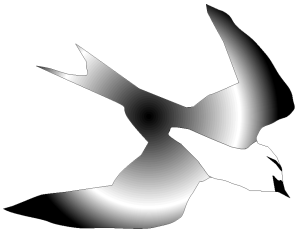

Getting Results Faster...

SwiftX/PSC1000

for Patriot Scientific PSC1000 Targets

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

First edition, January 1998
Printed 10/2/98

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

List of Figures

List of Tables

Welcome!

Important Information in This Book

This book is designed to accompany all SwiftX PSC1000 systems. It includes important information to help you connect the accompanying target board to your host PC. Failure to read and follow the instructions and recommendations in this book can cause you to experience frustration and, possibly, a damaged board. Since we want your experience with SwiftX to be a happy one, we urge you to make it easy on yourself. Read before plugging!

Scope of This Book

This book covers hardware-specific information about the setup and use of the target board supplied with your SwiftX system, and discusses hardware-related details of SwiftX development. It contains one appendix for each board-level product supported by SwiftX for the PSC1000; refer to the section for the board you will be using.

This book does not contain general user instructions about SwiftX and the Forth programming language, nor does it provide comprehensive information about the Patriot Scientific PSC1000. Refer to Patriot's documentation for information about the PSC1000, to the *SwiftX Reference Manual* to learn about the SwiftX Cross-Development System, and to the *Forth Programmer's Handbook* to learn about Forth.

Audience

This manual is intended for engineers developing software for processors in embedded systems. It assumes general familiarity with board-level issues such as power supplies and connectors.

How To Proceed

Begin with "Getting Started" on page 1. It will guide you through the process of connecting the target board supplied with this system and installing the software.

After you have installed and tested SwiftX on the PC and on the target board, all SwiftX functions will be available to you. That is a good time to refer to Section 1 of your *SwiftX Reference Manual* to learn how to operate your SwiftX system, including the demo application.

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 "road map" to help you get off to a good start with your SwiftX development system.

Three major steps are required to install SwiftX and begin using it:

1. *Connect the target board* provided with this system to your PC. To do so, follow the instructions given in the appendix for your board (see Table 1 below).

Table 1: Boards documented in this manual

Board	Connection instructions	Page
PSC1000 Evaluation Card II	Appendix A: PSC1000 Evaluation Card II Configuration and Use	35

We strongly advise you to set up and use the test board supplied with this system until you are familiar with SwiftX, even if your application's actual target hardware will be different. Once you understand the interactive development environment, you will find it much easier to compile and install the software on your specialized hardware.

2. *Install the software* by following the instructions in the *SwiftX Reference Manual*, Section 1.3. The installation procedure creates a SwiftX program group on Windows' Start > Programs menu, from which you may launch the main program by selecting the icon for your target board.

   The other icons in this program group provide access to documentation files and to an uninstall utility.

3. *Run the demo application* included with the system, following the instructions in the *SwiftX Reference Manual*, Section 1.4.3. For any board-specific issues

involved with running the demo, such as using a different serial port, refer to the appendix in this book that corresponds to your board.

General procedures for developing and testing software with SwiftX are provided in the *SwiftX Reference Manual*, Section 1.4.1.

2. THE PSC1000 ASSEMBLERS

The PSC1000 ("Sh-Boom") is a highly integrated 32-bit RISC processor that offers high performance at low system cost for a wide range of embedded applications. At 100 MHz internal clock rate, the processor delivers 100 MIPS peak performance. The 32-bit registers and data paths fully support 32-bit addresses and data types. The processor addresses up to 4 Gb of physical memory, and supports virtual memory with the use of external mapping logic.

Throughout this book, we assume you understand the hardware and functional characteristics of the PSC1000 microcontroller as described in Patriot Scientific's *PSC1000 Reference Manual*. We also assume you are familiar with the basic principles of Forth.

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

Where **boldface** type is used here, it distinguishes Forth words (such as register names) from Patriot Scientific names. Usually these are the same; for example, the name **push** can be used as a Forth word and as a Patriot Scientific mnemonic.

2.1 SWIFTX ASSEMBLER PRINCIPLES

Assembly routines are used to implement the Forth kernel, to perform direct I/O operations when desired, and to optimize the performance of interrupt handlers and other time-critical functions.

The SwiftX PSC1000 cross-compiler provides two assemblers: