS + !            \ Store in Ith cell
  LOOP ;                           \ Repeat

: RECORD ( n -- )                  \ User command: n RECORD
  MAX-SAMPLES MIN                  \ Clip n to legal value
  SCRIBE #SAMPLES HIS !            \ Pass n to task
  SCRIBE ACTIVATE                  \ Start task
  COLLECT STOP ;                  \ Task collects, stops
```

This version of **RECORD** would be used with the number of samples as its argument; the task address is built in **SCRIBE**. The following phrase would

record 500 samples:

500 RECORD

The glossary at the end of this section shows the user variables required by a fully interactive SwiftOS terminal task. Your CPU's implementation may require variables not listed—see your SwiftX implementation's system and user variables defined in **Swiftx\Src\<CPU>\data.f**.

Although the order in which they are defined varies, all systems possess these user variables. You can obtain the absolute address of location 0 in a task's user area by typing:

```
<task-name> @
```

All user variables return the address of the value for the task that is executing when they are invoked.

See **Data.f** for the exact organization of user variables in your system. Generally, the most-used user variables are defined first, so tasks needing only some of the user variables can have a minimal-sized user variable area.

The first glossary below lists words used to manage the user variables. It is followed by the user variables in two groups: those required for all tasks, and those used only by terminal tasks.

Glossary

User-variable management

+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)

Return the number of bytes currently allocated in a user area. This is an appropriate offset for the next user variable when this word is used to start a sequence of **+USER** definitions intended to add to previously defined user variables.

HIS

($addr_1$ n — $addr_2$)

Given a task address $addr_1$ and user variable offset n , returns the address of the referenced user variable in that task's user area. Usage:

<task-name> <user-variable-name> **HIS**

User variables required for all tasks

STATUS	(— <i>addr</i>)
Indicates whether the task is ready to become active, by containing a jump to the wake up code or to FOLLOWER .	
FOLLOWER	(— <i>addr</i>)
Address of the next task in the multitasking chain.	
SSAVE	(— <i>addr</i>)
Stack pointer, saved when the task was last active.	
S0	(— <i>addr</i>)
Pointer to the bottom of the data stack and the start of the message buffer (for terminals). The data stack grows toward low memory from <i>addr</i> , and the message buffer, if any, extends toward high memory from this same <i>addr</i> .	
CATCHER	(— <i>addr</i>)
Pointer to the latest exception frame (0 if none), set by CATCH .	

User variables required for terminal tasks

DEVICE	(— <i>addr</i>)
Terminal device address or other device information.	
BASE	(— <i>addr</i>)
Address of the variable containing the number conversion base (eight for octal, ten for decimal, sixteen for hex).	
'EMIT	(— <i>addr</i>)
Address of the task's EMIT routine.	
'TYPE	(— <i>addr</i>)
Address of the task's TYPE routine.	
'CR	(— <i>addr</i>)
Address of the terminal new-line routine.	

'PAGE	(— <i>addr</i>)
Address of the terminal screen clear or form-feed routine.	
'ATXY	(— <i>addr</i>)
Address of a routine to position the terminal's cursor.	
'CLEAN	(— <i>addr</i>)
Address of the task's "clear to end of line" routine.	
C#	(— <i>addr</i>)
Task's current cursor position (column), set to 0 by CR and PAGE , and maintained as appropriate by EMIT , TYPE , and AT-XY .	
L#	(— <i>addr</i>)
Task's current cursor position (line), set to 0 by PAGE , and maintained as appropriate by CR and AT-XY .	
TOP	(— <i>addr</i>)
Top of task's screen-scrolling area, expressed as a line number, with 0 signifying the top of the screen.	
'KEY	(— <i>addr</i>)
Contains the most recent character received since the last ACCEPT or KEY ; or 0, if none.	
'KEY?	(— <i>addr</i>)
Returns <i>true</i> if a key is ready to be read by KEY .	
'ACCEPT	(— <i>addr</i>)
Address of the task's ACCEPT routine.	
SPAN	(— <i>addr</i>)
Contains the number of characters read by the most recent ACCEPT .	
#TIB	(— <i>addr</i>)
Contains the actual number of characters available to interpret in the input stream.	

References **ACTIVATE**, Section 5.3.3
WAIT, Section 5.2
Terminal drivers, *Forth Programmer's Handbook*, Section 3.8

5.2.4 Sharing Resources

Some system resources must be shared by tasks without giving any single task permanent control of them. Disk units, tape units, printers, 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.

SwiftOS 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 multiplexor. 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 multiplexor while it obtains a value using the lower-level word (**A/D**). The phrase **MUX GET** waits in the **PAUSE** loop (see definition of **GET**, below) to obtain private access to this resource. The phrase **MUX RELEASE** releases it, without awakening another task.

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 in the multitasking loop until the facility is free or is owned by the task which is running **GET**. High-level code for **GET** would be:

```
: FREE ( a -- a t)    @ DUP 0=  SWAP
  STATUS =  OR ;
: GET ( a -- )    BEGIN  PAUSE  FREE UNTIL
  STATUS SWAP ! ;
```

GET checks a facility repeatedly until it is available. **GET** is written in code, and the overhead rarely exceeds two or three machine instructions. Maintaining a queue is almost always slower.

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 the facility is owned by another task, **RELEASE** does nothing. Using the definition of **FREE** above, a high-level definition of **RELEASE** would be:

```
      : RELEASE ( a -- )    FREE IF  0 SWAP !  ELSE
      DROP THEN ;
```

Note that **GET** and **RELEASE** can be used safely by any task at any time, as they don't let any task take a facility from another.

SwiftOS 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 them 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 multiplexor/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 **GETs**. However, a poorly written application might have the conflicting requests occur on different nesting levels, hiding the problems until a conflict occurs.



It is better to design an application to **GET** only one resource at a time—deadlocks are impossible in such a system.

Glossary

GET

(*addr* —)

Obtain control of the facility variable at *addr*, having first made one circuit of the SwiftOS round robin. If the facility is owned by another task, the task executing **GET** will wait until the facility is available.

GRAB

(*addr* —)

Obtain control of the facility variable at *addr*. If the facility is owned by another task, the task executing **GRAB** will wait until the facility is available. **GRAB** does not **PAUSE** before attempting to obtain control, so, in order to prevent deadlocks and avoid “resource hogging,” it should be used only in circumstances in which it is known that no other task could have the facility.

RELEASE

(*addr* —)

Relinquish the facility variable at *addr*. If the task executing **RELEASE** did not previously own the facility, this operation is a no-op.

5.3 BACKGROUND TASKS

Background tasks exhibit the general properties of all tasks in SwiftOS. In fact, terminal tasks are *supersets* of background tasks, possessing extended user areas to support the requirements for serial I/O.

Background tasks are suitable for virtually all chores that do *not* require serial I/O. Examples include many kinds of equipment control and data acquisition.

This section describes procedures for defining and managing background tasks.

5.3.1 Defining a Background Task

Background tasks have a data stack, a return stack, and a user area for variables whose values are not shared (the *user variables*, discussed in Section 5.2.3). Background tasks do not service a terminal or support a compiler.

BACKGROUND creates a *task definition table* in code space that is used, after target power-up, to build a background task’s user area and stacks in RAM (see Figure 16). **BACKGROUND** is a defining word that expects the sizes (in bytes) of the user area, data stack, and return stack. **BACKGROUND** does not link the task into the round robin or make the task run a program; that will be described in the next section.

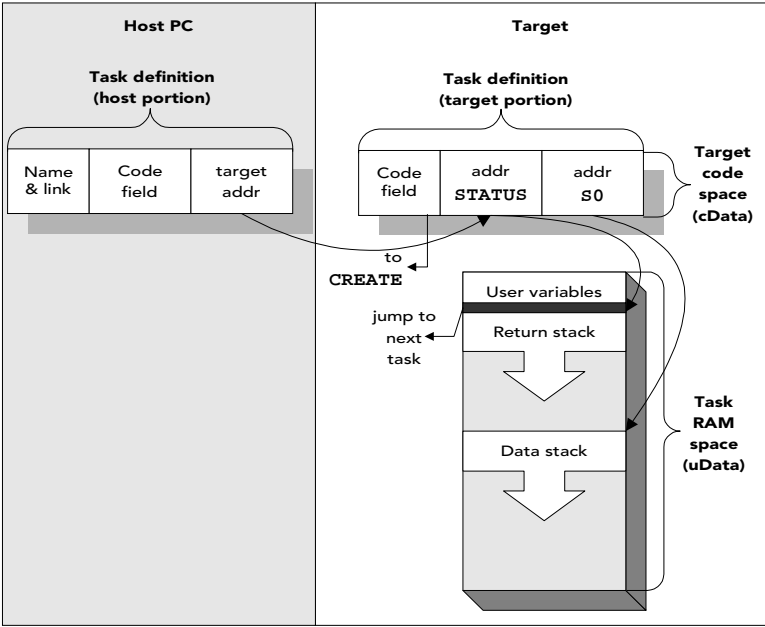


Figure 16. Memory allotted for a background task

An example of **BACKGROUND**'s use is:

```
32 128 64 BACKGROUND SCRIBE
```

This defines a task whose name is **SCRIBE**, which has 32 bytes of user area, 128 bytes of data stack, and 64 bytes of return stack. The total target size of **SCRIBE** in this example would be three cells (12 bytes on a 32-bit target, 6 bytes in a 16-bit target) in code space and 224 bytes of RAM. On a target with four bytes per cell, the 64-byte return stack allows the program to nest Forth words, loop parameters, etc., to a depth of 16 cells.

The smallest possible user area is the size of the region needed to keep the user variables **STATUS** through **CATCHER**, about five cells (see Section 5.2.3). The extra cells in **SCRIBE**'s user area could be used by **SCRIBE**'s program to keep a sample count, a virtual array address, and a device I/O address for a data logging application.



There must be sufficient space to store the registers **I** (or the program counter, in subroutine-threaded implementations) and **R** on the data stack, in addition to the maximum number of stack items that may be present when the task enters the multitasking loop (e.g., by executing **PAUSE**).

The run-time behavior of words created by **BACKGROUND** is to return on the stack the address of the task's definition table, the first cell of which contains a pointer to the first byte of the task's **STATUS**. Thus, the location of **SCRIBE**'s **STATUS** is found by the phrase:

SCRIBE @

Glossary

BACKGROUND <task-name> (*nu ns nr —*)
 Define the background task *task-name*—with *nu* bytes of user area, *ns* bytes of data stack and *nr* bytes of return stack—and set up its task definition table based on these parameters. Subsequent use of *task-name* returns the address of the table containing the parameters for building the task. The phrase <task-name> @ will return the address of the task's **STATUS** (the start of its user area).

5.3.2 Initializing a Background Task

All SwiftOS kernels contain, at power-up, a terminal task called **OPERATOR**. If no other task exists, **OPERATOR**'s jump address will contain the address of **OPERATOR**'s own **STATUS**. In this way, the round robin first consists of a single jump instruction, which jumps to itself.

After a new background task has been defined, the phrase:

<task-name> **BUILD**

is used to initialize *task-name*'s user area and to link it into the round robin. This usually is done as part of the power-up sequence.

In an earlier example, the background task named **SCRIBE** was defined:

32 128 64 BACKGROUND SCRIBE

To initialize the task and link it into the round robin, one would use:

SCRIBE BUILD

SCRIBE leaves the address of its task definition table on the stack. Then **BUILD** begins its work:

1. **BUILD** uses the address that is on the stack to set up the new task's stack space and user area in RAM.
2. **BUILD** copies the complete jump instruction, including the address of the next task, from the **STATUS** of **OPERATOR** into the **STATUS** of the new task.
3. **BUILD** stores the address of the new task's **STATUS** into **OPERATOR**'s jump address.

(Steps 2 and 3 enable the CPU to jump from **OPERATOR** to the new task, then to the task which formerly followed **OPERATOR** in the round robin.)

4. **BUILD** gets a copy of the value for the new task's **S0**—the bottom of its data stack, calculated by **BACKGROUND** and compiled just after the **STATUS** address—and stores it into the user variable **S0**.

At this point, although the task exists and is part of the round robin, it is not yet running a program. The words the task executes may be defined later.

Glossary

BUILD(*addr* —)

Initialize a task, given the address of the task's definition table which was constructed by **BACKGROUND**. The task will be linked in the round-robin after **OPERATOR**, and before the task previously linked to **OPERATOR**. This must be done at run time in the target before any attempt to **ACTIVATE** the task. Usage:

```
<task-name> BUILD
```

OPERATOR(— *addr*)

Return the address of the task definition table of the first task defined in any SwiftOS kernel. **OPERATOR** is a **TERMINAL** task. If no task has been defined, *addr* is **OPERATOR**'s own **STATUS**.

References

Defining a background task, Section 5.3.1
 Making a background task run a program, Section 5.3.3
 Terminal tasks, Section 5.4

5.3.3 Controlling a Background Task

After a task has been defined by **BACKGROUND** and linked into the round robin by **BUILD**, the new task is “asleep.” This is necessary because the task still has not been given the parameters that will determine what word(s) the task will execute.

The word that makes a task run a program is **ACTIVATE**, which must be used in a **:** definition. It expects on the stack the address of a task definition table, placed there by executing a task’s name. **ACTIVATE** clears the task’s data and return stacks, then awakens the task such that it will begin executing the word immediately following **ACTIVATE**. The words following **ACTIVATE** must end in **STOP** or in an endless loop.

The example below demonstrates everything needed to run a **BACKGROUND** task. Assume that a background task was defined and initialized by the following phrases (described in previous sections):

```
32 128 64 BACKGROUND SCRIBE
( power-up initialization)
SCRIBE BUILD
```

In our hypothetical application, the following definition will be used by a task to record data samples onto disk:

```
: RECORD ( addr -- )    ACTIVATE
    BEGIN COLLECT ARCHIVE PAUSE AGAIN ;
```

Then **RECORD** could be assigned to a task by the phrase:

```
SCRIBE RECORD
```

ACTIVATE uses the address returned by executing a task’s name (**SCRIBE**, in this example) to cause that task to execute the words following **ACTIVATE**. In the definition of **RECORD**,

- the words **COLLECT** and **ARCHIVE** perform the data logging;
- **PAUSE** gives any other tasks in the round robin an opportunity to execute; and
- the **BEGIN ... AGAIN** phrase fulfills the requirement that code assigned to a task never reach the **;** in the definition containing **ACTIVATE**.

Note that, in the above example, **BUILD** is kept separate from **RECORD** because a task can only be built once, but can be activated many times.

For convenience in the discussion below, a *slave task* is **ACTIVATED** by another task. A *master task* uses **ACTIVATE** to activate the slave task. When the phrase **SCRIBE RECORD** executes, **ACTIVATE** (in the definition of **RECORD**) will do the following:

- 1. Reset the slave task's (**SCRIBE**, in the example) stacks to an empty state. If the slave was previously active, it will forget what it was doing and its data and return stacks will be cleared.
- 2. Store the address of the next word to be executed as the only item on the slave's return stack. In this case, that is the address of **COLLECT**, since **BEGIN** only acts at compile time to set up the loop.
- 3. Push the slave's return stack pointer (with one item on it) onto the slave's data stack, and save the data stack pointer in the slave's user area until it begins executing.
- 4. Store a **WAKE** instruction into the slave's **STATUS** cell.

Because the master task's return stack was popped (in step 2, above), when it leaves **ACTIVATE** it will return *not* to the next word in **RECORD**, but to whatever called **RECORD**. In other words, the master task starts executing the word containing **ACTIVATE**, but goes no further than that; the slave task is the only one that executes the balance of the definition. This relationship is shown in Figure 17.

Master task executes this part	Slave task executes this part
: RECORD (addr --) ACTIVATE	BEGIN <words> AGAIN ;

Figure 17. Two tasks are involved in a word containing **ACTIVATE**



Code following an **ACTIVATE** must never reach the **;** because the **EXIT** or **RTS** that is compiled by a semicolon would attempt to pop an empty return stack. Therefore, the code following an **ACTIVATE** must end either in an endless loop (as in the example) or in the word **STOP**.

Here is another example of a control word, this time using the word **STOP**:

```
: NOD ( -- ) BEGIN STOP AGAIN ;
: HALT ( addr -- ) ACTIVATE NOD ;
```

Because **ACTIVATE** forcibly resets a task's execution environment, the definition of **HALT** in the example above will forcibly stop the task at *addr*, and start it executing an infinite loop in which, should it ever be awakened (e.g., by an interrupt), it will **STOP**. **HALT** is occasionally useful to provide a stable, inactive behavior. A **HALT**ed task may be **ACTIVATED** in order to perform some other function, when needed.

Note that **ACTIVATE** does not distinguish whether a task was previously assigned a function that it is performing. In fact, after a task is **ACTIVATED** the first time, it will never be without a function to perform, even if that function is **NOD**. Nor is the concept of "busy" relevant; the master task is running, or it couldn't be executing **ACTIVATE**! So, if you want to avoid interrupting a task until it has finished performing an assigned function, let the task set and clear a flag or variable to indicate when it is busy or free for re-assignment. Facility variables (described in Section 5.2.4) are convenient for this purpose.

Glossary

ACTIVATE	(<i>addr</i> —)
Start the task at <i>addr</i> executing part of the definition in which ACTIVATE appears, starting with the word following ACTIVATE . ACTIVATE may only be used inside a colon definition. The task executing the balance of the definition must be prevented from ever reaching the end of the definition (e.g., by STOP , NOD , or by being in an infinite loop that describes its desired behavior).	
NOD	(—)
Infinite loop designed to ensure that a task remains inactive until ACTIVATED to do something else.	
HALT	(<i>addr</i> —)
Cause the task at <i>addr</i> to perform NOD . Usage:	
<code><task-name> HALT</code>	

References **BEGIN**, in assembler, see target-specific SwiftX documentation
 Definition of **HALT** for a terminal task, Section 5.4.3
PAUSE, **STOP**, and **WAIT**, Section 5.2

5.4 TERMINAL TASKS

A terminal task is a background task with additional user variables that enable it to do serial I/O. Optionally, terminal tasks also may be given user variables to enable it to support an interpreter.

This section discusses issues peculiar to terminal tasks.

5.4.1 Defining a Terminal Task

The word **TERMINAL** allots memory to a terminal task and creates the task definition table for a task that may control a serial port. The return stacks of all terminal tasks in a single system have the same size, and space is allowed for input message buffers and private dictionaries. Because terminal tasks can have a private dictionary, if the target system has an interpreter, they may have user variable space for dictionary management, disk access, editing, and interpretation of text input (from disk or keyboard); see Table 21 on page 118 for related details.

TERMINAL is used as follows:

```
<n> TERMINAL <task-name>
```

where *n* is size in bytes to be allotted to the task's private dictionary (Section 7.1.2 discusses terminal tasks as they relate to the optional target-resident interpreter). The sizes of individual stacks, user area, and buffers belonging to a terminal task are configuration parameters, specified in **Swiftx\Src\<CPU>\Config.f**.

Besides allotting space, **TERMINAL** may also establish driver routines for the terminal task. The exact method depends on the serial interface hardware; see your hardware-specific SwiftX documentation for details.

A terminal task also has provision for device-specific words for various common functions performed by serial devices. These correspond to certain of the terminal Application Programming Interface (API) functions described in the *Forth Programmer's Handbook*, Section 3.8, which often require device-specific definition. For example, the control character sequence required for cursor positioning (**AT-XY**) varies from one terminal to another, and cursor positioning may or may not be supported on a particular printer. SwiftOS provides for these by including execution vectors for these functions in a terminal task's user area. Each function supported in this way has a vector and a standard name for a low-level, device- or task-specific version of its behavior. These are summarized in Table 19.

Table 19: Vectored terminal-specific words

API word	Vector	Device-specific word	Description
EMIT	' EMIT	(EMIT)	Display a character.
TYPE	' TYPE	(TYPE)	Display a string.
CR	' CR	(CR)	Go to next line (carriage-return).
PAGE	' PAGE	(PAGE)	Go to next page (form-feed).
AT-XY	' AT-XY	(AT-XY)	Go to specified column/row.
KEY	' KEY	(KEY)	Await one key-press.
KEY?	' KEY?	(KEY?)	Check whether keypress occurred.
ACCEPT	' ACCEPT	(ACCEPT)	Accept a string.

S0 will be copied to the task's user variable area by **CONSTRUCT**. Consult your platform documentation for the procedure for initializing the other vectors for your target system.

As with **BACKGROUND**, **TERMINAL** must be executed at compile time, because **TERMINAL** builds the ROM table used to construct the task in RAM after power-up in the target system.

Glossary

TERMINAL <task-name> (n —)
Define the terminal task *task-name* and set up its task definition table for a task

whose private dictionary is n bytes in size. (A non-zero value for n is only appropriate when using the interpreter option; see Section 7.) Subsequent use of *name* returns the address of the task definition table, which contains the parameters for building the task. The phrase <task-name> @ will return the address of the task's **STATUS** (the start of its user area).

References

CONSTRUCT, Section 5.4.2

User variables, Section 5.2.3

Terminal drivers, *Forth Programmer's Handbook*, Section 3.8

5.4.2 Initializing a Terminal Task

After a terminal task has been defined by the word **TERMINAL**, the terminal task must be initialized by the use of the word **CONSTRUCT**, and then be made to run a program by the use of **ACTIVATE**. **CONSTRUCT** is used for terminal tasks similarly to how **BUILD** is used for background tasks.

Terminal task initialization consists of linking the task into the round robin and setting the task's user variables. The only remaining initialization required before a task is running is to set up the task's stacks, which will be performed by **ACTIVATE** (described in Section 5.3.3).

The word **CONSTRUCT** must be executed after power-up in the target system. It performs four separate functions:

1. It links the new terminal task into the round robin (using **BUILD**).
2. It copies non-device-dependent initialization from **OPERATOR**'s user variables to the new task's user variables.
3. It copies device-dependent data from the task definition table compiled by **TERMINAL** into the user variable area of the new task. Typical device-dependent data includes **DEVICE** and vectored routines for the terminal-specific functions described in Table 19.
4. If the target has an interpreter, it sets the task's interpreter pointers.

After **CONSTRUCT** has been used, the task may be given a function to perform, as described in Section 5.4.3.

Glossary

CONSTRUCT(*addr* —)

Initialize a task, given the *addr* of its task definition table that was constructed by **TERMINAL**. The task will be linked in the round-robin after **OPERATOR** and before the task previously linked to **OPERATOR**. This must be done at run time in the target system, before any attempt to **ACTIVATE** the task. Usage:

```
<task-name> CONSTRUCT
```

References

Controlling a terminal task, Section 5.4.3

Defining a terminal task, Section 5.4.1

Configuring a terminal task for the interpreter, Section 7.1.2

5.4.3 Controlling a Terminal Task

After a terminal task has been **CONSTRUCTED**, it may be made to run a program. This is done by **ACTIVATE**, which controls **TERMINAL** tasks in the same way it controls **BACKGROUND** tasks.

The usual cautions about using **ACTIVATE** apply:

- **ACTIVATE** must be in a **:** definition.
- **ACTIVATE** takes a task address from the stack, and starts that task executing the words following **ACTIVATE**. The task must never reach the **;** of the definition containing **ACTIVATE**.
- The task that starts executing **ACTIVATE** exits from the definition containing **ACTIVATE** without executing any of the words following **ACTIVATE**. That part of the definition is executed by the task that is **ACTIVATED**. See the reference on **ACTIVATE**, Section 5.3.3, for more information.

References

ACTIVATE, Section 5.3.3**EMIT**, *Forth Programmer's Handbook***PAUSE**, **STOP**, and **WAIT**, Section 5.3

5.5 A MULTITASKING DEMO

To demonstrate the SwiftOS multitasker at work, you may run the Conical Pile Calculator demo (described in Section B.2) with a separate terminal task. To do this, you will need to:

1. Follow directions accompanying your SwiftX product to connect the serial output port on the demo board to a COM port on your computer that is not being used for your XTL link.
2. Launch a terminal emulator (such as Terminal or HyperTerminal) on your PC, and configure it for the COM port connected to your board.
3. Check that the following lines are included near the end of **Debug.f**:

```
INCLUDE ... \CONICAL
INCLUDE ... \DUMB
```

```
256 TERMINAL CONSOLE    CONSOLE CONSTRUCT
: DEMO    CONSOLE ACTIVATE DUMB SCI-TERMINAL CALCULATE ;
```

4. Launch SwiftX (if it is not already running) and initialize your target by using Project > Debug.
5. Run the demo by typing **DEMO** in the SwiftX command window. If you have positioned your terminal emulator window so both it and the SwiftX window are visible, you should see the application's prompt in the terminal emulator window.
6. In the terminal emulator window, type any parameters you wish for the demo, as described in Section 1.4.3.

Observe that your target system is responsive to demo program commands in the terminal emulator window, and to normal XTL commands in the SwiftX command window. In fact, you may run the demo from the command window, as well; but you may find that the application is not re-entrant (see the discussions in Section 5.1 and Section 5.2.3), due to its use of variables. If you are feeling ambitious, you might try defining those as user variables, and see if that helps!

5.6 COMPARING BACKGROUND TASKS AND TERMINAL TASKS

Terminal tasks and background tasks are architecturally similar. In fact, a terminal task is technically a superset of a background task. Terminal tasks differ in that they have larger user areas to accommodate user variables related to servicing terminals and other serial devices.

Table 20: Functions applied to background and terminal tasks

	Background tasks	Terminal tasks
Define	<nu> <ns> <nr> BACKGROUND <name>	<n> TERMINAL <name>
Initialize	<name> BUILD	<name> CONSTRUCT
Assign functions	<addr> ACTIVATE (<i>addr</i> returned by <i>name</i> ; must be inside a colon definition)	<addr> ACTIVATE (<i>addr</i> returned by <i>name</i> ; must be inside a colon definition)
Minimum user variables	STATUS through CATCHER	STATUS through DEVICE

6. TARGET LIBRARIES

A SwiftX system includes libraries for the target system, all supplied in source form. Depending upon the extent of a library, it may be found as an individual file or as a directory. This differs from the method of providing libraries as linkable objects, but it is generally more flexible, and is more consistent with Forth's normal practice of compiling directly from source to executable form without a link phase.

To add a library to your target, just include the relevant files in the main load file, **Kernel.f**, prior to the line that includes the startup file **Start.f**.

6.1 ENHANCED NUMBER CONVERSION

Number conversion is often very simple in embedded systems, consisting only of obtaining a key and subtracting the offset to ASCII 0:

```
      : DIGIT ( -- n ) KEY [CHAR] 0 - 0 MAX 9 MIN ;
```

Such a word could be put inside a loop, starting with a 0 “accumulator” on the stack and, for each digit, multiplying the accumulator by 10 (assuming decimal) before adding the next digit.

For more complex situations, SwiftX provides a more powerful set of number-conversion words. For example, you may need to accept numbers with:

- decimal points (dots or commas, depending on American or European usage)
- colons (used in angles and times)
- slashes (used in dates)
- dashes (used in part numbers, telephone numbers, etc.)

SwiftX's number-conversion words are based on the low-level number-conversion word from ANS Forth, **>NUMBER** (see the *Forth Programmer's Handbook*, section 1.1.6).

The word **NUMBER?** takes the address and length of a string, and attempts to convert it until 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 in the current base (or *radix*) in effect for the task. All numeric strings must end with a blank or expiration of the length. If another character is encountered (i.e., a character which is neither a digit in that base nor punctuation), conversion will end. There must be no spaces within the number, since a space is a delimiter. If a leading minus sign is present, it must immediately precede the leftmost digit or punctuation character.

Any of the following punctuation characters may appear in a number:

, . + - / :

All punctuation characters are functionally equivalent, causing the digits that follow to be counted. This count may be used later by certain conversion words. The punctuation character itself performs no function other than to set a flag indicating 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 single value zero.

- If the number is single-precision (i.e., not punctuated), 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 variable **DPL** is used to track punctuation during the conversion process. **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 indicates potentially useful information:

- If it is negative, the number was not punctuated and is single-precision.
- Zero or a positive non-zero value indicates the presence of a double-precision number, and equals 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 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 SwiftX, both in the host's interpreter that processes source text and in the target kernels. 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 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 (**DPL** remains negative), 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. This can be useful when dealing with naturally unpunctuated numbers whose values may exceed 65536 on a 16-bit system—for example, United States zip codes (e.g., 90266).

Glossary

>NUMBER

$$(ud_1 \text{ } c\text{-}addr_1 \text{ } u_1 \text{ --- } ud_2 \text{ } c\text{-}addr_2 \text{ } u_2)$$

Convert the characters in the string at $c\text{-}addr_1$, 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 any algebraic sign) is encountered or the string is entirely converted; the result is ud_2 . $c\text{-}addr_2$ is the location of the first unconverted character or, if the entire string was converted, of the first character beyond the string. u_2 is the number of unconverted characters in the string. "to-number"

NUMBER?

$$(c\text{-}addr \text{ } u \text{ --- } 0 \mid n \mid 1 \mid d \mid 2)$$

Attempt to convert the characters in the string at $c\text{-}addr$, whose length is u , into digits, using the radix in **BASE**, until the length expires. If valid punctuation is encountered (, . + - / :), 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

$$(c\text{-}addr \text{ } u \text{ --- } n \mid d)$$

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

DPL

$$(\text{--- } addr)$$

Returns 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 not punctuated. If it is non-negative, it represents the number of digits to the right of the rightmost punctuation character.

NH

$$(\text{--- } addr)$$

Returns 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*, Section 1.1.6
 Number conversions in SwiftX, Section 4.1.1

6.2 TIMING FUNCTIONS

The words in this section support a hardware time-of-day clock and calendar. On targets without a built-in clock, you may enter a date and time, and SwiftX will maintain it using an internal clock or programmable timer, if one is available. See your hardware-specific documentation for details. The accuracy with which this can be done depends, of course, upon the resolution of the available clock hardware.

The functions described in this section require the extended number-conversion facilities described in Section 6.1.

6.2.1 Date and Time of Day Functions

SwiftX 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.

SwiftX stores time information internally as an unsigned, double number representing seconds since midnight. There are 86,400 seconds in a day, so a double number is required for portability across 16- and 32-bit implementations.

Dates are represented internally as the number of days since January 1, 1900, also known as the *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 range of dates that can be converted accurately by this algorithm is from March 1 1900 through February 28, 2100. Both of these are not leap years and are not handled by this algorithm, which is good only for leap years that are divisible by four 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 80340 to **M/D/Y**, which will convert it to the MJD for August 3, 1940. This takes advantage of the enhanced SwiftX number-conversion facility that automatically processes punctuated numbers as double-precision (see Section 6.1).

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

On most target platforms, 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).

The alternative input form **D/M/Y** is also available (see “Date functions” on page 106).

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 as **mm/dd/yyyy**. The word **.DATE** will take an MJD and display it in that format.

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

@NOW (— *ud n*)
 Return the system time as an unsigned, double number *ud* representing seconds since midnight, and the system date as *n* days since 01/01/1900. Used (for example) by **TIME&DATE** (see the *Forth Programmer's Handbook*).

!NOW (*ud u* —)
 Take the same parameters as those returned by **@NOW** and set the system time and date.

!TIME&DATE (*u₁ u₂ u₃ u₄ u₅ u₆* —)
 Convert stack arguments *u₁* seconds (0–59), *u₂* minutes (0–59), *u₃* hours (0–23), *u₄* day (1–31), *u₅* month (1–12), *u₆* 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* — *c-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	$(u_1\ u_2\ u_3 \rightarrow u_4)$
Converts day u_1 , month u_2 , and year u_3 into MJD u_4 .	
M/D/Y	$(ud \rightarrow u)$
Accept an unsigned, double-number date which was entered as mm/dd/yyyy, and convert it to MJD.	
@DATE	$(\rightarrow u)$
Return the current system date as an MJD.	
(DATE)	$(u_1 \rightarrow c\text{-}addr\ u_2)$
Format the MJD u_1 as a string with the format mm/dd/yyyy, returning the address and length of the string.	
.DATE	$(u \rightarrow)$
Display the MJD u in the format applied by (DATE) above.	
DATE	(\rightarrow)
Display the current system date.	
NOW	$(ud \rightarrow)$
Set the current system date to the value represented by the unsigned, double number which was entered as mm/dd/yyyy.	

References Enhanced number conversion, Section 6.1

6.2.2 Interval Timing

SwiftX includes facilities to time events, for specifying when something will be done, and for 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** is dependent upon the target system's clock, the accuracy of the measured interval depends upon the resolution of that clock. As a general statement, the error on an interval will be approximately the number of milliseconds in one clock tick. If you need to respond more promptly to an external event, the best way is to associate an interrupt directly with the event.

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 ( -- )      \ Measure the measurement overhead
  COUNTER              \ Initial value
  100000 0 DO          \ Total time for 100,000 trials
    PAUSE              \ One loop 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 maximum clock error per iteration (in 1/100ths of a millisecond) may be calculated by:

```
1000 100 T/SEC */ <n> / .
```

where **T/SEC** is the number of clock interrupts per second (usually defined as an **EQU** in the **INTERPRETER** scope on the host, and used to compile the timing

definitions discussed here), and n is the number of iterations, as above.

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 low-order cell of the millisecond timer.

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

EXPIRED ($u - \text{flag}$)
 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 \ ...till timer expires
  DROP ;
```

References Time and timing functions, *Forth Programmer's Handbook*, Section 3.7
 Interrupt (exception) handlers, see your hardware-specific SwiftX documentation

6.2.3 Benchmarks

The file `\Swiftx\Src\Bench.f` contains a small suite of benchmarks, useful for comparing various targets. It is also a good example of how to time functions using the tools described in Section 6.2.2. You may wish to apply similar techniques for measuring time-critical application functions.

To run the benchmarks, load the file by typing (where `<cr>` represents the Enter key):


```

HOST <cr> ok
CD ..\...\ <cr> ok
INCLUDE BENCH <cr> ok

```

and run the benchmarks by typing **BENCH**. The output displayed is a list of words and the time each takes to execute on the target. Timing is only as accurate as the granularity of the target's millisecond counter.

The first of these benchmarks, **DO**, measures the **DO ... LOOP** overhead associated with the measurement process. The last, **PAUSE**, measures the time it takes for the multitasking loop; it is most relevant if you have several tasks running application functions. The other tests in **BENCH** are general measurements of arithmetic functions.

6.3 ARITHMETIC LIBRARY

SwiftX provides a variety of arithmetic operators for use in applications. They are designed to work efficiently on small microcomputer systems, trading extreme precision for size and speed. We recommend that you study the requirements of an application carefully; the provided routines have nearly always been found adequate when the limited precision of the associated I/O is properly taken into account. Most analog I/O has a precision of 12 bits or less; rarely does it exceed 16 bits. Numeric I/O, in real-time applications typical of SwiftX projects, rarely needs to be entered or displayed with any more precision than this.

6.3.1 Double-Precision Arithmetic

The standard double-precision operations (64 bit, in this case) defined in the *Forth Programmer's Handbook* have all been included in SwiftX as separate files, so they can be included or excluded, depending on the application. High-level words may be found in **SwiftX\Src\Double.f**, while code primitives may be found in **SwiftX\Src\<cpu>\Double.f**.

References Arithmetic and logical operators, *Forth Programmer's Handbook*

6.3.2 Fixed-Point Fractions

Fixed-point fractions are very simple and are widely applicable. In most cases, math using them executes much faster than math using floating-point numbers—hence their inherent desirability. As a basic starting point, this SwiftX includes 14-bit fractions. These are convenient, because they have sufficient headroom in 16 bits to exactly describe unity and, in fact, can almost reach two (handy for vector magnitudes, etc.); and they have sufficient precision for most applications.

For those who are unfamiliar with fractional arithmetic, a brief overview follows.

Fractional arithmetic involves scaling and an implied decimal point. Instead of scaling by multiples of ten, as we do with decimal numbers (digits 0–9), we scale by multiples of two, as you would expect with binary numbers (digits 0 and 1). The implied decimal point is actually a binary point. We need to establish some sort of convenient scale to represent the positive integer 1. For 14-bit fractions, we define:

16384 CONSTANT +1

As a 16-bit binary number, 16384 looks like this:

0100000000000000

The one bit is scaled up in binary, along with the binary point, with 14 bits to its right to represent the fraction. A number scaled in this way has a precision of one part in 16384. If the number is used to represent a fraction of a circle (see below), the angular resolution is about 0.022 degrees,

Addition and subtraction of fractional numbers is done with the usual **+** and **-** operators, but some way of multiplying and dividing is needed. The words ***.** and **/.** perform this work. To assure ourselves that these operators really work, we can divide 1 by 1, and get 1 (16384).

In order to display the results of fractional arithmetic operations in a meaningful form, the word **.F** is included. It displays a fraction as a real number with four decimal places.

Readers who wish to learn more about fractional arithmetic should read the

book *Starting Forth*, which offers an excellent discussion of rational numbers and fractional arithmetic.

Glossary

+1	$(-n)$ Return the value 1.00000000000000 (binary) in 14-bit fraction notation. There is an implied binary point just after the second-most-significant bit.
*.	$(nf - n)$ Multiply two numbers, one of which is a fraction, to return a whole number.
/.	$(n_1 n_2 - f)$ Divide n_1 by n_2 and return a fractional result. The divisor must be positive.
.F	$(f -)$ Display a fraction with four decimal places.

References Fixed-point fractions, *Starting Forth*

6.3.3 Angles

Angles are hard to work with in conventional programming environments, for two major reasons: they usually are expressed in radians, and they are realized as floating-point numbers. This can be costly in both code and time, because many practical applications (such as encoders) work in revolutions, not radians; representation in radians wastes a fractional bit, and the arithmetic to convert back to revolutions adds time and computational noise.

The SwiftX approach is to represent angles in terms of revolutions; specifically, as fixed-point fractions of a revolution. For the usual 14-bit fractions, 360 degrees is represented internally as 16384, 45 degrees as 2048, and so forth; the least-significant bit is equal to approximately 0.022 degrees. In addition to gaining substantial convenience, the other benefits of representing angles in this way are legion; among them are extreme simplification of range reductions, maintenance of maximum precision and accuracy regardless of the manipulation involved, scale-independent error functions, and much faster execution. Thus, overall execution speed can be significantly improved in

end-user applications or library functions that use large numbers of angular functions (e.g., Fast Fourier Transform). In those rare formulae (such as Kepler's equation) where the angle must be in radians, it's usually worthwhile to scale into radians, as necessary, but to keep most of the work in revolutions.

References Fixed-point fractions, Section 6.3.2

6.3.4 Transcendentals

The most often needed transcendental routines are square root, trigonometric, and log/exponential, usually in that order. With the exception of square root, we usually employ Chebyshev polynomial approximations and consider Hart, et al., to be the definitive reference work. In the interest of providing reasonable standards, most of our routines apply the highest-precision polynomial that fits within the precision of the data type. The nice thing about using Hart for a source is that you can easily adjust the precision of the algorithm to match the required accuracy of the calculation, often with a substantial reduction of execution time. (Remember that few analog input or output devices exceed 12 or 16 bits of precision.)

References Hart, John F. et al., *Computer Approximations*, Krieger Publishing Co., Inc., P.O. Box 9542, Melbourne, FL 32902-9542, (305) 724-9542.

6.3.4.1 Square Root

The basic square root algorithm has many uses and, with suitable prescaling, may be used for any fixed-point data when a root accurate to 15 bits is sufficient.

6.3.4.2 Trigonometric Functions

A full set of 14-bit fractional trigonometric functions, along with some support words, is supplied, as well as a test file that loads all the required arithmetic and which contains some useful examples.

As discussed in Section 6.3.3, angles are expressed as fractions of a circle, scaled by 16384. That is, a value of 8192 is equivalent to 180 degrees. The out-

put fractional value is scaled to 16384 in the same way. For example, the **SIN** of 2048 (45 degrees) returns 11585 (0.707).

All trig functions that can go to infinity represent their potentially infinite values as ratios. Only in rational form are such values manipulable and meaningful; more to the point, the most common uses of these functions involve direction cosines for rotations. By using this representation, we affirm the purposes of the functions and maintain the utility of their results. Reduction of a ratio to a fraction, where needed, is done by /.

<i>Glossary</i>		
SIN	Sine	(<i>n</i> — <i>f</i>)
COS	Cosine	(<i>n</i> — <i>f</i>)
TAN	Tangent	(<i>n</i> — <i>f</i> ₁ <i>f</i> ₂)
COT	Cotangent	(<i>n</i> — <i>f</i> ₁ <i>f</i> ₂)
CSC	Cosecant	(<i>n</i> — <i>f</i> ₁ <i>f</i> ₂)
SEC	Secant	(<i>n</i> — <i>f</i> ₁ <i>f</i> ₂)
ASIN	Arc-sine	(<i>f</i> — <i>n</i>)
ACOS	Arc-cosine	(<i>f</i> — <i>n</i>)
ATAN	Arc-tangent of the ratio <i>n</i> ₁ / <i>n</i> ₂ , giving <i>n</i> ₃ .	(<i>n</i> ₁ <i>n</i> ₂ — <i>n</i> ₃)

DEG ($n - f$)
Converts an angle in degrees into a fraction representing part of a circle.
For example, 45 degrees is 1/8 of a circle. If you type:

45 DEG .

you should get 2048.

REV ($f - n$)
Converts a fractional angle into an integer with two implied decimal places.

<.5 ($f_1 - f_2$)
Converts an angle from a fraction in the range 0–360 degrees to the range
-180 – +180 degrees.

7. INTERPRETER/COMPILER OPTION

SwiftX offers a target-resident interpreter and high-level Forth compiler as an option. The purpose of this option is to provide Forth services via a serial line to a terminal or terminal emulator from the target. This facility supports:

- Field service debugging without a full SwiftX host system
- Simple command-line user interfaces
- Ability to change configuration parameters easily

This facility is far less powerful than the full SwiftX development environment. It supports programming only to the extent of adding simple high-level Forth definitions; no target-resident assembler is provided. For complex programming or debugging chores, the SwiftX host remains the environment of choice.

If your SwiftX system doesn't include this option and you would like to add it, please contact your FORTH, Inc. sales representative.

7.1 CONFIGURING SWIFTX FOR THE INTERPRETER OPTION

In order for your target program to use the interpreter option, there are some basic requirements:

- Some or all of your target definitions must have heads in order to be found from the terminal. This is described in Section 7.1.1 below.
- The terminal task communicating with your serial port must be configured for additional user variables and sufficient memory for a private dictionary. This is described in Section 7.1.2 below.

7.1.1 Configuring the Target Dictionary

SwiftX normally compiles only executable code in its target program space, making an extremely compact system. In order to use the interpreter option, however, at least some definitions must have heads in order to make them accessible. There are two ways to configure your SwiftX cross-compiler for this, depending upon your needs:

1. Put heads on all words, but retaining the ability to specify certain words to be without heads.
2. Omit heads except for the selected words you wish to make available.

Option 1 is appropriate if you wish most words in your target to be available for debugging or added programs, and if you have sufficient memory. Option 2 is available if you need to keep your program small, or wish to make only certain words available for user interface purposes.

Managing these options involves two words to set defaults, and two to override the current default for a single definition:

- **+HEADS** configures the cross-compiler to put heads on all words; **|** (vertical bar, pronounced “bar”) causes the next target definition to be compiled without a head.
- **-HEADS** configures the cross-compiler to compile definitions without heads; **~** (tilde) causes the next target definition to be compiled with a head.

Both the defaults and the overrides affect all SwiftX defining words operating in the **TARGET** scope, constructed with **:** (colon), **CODE**, **CREATE**, **VARIABLE**, etc. It will *not* affect **EQU** definitions, because they compile target literals rather than actual definitions.

You may change your default at any time. Thus, you may have certain files compiled with the **+HEADS** setting and others with the **-HEADS** setting.

An override symbol must precede the defining word you wish it to affect. (Since these are words themselves, they must be delimited by spaces.) For example, if you are compiling with the **-HEADS** default set and wish to put a head on a word that displays a user menu, you might use:

```
~ : MENU . . .
```


Similarly, if you are compiling with the **+HEADS** default and wish to specify no head on a word containing a password, you might use:

3728493 | CONSTANT PASS

or

| 3728493 CONSTANT PASS

Remember, only words compiled with heads may be accessed from the target’s interpreter, either by users or by added definitions. All words remain available in the cross-compiler’s **TARGET** scope.

Glossary

+HEADS	(—)
Set the cross-compiler to put heads on all subsequent target definitions.	
-HEADS	(—)
Set the cross-compiler to omit heads on all subsequent target definitions.	
~	(—)
Cause the next target definition to have a head, regardless of the current default.	
 	(—)
Cause the next target definition to have no head, regardless of the current default.	

References

TARGET scope, Section 4.6.2
Defining words in SwiftX, Section 4.4

7.1.2 Configuring a Terminal Task for the Interpreter

The interpreter and compiler are accessed through a terminal task attached to the serial port through which interactivity will take place. All terminal tasks have the ability to support serial-type I/O and, potentially, an interpreter and a private dictionary. (Basic principles of defining and managing terminal tasks

are described in Section 5.4.)

A terminal task is defined using the form:

```
<n> TERMINAL <name>
```

where *n* is the size of its private dictionary. The dictionary size may be zero for tasks with no private dictionary (such tasks may still use the interpreter, but may not compile anything). The total amount of space used by the task is the sum of *n* plus all the parameters listed in Table 21. The default sizes for these parameters on your system may be found in `\Src\<CPU>\Config.f`.

Table 21: Parameters governing the space allocation for a terminal task

Name	Description	Typical size (bytes)
U	Maximum size of user area	256
S	Maximum size of data stack	256
R	Maximum size of return stack	256
NUM	Space reserved for output number conversions	66
PAD	Space reserved for PAD (scratch string storage)	84
TIB	Terminal input buffer (TIB)	80

The task space will be allocated out of `uData` when the target program is compiled. The target-resident compiler doesn't have different memory sections.

The locations of the user area, return stack, and data stack are permanent, as is the location of the start of the private dictionary (**H0**). However, **PAD** "floats" above the current top of the dictionary (the location returned by **HERE**), at a distance given by `|NUM|`. Therefore, if you put data at **PAD** and then extend your dictionary (by adding definitions), **PAD** will have moved and that data will no longer be accessible.

A map of the various spaces in a terminal task area is shown in Figure 18. The arrows show the direction of growth of the various resources. Since various of these resources are configured to grow towards each other, they effectively share a larger pool of space than is allocated for each individually. For example, if the return stack is small (as it tends to be), a string longer than 80 bytes may be accepted into the terminal input buffer. This greatly improves the

overall use of space and system reliability. However, if you know your application will regularly require more space for any particular resource, you should adjust the default allocations.

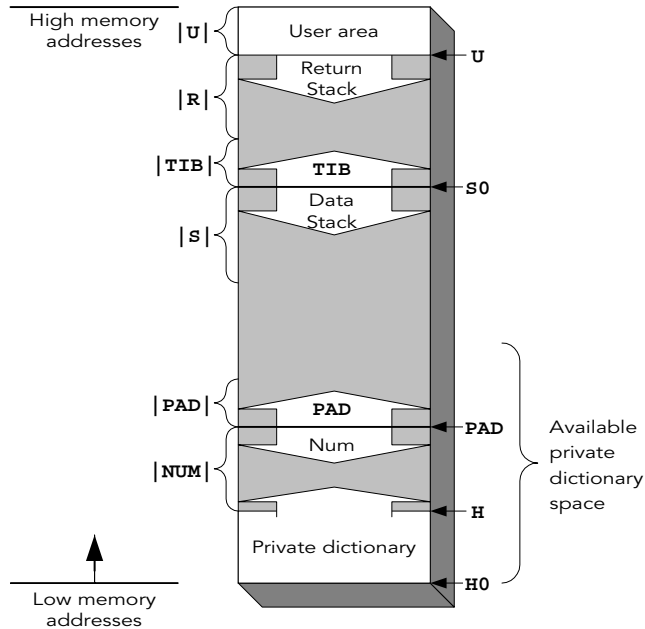


Figure 18. Terminal task use of memory

The terminal task that is to run the interpreter should be initialized as part of the start-up code in your target program. You will need to **CONSTRUCT** this task, as described in Section 5.4.2. You must then connect it to its serial port, as described in your CPU-specific documentation. Finally, if you wish its behavior to be that of a normal Forth system (accepting and interpreting user input), you must assign this behavior to it by using a definition such as:

```
: LISTEN ( -- )    <task-name> ACTIVATE QUIT ;
```

References **ACTIVATE**, Section 5.4.3
 QUIT, Section 7.2
 Terminal tasks, Section 5.4

7.1.3 Additional User Variables

A number of user variables are added to the SwiftX target in order to support the interpreter. These are described in the following glossary, along with some related words.

<hr/> <i>Glossary</i> <hr/>		
BASE		(— <i>addr</i>)
	Contains the current radix used for input and output number conversions.	
H		(— <i>addr</i>)
	Contains the address of the next available location in the user dictionary.	
HERE		(— <i>addr</i>)
	Returns the address of the next available location in the user dictionary (equivalent to the H @ sequence).	
H0		(— <i>addr</i>)
	Contains the address of the start of the user dictionary.	
LAST		(— <i>addr</i>)
	Contains the address of the start of the most recent definition.	
STATE		(— <i>addr</i>)
	Contains zero if the task is in interpret mode, non-zero if it's compiling (i.e., inside a colon definition).	
TIB		(— <i>addr</i>)
	Returns the address of the start of the terminal input buffer.	
>IN		(— <i>n</i>)
	Returns the offset from the start of the TIB indicating the next bytes to be interpreted.	

<hr/> <i>References</i> <hr/>	# TIB , Section 5.2.3
-------------------------------	------------------------------

7.2 USING THE INTERPRETER

The target-resident interpreter works as described in the *Forth Programmer's Handbook*, Sections 1.1.5 and 2.6. It is case-sensitive, so you should give careful thought to use of case in your word names.

A number of the low-level components of the interpreter are available, and may be of use on occasion. These are summarized in the glossary below.

INTERPRET uses **WORD** to parse the current input stream in a loop, and uses **FIND** to search the dictionary for it. It will compile or execute each word, depending on whether it is inside a colon definition, as indicated by a non-zero value in **STATE**. A flowchart of this process is provided in the *Forth Programmer's Handbook*, Figure 5. This continues until the input stream is exhausted.

This input stream is normally received by **ACCEPT** into the terminal input buffer. Its parameters (address and length) are returned by **SOURCE**. The high-level word that does this, including setting parameters as necessary, is **QUERY**.

Glossary

QUERY	(— <i>n</i>)
Accepts a line of text into TIB and leaves its actual length in #TIB .	
SOURCE	(— <i>c-addr n</i>)
Returns the address and length of the input buffer.	
INTERPRET	(—)
Attempts to execute, or convert to a number, each word found in the current input buffer. Throws -4 if the stack underflowed as a result of executing a word.	
PARSE	(<i>char</i> — <i>c-addr n</i>)
Parses the current input stream (whose parameters are given by SOURCE) looking for the first occurrence of <i>char</i> , or the end of the input stream if it is not found. Returns the address and length of the found string. The length is zero if the string cannot be parsed.	
WORD	(<i>char</i> — <i>c-addr</i>)
Parses the input stream (using PARSE) looking for the delimiter <i>char</i> . Returns the address of a resulting counted string (length in its first byte).	

FIND*(c-addr — c-addr 0 | xt 1 | xt -1)*

Searches the dictionary for the counted string at *c-addr*. If a match is found, returns its *xt* and 1 if the word is **IMMEDIATE** (bit 7 is set), or its *xt* and -1 if the word is not **IMMEDIATE**. Returns *c-addr* and *false* if the word is not found.

References Dictionary searches, *Forth Programmer's Handbook*, Section 2.6.3
IMMEDIATE words, *Forth Programmer's Handbook*, Section 2.10

7.3 USING THE COMPILER

The target-resident compiler is intended only for relatively simple programming, such as simple definitions used for troubleshooting or for adding minor program features. It is optimized for small size and portability, rather than power and speed. The principle programming support for SwiftX remains the host cross-compiler and XTL.

As a result, the following features are *not* supported:

- Assembler
- Word lists, or scopes
- Ability to add new data types or compiler directives
- Local storage for program source
- Other advanced programming features

Source may be transmitted to the target using a terminal emulator to send a text file. The target can interpret such source a line at a time. It will acknowledge each line as a line typed by a user, by saying "ok" followed by a CR. Therefore, you can pace transmission by configuring your terminal emulator to wait for a CR before sending each line.

The target-resident compiler can handle references to all words that were cross-compiled with heads, as described in Section 7.1.1. You may type **WORDS** to see a list of available words.

All compiling actions that increases the size of the dictionary will check the remaining space, and will abort if there is not enough for the sum of the stack size, **PAD**, and the number-conversion buffer.

APPENDIX A: BIBLIOGRAPHY

- American National Standard for Information Systems: Programming Language Forth* (ANSI X3.215–1994). American National Standards Institute, 11 W. 42nd St., New York, NY 10036, (212) 642-4900.
- Bailey, G., Sanderson, D., Rather, E. “clusterFORTH, A High-Level Network Protocol,” *Proceedings of the 1984 FORTH Conference*. Rochester, NY: The Institute for Applied Forth Research, 1984.
- Brodie, L. *Starting Forth*, Englewood Cliffs, NJ: Prentice-Hall, 1981, 2nd ed. 1987. Contact: Forth Interest Group, 100 Dolores St., Suite 183, Carmel, California 93923.
- Brodie, L. *Thinking Forth*, Englewood Cliffs, NJ: Prentice-Hall, 1984. Reprinted 1994 by the Forth Interest Group, 100 Dolores St., Suite 183, Carmel, California 93923.
- Kelly, M.G., and Spies, N. *Forth: A Text and Reference*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
- Koopman, P. *Stack Computers, The New Wave*. Chichester, West Sussex, England. Ellis Horwood, Ltd. 1989.
- Martin, T. *A Bibliography of Forth References*, 3rd ed. Rochester, NY: Institute for Applied Forth Research, 1987.
- Moore, C.W. “The Evolution of Forth — An Unusual Language,” *Byte*, August 1980.
- Noble, J.V. *Scientific Forth*. Charlottesville, VA: Mechum Banks Publishing, 1992.
- Pountain, R. *Object Oriented Forth*. New York: Academic Press, 1987.
- Rather, E.D. “Forth Programming Language” *Encyclopedia of Physical Science & Technology* (V. 5) Academic Press, Inc., 1987, 1992.
- Rather, E.D. “Fifteen Programmers, 400 Computers, 36,000 Sensors and Forth,” *Journal of Forth Application and Research* (V. 3, #2, 1985), P.O. Box 27686, Rochester, NY 14627.
- Rather, E.D., Colburn, D.R., and Moore, C.W. “The Evolution of Forth,” *ACM SIGPLAN Notices*,

Vol. 28, No. 3, March 1993.

Rather, E.D. and Conklin, E.K. *Forth Programmer's Handbook*. ISBN 0-9662156-0-5. FORTH, Inc. 111 N. Sepulveda Blvd., Suite 300, Manhattan Beach, CA 90266-6847.

Terry, J.D. *Library of Forth Routines and Utilities*. New York: Shadow Lawn Press, 1986.

Tracy, M. and Anderson, A. *Mastering Forth* (2nd ed.). New York: Brady Books, 1989.

APPENDIX B: PROGRAMMING EXAMPLES

This section presents some simple application examples to illustrate the process of writing and testing SwiftX programs. Since this book is designed to be independent of the particular board or processor you're using, these examples will necessarily omit detailed memory mapping and hardware considerations; refer to your platform-specific documentation for these. However, these examples will illustrate some important principles:

- Factor your definitions into very short, preferably re-usable, components.
- Give your definitions brief but meaningful names to improve your code readability.
- Design “top-down.”
- Test “bottom-up” using the intrinsic interactivity of Forth.

B.1 THE WASHING MACHINE

This is a simple example that illustrates some basic principles of application organization and design. We will also describe how you would actually go about developing and testing this simple program.

There are two basic aspects to this program: the control of the I/O (consisting of six digital switches that control various components of the washing machine plus the standard system timer), and the logic of the washing machine's behavior. These follow directly from the basic design.

This example represents a minimal kind of embedded system. It can run on a stripped kernel only a few hundred bytes in size, although it would run equally well (just take more space) on a full kernel as shipped with SwiftX.

The complete source for this example is given in Section B.1.3

B.1.1 Top-down Program Design

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

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

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

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

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

The code in this example (given in Section B.1.3) is nearly self-documenting; the few comments show the parameters being passed to certain words and identify groups of words. Forth allows as many comments as desired, with no penalty in object code size or performance.

When reading,

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

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

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

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

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

Reading further, one finds that **FAUCET** is simply a constant which returns the bit mask for the port that controls the I/O, while **ON** is a simple word that turns on the selected bit.

Even from this simple example, it may be clear that Forth is not so much a language, as a tool for building application-oriented command sets. The definition of **WASHER** is based not on low-level Forth words, but on washing-machine words like **SPIN** and **RINSE**. By “factoring” the program into tiny components like this, you make your program not only more readable, but also easier to test. And the components are much more likely to be reusable: here, we use words like **FILL-TUB**, **AGITATE**, and **DRAIN** several times, and the **ON** and **OFF** commands frequently.

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

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

B.1.2 Hardware Control

This example assumes six digital switches, which are represented as individual bits in an 8-bit control register. For simplicity, we assume this control register is memory mapped to a port address that we’ll show here as 01. Where such a port would be in actual address space depends upon your CPU and other engineering and design decisions beyond the scope of this manual. Timing is done using the standard SwiftX word **MS** (see Section 6.2.2), which is defined in terms of your system clock; see your platform-specific documentation for details.

From a software perspective, a memory-mapped port is simplest, because you can read and write it using high-level fetch and store operators.

If your hardware is available, your first step would be to see if it works. Even without writing a formal driver or application code, you can read and write the hardware registers by typing phrases such as:

```
HEX 01 C@ .
```

This would read the port and display its value in hex, so you can see the individual bits. You could do things to the hardware that would cause the bit values to change (e.g., press a button or throw a switch) and see if the bit on the port does change appropriately. You could write to the port, for example, by typing:

```
1 01 C!
```

and see if the motor turns on. In this way, you can test that your hardware is connected and functioning before you even write a line of code.

Program access to the I/O is provided by naming the port, and then defining various bit masks that can be used to isolate individual switches on the port:

```
\ Port assignments
01 CONSTANT PORT
```

\ bit-mask	name	bit-mask	name
1	CONSTANT MOTOR	8	CONSTANT FAUCET
2	CONSTANT CLUTCH	16	CONSTANT DETERGENT
4	CONSTANT PUMP	32	CONSTANT LEVEL

If the hardware is unavailable, you might temporarily re-define **PORT** as a variable, so you can read and write it, and thus test the rest of the logic.

These definitions would be put at the beginning of your source file. Naming the port and device bits in this way provides *information hiding*, which means, in this case, that if the port or bit assignments change, all you have to do is modify these definitions, and the rest of your code will run unchanged.

You can load your source file by using the command **INCLUDE** <filename>, whereupon all its definitions are available for testing. You can further exercise your I/O by typing phrases such as:

```
MOTOR ON or MOTOR OFF
```

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

```
DETERGENT ADD
```

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

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

B.1.3 Code for the Washing Machine

```
( Washing Machine Application )
\ Port assignments
01 CONSTANT PORT

\ bit-mask      name      bit-mask      name
1 CONSTANT MOTOR      8 CONSTANT FAUCET
2 CONSTANT CLUTCH     16 CONSTANT DETERGENT
4 CONSTANT PUMP       32 CONSTANT LEVEL

\ Device control
: ON ( mask -- ) PORT C@ OR PORT C! ;
: OFF ( mask -- ) INVERT PORT C@ AND PORT C! ;

\ Timing functions
: SECONDS ( n -- ) 0 ?DO 1000 MS LOOP ;
: MINUTES ( n -- ) 60 * SECONDS ;

: TILL-FULL ( -- ) \ Wait till level switch is on
  BEGIN PORT C@ LEVEL AND UNTIL ;

\ Washing machine functions
: ADD ( mask -- ) DUP ON 10 SECONDS OFF ;
: DRAIN ( -- ) PUMP ON 3 MINUTES ;
: AGITATE ( -- ) MOTOR ON 10 MINUTES MOTOR OFF ;
: SPIN ( -- ) CLUTCH ON MOTOR ON 5 MINUTES
  MOTOR OFF CLUTCH OFF PUMP OFF ;
```

```
: FILL-TUB ( -- ) FAUCET ON TILL-FULL FAUCET OFF ;

\ Wash cycles
: WASH ( -- ) FILL-TUB DETERGENT ADD AGITATE DRAIN ;
: RINSE ( -- ) FILL-TUB AGITATE DRAIN ;

\ Top-level control
: WASHER ( -- ) WASH SPIN RINSE SPIN ;
```

B.2 THE CONICAL PILE CALCULATOR

SwiftX includes an example of a complete (though small) application based on an example in the popular Forth tutorial book *Starting Forth* by Leo Brodie. The following description is adapted from the 2nd edition, p. 302 ff.

The SwiftX source for this program is in the file **SwiftX\Src\Conical.f**. Instructions for running this demo application using the Cross-Target Link may be found in Section 1.4.3. Instructions for running it on a separate serial port using a second multiprogrammed task are given in Section 5.5.

B.2.1 Algorithm Description

This example is a math problem that many people would assume could be solved only by using floating point. It will illustrate how to handle a fairly complicated equation with fixed-point arithmetic, and will demonstrate that for all the advantages of using fixed-point, range and precision need not suffer.

In this example, we will compute the weight of a cone-shaped pile of material, knowing the height of the pile, the angle of the slope of the pile, and the density of the material.

To make the example more “concrete,” let’s weigh several huge piles of sand, gravel, and cement. The slope of each pile, called the *angle of repose*, depends on the type of material. For example, sand piles more steeply than gravel.

(In reality, these values vary widely, depending on many factors; we have chosen approximate angles and densities for purposes of illustration.)

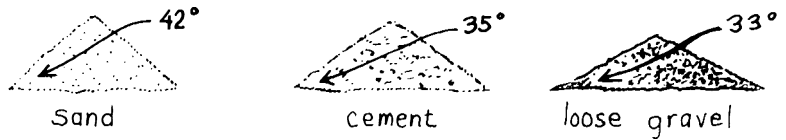


Figure 19. Angles of repose

Here is the formula for computing the weight of a conical pile h feet tall with an angle of repose of Θ degrees, where D is the density of the material in pounds per cubic foot:*

$$W = \frac{\pi h^3 D}{3 \tan^2(\Theta)}$$

This will be the formula that we must express in Forth.

Let's design our application so that we can enter parameters to specify the material, then the height of the pile, using single letters to identify the parameter. This will be easy to support with a terminal emulator or the XTL, and in a real device, the single letters could easily map to special function keys. So, we might enter 1M to select the first material, 10F to specify 10 feet, and 6I for six inches. You can enter these selections repeatedly, and then enter = to perform the calculations and get a result.

Let's assume that for any one type of material, the density and angle of repose never vary. We can store both of these values, for each type of material, into a table. Since we ultimately need each angle's tangent, rather than the number of

* Derivation: The volume V of a cone is given by:

$$V = 1/3(\pi b^2 h)$$

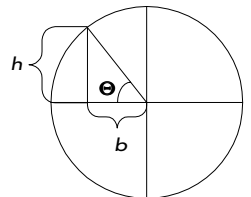
where b is the radius of the base and h is the height. We can compute the base by knowing the angle or, more specifically, the tangent of the angle. The tangent of an angle is simply the ratio of the segment marked h to the segment marked b in the drawing. If we call this angle Θ ("theta"), then:

$$\tan \Theta = h/b$$

Thus, we can compute the radius of the base with:

$$b = h/\tan \Theta$$

When we substitute this into the expression for V and then multiply the result by the density D in pounds per cubic foot, we get the foregoing formula.



degrees, we will store the tangent. For instance, the angle of repose for a pile of cement is 35° , for which the tangent is .700. We will store this as the integer 700.

Bear in mind that our goal is not just to get an answer; we are programming a computer or device to get the answer for us in the fastest, most efficient, and most accurate way possible. To write equations using fixed-point arithmetic requires an extra amount of thought. But the effort pays off in two ways:

1. Vastly improved run-time speed, which can be very important when there are millions of steps involved in a single calculation, or when we must perform thousands of calculations every minute. Also,
2. Program size, which would be critical if, for instance, we wanted to put this application in a hand-held device specifically designed as a pile-measuring calculator. Forth is often used in this type of instrument, for this very reason.

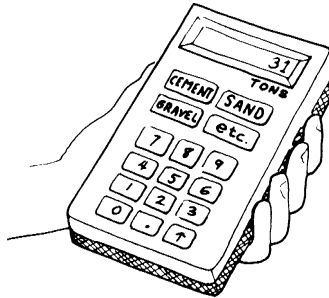


Figure 20. Conical pile calculator

Let's approach our problem by first considering scale. The height of our piles ranges from 5 to 50 feet. By working out our equation for a pile of cement 50 feet high, we find that the weight will be nearly 35,000,000 pounds.

However, because our piles will not be shaped as perfect cones, and because our values are averages, we cannot expect better than four or five decimal places of accuracy.* If we scale our result to tons, we get about 17,500. This value will comfortably fit within the range of a 16-bit number. For this reason, let's write this application entirely with single-length arithmetic operators.

* For Math Experts: In fact, since our height will be expressed in three digits, we can't expect greater than three-digit precision. For purposes of our example, however, we'll keep better than four-digit precision.

Applications that require greater accuracy can be written using double-length arithmetic. But we intend to show the accuracy that Forth can achieve even with 16-bit math.

By running another test with a pile 40 feet high, we find that a difference of one-tenth of a foot in height can make a difference of 85 tons in weight! So we decide to scale our input to tenths of a foot, not just whole feet.

We'd like the user to be able to enter:

```
15F    2I    =
```

where the function keys **F** and **I** will cause the execution of words **FOOT** and **INCH**, respectively, to convert the feet and inches into tenths of an inch, and **=** will execute the word **PILE** to do the calculation. Here's how we might define **FOOT** and **INCH**:

```
\ Convert n1 feet to n2 tenths of an inch.
: FOOT ( n1 -- n2 )    10 * ;

\ Convert n2 inches to tenths and add n1.
: INCH ( n1 n2 -- n3 )  100 12 */ 5 + 10 / + ;
```

(The use of **INCH** is optional.) Thus, **23 FOOT** will put the number 230 on the stack; **15 FOOT 4 INCH** will put 153 on the stack, and so on. We could as easily have designed input to be in tenths of an inch, with a decimal point, like this:

```
15.2
```

In such a case, the number-conversion routine would convert the input as a double-length value. Since we are only doing single-precision arithmetic, **PILE** could simply begin with **DROP**, to eliminate the high-order part.

In writing the definition of **PILE**, we must try to maintain the maximum number of places of precision without overflowing 15 bits. According to the formula on page 131, the first thing we must do is cube the argument. But let's remember that we will have an argument that may be as high as 50 feet, which will be 500 as a scaled integer. Even to *square* 500 produces 250,000,

* In fact, the definitions in the file that ships with SwiftX are slightly more complicated, in order to provide error checking and some user feedback.

which exceeds the capacity of a single-precision number on a 16-bit system.

We might reason that, sooner or later in this calculation, we're going to have to divide by 2000 to yield an answer in tons. Thus, the phrase:

```
DUP DUP 2000 */
```

will square the argument and convert it to tons at the same time, taking advantage of `*/`'s double-length intermediate result. Using 500 as our test argument, the above phrase will yield 125.

However, our pile may be as small as 5 feet, which when squared is only 25. To divide by 2000 would produce a zero in integer arithmetic, which suggests that we are scaling down too much.

To retain maximum accuracy, we should scale down no more than necessary. 250,000 can be safely accommodated by dividing by 10. Thus we will begin our definition of **PILE** with the phrase:

```
DUP DUP 10 */
```

The integer result at this stage will be scaled to one place to the right of the decimal point (25000 for 2500.0).

Now we must cube the argument. Once again, straight multiplication will produce a double-length result, so we must use `*/` to scale down. We find that by using 1000 as our division, we can stay just within single-length range. Our result at this stage will be scaled to one place to the left of the decimal point (12500 for 125000), and is still accurate to five digits.

According to our formula, we must multiply our argument by π . We know that we can do this in Forth with the phrase:

```
355 113 */
```

But we must also divide our argument by 3; we can do both at once with the phrase:

```
355 339 */
```

which causes no problems with scaling.

Next, we must divide our argument by the tangent squared, which we can do by dividing the argument by the tangent twice. Because our tangent is scaled to 3 decimal places, to divide by the tangent we multiply by 1000 and divide by the table value. Thus, we will use the phrase:

```
1000 THETA @ */
```

Since we must perform this twice, let's make it a definition, called **/TAN** (for "divide by the tangent") and use the word **/TAN** twice in our definition of **PILE**. Our result at this point will still be scaled to one place to the left of the decimal (26711 for 267110, using our maximum test values).

All that remains is to multiply by the density of the material, of which the highest is 131 pounds per cubic foot. To avoid overflowing, let's try scaling down by two decimal places with the phrase:

```
DENSITY @ 100 */
```

By testing, we find that the result at this point for a 50-foot pile of cement will be 34,991, which just exceeds the 15-bit limit. Now is a good time to take the 2000 (pounds per ton) into account. Instead of:

```
DENSITY @ 100 */
```

we can say:

```
DENSITY @ 200 */
```

and our answer will now be scaled to whole tons.

By using double-precision arithmetic on a 16-bit system, or by using a 32-bit system and scaling differently, we are able to compute the weight of the pile to the nearest whole pound. The range of 32-bit arithmetic compares with that of most floating-point arithmetic, but is much slower on processors that lack floating-point hardware or firmware.

The following is a comparison of the results obtained using a 10-decimal-digit calculator, 16-bit Forth and 32-bit Forth. The test assumes a 50-foot pile of cement, using the table values.

Table 22: Comparison of results using 16-bit and 32-bit arithmetic

	In Pounds	In Tons
Calculator	34,995,634	17,497,817
Forth, 16-bit	n/a	17,495
Forth, 32-bit	34,995,634	17,497,817

B.2.2 Material Selection

This application supports six different materials: cement, loose gravel, packed gravel, dry sand, wet sand, and clay. Each material has a characteristic density and theta, and a string to identify it for the user. This is an excellent example of an occasion where Forth's ability to define special data types* comes in handy. The defining word **MATERIAL** (see the file `SwiftX\Src\Conical.f`) can be used to define each material:

```
\ Density Theta
  131   700 MATERIAL CEMENT      "Cement"
    93   649 MATERIAL LOOSE-GRAVEL "Loose gravel"
  100   700 MATERIAL PACKED-GRAVEL "Packed gravel"
    90   754 MATERIAL DRY-SAND    "Dry sand"
  118   900 MATERIAL WET-SAND     "Wet sand"
  120   727 MATERIAL CLAY        "Clay"
```

Each of the words defined by **MATERIAL** (**CEMENT**, etc.) will, when executed, store its density and theta in the variables **DENSITY** and **THETA** used for the calculation, and will put the address of the descriptive string in the variable **MATTER**.

Next, a table is constructed listing the *execution tokens* of these materials:

```
\ List of xt's of material words.
CREATE TABLE ' CEMENT , ' LOOSE-GRAVEL , ' PACKED-GRAVEL ,
  ' DRY-SAND , ' WET-SAND , ' CLAY ,
```

A numeric parameter in the range 0–5 can be used to index into this table, and

* See the *Forth Programmer's Handbook*, Section 2.7, and this manual, Section 4.7.

then **EXECUTE** the selected *xt*. This is done by the word **SELECT**:

```
\ Select material n, execute its xt in TABLE above.
: SELECT ( n)   DUP 6 < IF
    CELLS TABLE + @EXECUTE  .SUBSTANCE
    ELSE  .ERROR  THEN ;
```

SELECT is even used by **.DIRECTIONS** to display the menu.

The highest level of the program is a loop called **CALCULATE**. It processes the input. When feet and inches are entered, that value is accumulated in the variable **ACCM**, which is then given as a parameter to **PILE**.

```
: CALCULATE ( -- )           \ CALCULATE is the main program entry point.
0 ACCM ! .DIRECTIONS CEMENT \ Initialize, display prompt.
BEGIN #INPUT CASE           \ Get number followed by a letter.
  [CHAR] M OF SELECT ENDOF   \ M selects material
  [CHAR] F OF FOOT ACCM ! ENDOF \ F and I set feet and inches.
  [CHAR] I OF ACCM @ SWAP INCH ACCM ! ENDOF
  NIP [CHAR] = OF ACCM @ PILE 0 ACCM ! ENDOF \ = does the calculation.
  $1B ( Esc) OF EXIT ENDOF    \ Esc exits.
ENDCASE AGAIN ;
```

We recommend that you read the full source for this application in the file **SwiftX\Src\Conical.f**, as it provides a good example of how a program may be constructed.

References Execution tokens, *Forth Programmer's Handbook*, Section 2.5.6

B.2.3 Testing the Calculator

As with the Washing Machine example (Section B.1), we develop and test the program incrementally.

First, we would test the computational words by storing various values in the variables **DENSITY**, **THETA**, and **ACCM**, and by executing **PILE** with various heights. Remember that if you're connected to your target, you can fetch and store its variables from the keyboard, by typing phrases such as:

```
131 DENSITY ! or DENSITY @ . (which would return 131.)
```

Next, we would construct the list of materials, and test the defining word **MATERIAL** by, for example, typing **1 SELECT** and then examining the values in the variables to verify that the correct values were stored and that the behavior assigned to the new words by **MATERIAL** is what you intended.

You can look at the user directions by typing **.DIRECTIONS**. You can adjust the message, spacing, etc., as much as you like until you get a pleasing display.

Finally, when you are confident that the low-level components of your program are working, you may type **CALCULATE**. If one of its components fails to work, you can exercise it individually from the keyboard. For example, if you seem to be getting the wrong material, you might type:

```
1 SELECT  MATTER COUNT TYPE
```

and see if you get the string Loose gravel. Or, if you're getting wrong answers, you might try:

```
1 SELECT  10 FOOT PILE
```

and compare the result you get with results from a pocket calculator. Check the values stores in **DENSITY** and **THETA** to be sure they were set properly. If they aren't, your problem may be with **SELECT**; if they are, check your computational definitions again by executing their components.

This style of informal, interactive testing usually gets results much more quickly than steppers, breakpoints, etc.

APPENDIX C: INDEX TO FORTH WORDS

This section provides Table 24, 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.

The stack-argument notation is described in Table 23. Where several arguments are of the same type, and clarity demands that they be distinguished, numeric subscripts are used.

Table 23: Notation used for data types of stack arguments

Notation	Description
<i>a-addr</i>	A cell-wide byte address that is cell-aligned (i.e., the address is evenly divisible by the cell size in bytes).
<i>c-addr</i>	A cell-wide byte address that is character-aligned (because a character is always one byte on current systems, this amounts to an arbitrary byte address).
<i>addr</i>	A cell-wide byte address.
<i>b</i>	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.
<i>c</i>	An ASCII character, stored as a byte (see above) with the parity bit reset to zero.
<i>d</i>	A double-precision, signed, 2's complement integer, stored as two stack entries (least-significant cell underneath the most-significant cell). On 16-bit machines, the range is from -2^{31} through $+2^{31}-1$. On 32-bit machines, the range is from -2^{63} through $+2^{63}-1$.
<i>flag</i>	A single-precision Boolean truth flag (zero means <i>false</i> , non-zero means <i>true</i>).
<i>i*x, j*x, etc.</i>	Zero or more cells of unspecified data type.

Table 23: Notation used for data types of stack arguments (*continued*)

Notation	Description
<i>n</i>	A signed, single-precision, 2's complement number. On 16-bit machines, the range is from -2^{15} through $+2^{15}-1$. 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 <i>n</i> , <i>u</i> 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.
<i>+n</i>	A single-precision, unsigned number with the same positive range as <i>n</i> above. An input stack argument shown as <i>+n</i> must not be negative.
<i>u</i>	A single-precision, unsigned number with a range from 0 to $2^{16}-1$ on 16-bit machines, or 0 through $2^{32}-1$ on 32-bit machines.
<i>ud</i>	A double-precision, unsigned integer with a range from 0 to 2^{32} on 16-bit machines, or 0 through $2^{64}-1$ on 32-bit machines.
<i>x</i>	A cell (single stack item), otherwise unspecified
<i>xt</i>	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 24: Index to Forth words

Name	Stack	Page
,	(<i>x</i> —)	45
+1	(— <i>n</i>)	111
!	(<i>x addr</i> —)	49
!C	(<i>x addr</i> —)	49
!NOW	(<i>ud u</i> —)	105
!TIME&DATE	(<i>u₁ u₂ u₃ u₄ u₅ u₆</i> —)	105
#TIB	(— <i>addr</i>)	83
#USER	(— <i>n</i>)	81
'ACCEPT	(— <i>addr</i>)	83
'ATXY	(— <i>addr</i>)	83
'CLEAN	(— <i>addr</i>)	83

Table 24: Index to Forth words (continued)

Name	Stack	Page
'CR	(— <i>addr</i>)	82
'EMIT	(— <i>addr</i>)	82
'KEY	(— <i>addr</i>)	83
'KEY?	(— <i>addr</i>)	83
'PAGE	(— <i>addr</i>)	83
'TYPE	(— <i>addr</i>)	82
(DATE)	(u_1 — <i>c-addr</i> u_2)	106
(TIME)	(<i>ud</i> — <i>c-addr</i> u)	105
*.	(<i>n f</i> — n)	111
+HEADS	(—)	117
+USER	($n_1 n_2$ — n_3)	81
,C	(x —)	45
-HEADS	(—)	117
.DATE	(u —)	106
.F	(f —)	111
.TIME	(<i>ud</i> —)	105
/.	($n_1 n_2$ — f)	111
2CONSTANT <name>	($x_1 x_2$ —)	47
2TWIN <name>	($x_1 x_2$ —)	61
: <name>	(—)	51
;	(—)	51
;CODE	(—)	51
<.5	(f_1 — f_2)	114
>IN	(— n)	120
>NUMBER	(ud_1 <i>c-addr</i> ₁ u_1 — ud_2 <i>c-addr</i> ₂ u_2)	102
@	(<i>addr</i> — x)	48
@C	(<i>addr</i> — x)	49
@DATE	(— u)	106

Table 24: Index to Forth words (continued)

Name	Stack	Page
@NOW	(— <i>ud n</i>)	105
@TIME	(— <i>ud</i>)	105
[+ASSEMBLER]	(—)	55
[+HOST]	(—)	55
[+INTERPRETER]	(—)	55
[+TARGET]	(—)	55
[PREVIOUS]	(—)	55
\\	(—)	33
{	(—)	33
	(—)	117
~	(—)	117
ACOS	(<i>f</i> — <i>n</i>)	113
ACTIVATE	(<i>addr</i> —)	92
ALIGN	(—)	45
ALIGNED	(<i>addr</i> — <i>a-addr</i>)	45
ALLOT	(<i>n</i> —)	44
ASIN	(<i>f</i> — <i>n</i>)	113
ASSEMBLER	(—)	54
ATAN	(<i>n</i> ₁ <i>n</i> ₂ — <i>n</i> ₃)	113
BACKGROUND <task-name>	(<i>nu ns nr</i> —)	88
BASE	(— <i>addr</i>)	82, 120
BLANK	(<i>c-addr len</i> —)	50
BUFFER: <name>	(<i>n</i> —)	47
BUILD	(<i>addr</i> —)	89
C!	(<i>b addr</i> —)	49
C!C	(<i>b addr</i> —)	49
C#	(— <i>addr</i>)	83
C,	(<i>b</i> —)	45
C,C	(<i>b</i> —)	45

Table 24: Index to Forth words (continued)

Name	Stack	Page
C@	(<i>addr</i> — <i>b</i>)	48
C@C	(<i>addr</i> — <i>b</i>)	49
CATCHER	(— <i>addr</i>)	82
CDATA	(—)	44
CODE <name>	(—)	51
COMPILER	(—)	54
CONNECT	(—)	65
CONSTANT <name>	(<i>x</i> —)	47
CONSTRUCT	(<i>addr</i> —)	96
COS	(<i>n</i> — <i>f</i>)	113
COT	(<i>n</i> — <i>f</i> ₁ <i>f</i> ₂)	113
COUNTER	(— <i>u</i>)	108
CREATE <name>	(—)	47
CSC	(<i>n</i> — <i>f</i> ₁ <i>f</i> ₂)	113
CVARIABLE <name>	(—)	47
D/M/Y	(<i>u</i> ₁ <i>u</i> ₂ <i>u</i> ₃ — <i>u</i> ₄)	106
DASM	(<i>addr</i> —)	26
DATE	(—)	106
DEG	(<i>n</i> — <i>f</i>)	114
DEVICE	(— <i>addr</i>)	82
DISCONNECT	(—)	65
DOES>	(—)	51
DOWNLOAD-ALL	(—)	65
DPL	(— <i>addr</i>)	102
DUMP	(<i>addr</i> <i>u</i> —)	26
DUMPC	(<i>addr</i> <i>u</i> —)	26
EDIT <text>	(—)	24
END-CODE	(—)	51
EQU <name>	(<i>x</i> —)	46

Table 24: Index to Forth words (continued)

Name	Stack	Page
ERASE	(<i>c-addr len</i> —)	50
EXPIRED	(<i>u</i> — <i>flag</i>)	108
FILL	(<i>c-addr len b</i> —)	50
FIND	(<i>c-addr</i> — <i>c-addr 0</i> <i>xt 1</i> <i>xt -1</i>)	122
FOLLOWER	(— <i>addr</i>)	82
GAP	(<i>n</i> —)	45
GET	(<i>addr</i> —)	85
GRAB	(<i>addr</i> —)	86
H	(— <i>addr</i>)	120
H0	(— <i>addr</i>)	120
HALT	(<i>addr</i> —)	92
HERE	(— <i>addr</i>)	44, 120
HIS	(<i>addr₁ n</i> — <i>addr₂</i>)	81
HOST	(—)	54
HOURS	(<i>ud</i> —)	105
IDATA	(—)	44
INCLUDE <filename>[<.ext>]	(—)	32
INTERPRET	(—)	121
INTERPRETER	(—)	54
L	(—)	24
L#	(— <i>addr</i>)	83
LABEL <name>	(—)	51
LAST	(— <i>addr</i>)	120
LOCATE <name>	(—)	24
M/D/Y	(<i>ud</i> — <i>u</i>)	106
MOVE	(<i>c-addr₁ c-addr₂ len</i> —)	50
MS	(<i>n</i> —)	108
NH	(— <i>addr</i>)	102

Table 24: Index to Forth words (continued)

Name	Stack	Page
NOD	(—)	92
NOW	(<i>ud</i> —)	106
NUMBER	(<i>c-addr u</i> — <i>n</i> <i>d</i>)	102
NUMBER?	(<i>c-addr u</i> — 0 <i>n</i> 1 <i>d</i> 2)	102
OPERATOR	(— <i>addr</i>)	89
ORG	(<i>addr</i> —)	44
PARSE	(<i>char</i> — <i>c-addr n</i>)	121
PAUSE	(—)	76
QUERY	(— <i>n</i>)	121
RELEASE	(<i>addr</i> —)	86
RESERVE	(<i>n</i> — <i>addr</i>)	47
RESTORE-SECTIONS	(<i>n*x n</i> —)	44
REV	(<i>f</i> — <i>n</i>)	114
S" <text>"	(— <i>c-addr len</i>)	50
S0	(— <i>addr</i>)	82
SAVE-CODE <name.ext>	(—)	62
SAVE-DATA <name.ext>	(—)	62
SAVE-SECTIONS	(— <i>n*x n</i>)	44
SEC	(<i>n</i> — <i>f</i> ₁ <i>f</i> ₂)	113
SEE <name>	(—)	26
SET-DOWNLOAD	(—)	65
SIN	(<i>n</i> — <i>f</i>)	113
SLEEP	(— <i>x</i>)	76
SOURCE	(— <i>c-addr n</i>)	121
SPAN	(— <i>addr</i>)	83
SSAVE	(— <i>addr</i>)	82
STATE	(— <i>addr</i>)	120
STATUS	(— <i>addr</i>)	82
STOP	(—)	76

Table 24: Index to Forth words (continued)

Name	Stack	Page
SYNC-CORE	(—)	65
TAN	($n - f_1 f_2$)	113
TARGET	(—)	55
TERMINAL <task-name>	($n -$)	94
THERE	(— <i>addr</i>)	45
TIB	(— <i>addr</i>)	120
TIME	(—)	105
TIMER	($u -$)	108
TO <name>	($x -$)	47
TOP	(— <i>addr</i>)	83
TWIN <name>	($x -$)	61
UDATA	(—)	44
VALUE <name>	($x -$)	47
VARIABLE <name>	(—)	47
W!	(x <i>addr</i> —)	49
W!C	(x <i>addr</i> —)	49
W,	($x -$)	45
W,C	($x -$)	45
W@	(<i>addr</i> — x)	48
W@C	(<i>addr</i> — x)	49
WAIT	(—)	76
WAKE	(— x)	76
WH <name>	(—)	25
WHERE <name>	(—)	25
WORD	(<i>char</i> — <i>c-addr</i>)	121
WVARIABLE <name>	(—)	47

GENERAL INDEX

Also see *Appendix C: Index to Forth Words* starting on page 139.

14-bit fractions 110

- A**
 - abort 16
 - align
 - an address 45
 - section allocation pointer 45
 - angles 111
 - App.f 37
 - arithmetic precision 109
 - assembler 11
 - documentation 21
 - assembly language
 - decompiled 26

- B**
 - background tasks 86
 - BASE**
 - host vs. target 40
 - base (See number conversion)
 - benchmarks 108–109
 - BUILD** 21
 - bye
 - menu equivalent 16

- C**
 - “Can’t locate keyboard definition” 14
 - case sensitivity 18
 - CD**, usage restrictions 32
 - cData 44
 - and object files 62
 - code
 - find a word
 - in compiled 25
 - in source 14, 22–24

- scroll source file 15
- code space
 - compiling values in 46
- colors 18, 19
- command history 28
- command window 12, 28
 - history 13
 - history of user commands 14
 - keyboard controls 15
 - print contents 16
 - record history 20
 - save contents 16, 28
 - select font 18
 - status bar 14
- comments
 - multi-line 33
- compile
 - for PROM 21
 - for testing 21
- compiler 10
 - control of file loading 31
 - error recovery 24
 - principles 50
 - search orders (See scopes)
 - target-resident interpreter option 115, 122
- conditional compilation 40
- configuration 21
 - of interface 18
 - save 20
- CREATE ... DOES>**, instances of 57
 - section type 58

Cross-Target Link (XTL) 11, 62

D **DASM** 26

data objects

 custom 57–61

 in iData 57

 in uData 57

day of the week, from MJD 104

DEBUG 21

debug tools 22

 disassembler/decompiler 25

L 24

LOCATE 22

 memory dump 26

defining words 46–47, 51, 59

 and section type 58

 custom

 example 60

demo program 6

 how to run 6

 multitasking version 97

dictionary

 in target (with interpreter option) 116–117

 private (with interpreter option) 117

directory paths (See path names)

disassembler/decompiler 25

DOES> in cross-compiling 58

double-precision number 100

double-precision operations 109

drag and drop 19

E editor

 select and configure 18

error handling (See exceptions)

exceptions

 manual abort 16

F facility variables 84

File menu 16–17, 28

filenames

 extension 31

 determines object format 61–62

files

 batch downloading 64

 loading 16, 31–32, 34

INCLUDE vs. menu options 32

 use with interpreter option 122

find a word

 in compiled code 25

 in source 22–24

fixed-point fractions 110

font 18

Forth glossary 21

FORTH, Inc. xii

H Help menu 21

hex format object file 61

history 28

 command window 13

 of user commands 14

I iData 44

 and object files 62

INCLUDE 8

 menu equivalent 16

 vs. menu/toolbar 31

initialization, target 35–37

input stream

 interpreting 121

installation instructions 3

Intel HEX format 62

interactive development 29

interpreter option

 case sensitive 121

 requirements for use 115

L **L** 24

LABEL 26

library

 adding to target 99

- LOCATE** 22, 26
- log
 - entire session 16
 - session activity 28
 - typed commands 16
- LOGGING** flag 32

- M**
 - memory (See also sections)
 - access 48–49
 - and target connection 63
 - configuring 41
 - and interpreter option 119
 - dump commands 26
 - overview 10, 41–42
 - menu
 - File 16–17, 28
 - Help 21
 - Options 18
 - Project 21
 - Tools 20
 - “Mismatch” 64
 - modified Julian date (MJD) 103
 - Motorola S19 format 62
 - multitasker (see round-robin algorithm)

- N**
 - “No target” 5
 - “Non-existent file” 14
 - NUMBER** 101
 - number conversion 39–40, 99
 - BASE** affected by **INCLUDE** 32
 - NUMBER?** 100
 - numbers
 - compiling vs. interpreting 39
 - negative 40

- O**
 - object file formats 61
 - Options menu 18

- P**
 - path names
 - absolute vs. relative 31–32
 - and including files 31
 - spaces not allowed 32
 - PAUSE**, detailed analysis 75
 - polynomial approximations 112
 - POWER-UP** 36
 - print 16
 - printer support 16
 - project 21
 - Project menu 21
 - PROM
 - build object file 63
 - create object file 61
 - punctuation
 - in numbers 100

- R**
 - re-entrancy, in Forth 72
 - resource sharing 84
 - round-robin algorithm 73
 - RUN** 20

- S**
 - “Save Command Window” 28
 - “Save Keyboard History” 28
 - scopes 52–57
 - change in a definition 53
 - list words in 20
 - overview 10
 - the default 52
 - search order (See scopes)
 - sections
 - allocate space 44
 - align 45
 - and defining words 58
 - and target-resident compiler 118
 - default type 58
 - display contents of 26
 - in memory 42–43
 - return next location in 44
 - save and restore context 44
 - specify next location in 44

SEE 25

“Session Log” 28

single-precision number 100

source code (See *also* file)

 filenames 31

 loading 31–32

square root 112

Start.f 37

STATUS 82

status bar 14

strings 49

STRIP 21

Strip 36

SwiftOS

 background tasks 86

 objectives 73

 resource sharing 84

 scheduling algorithm 73

 terminal tasks 93

 user variables 77

T terminal tasks 93

 private dictionary 118

 user area (with interpreter option) 118

 with interpreter option 117–120

toolbar 14

 button appearance 20

Tools menu 20

trigonometric functions 112

twins 60–61

U uData 44

 access requires connection 48, 63

user variables 77

 added for interpreter option 120

W **WAKE**, used by **PAUSE** 75

WH 25

WHERE 25

wordlists

 not supported in interpreter

 option 122

WORDS 20

words

 in current scope 20

X XTL 11, 20, 29, 62–69

 activate 21

 interactive vs. batch mode 64

 protocol 67

 status affects dumps 26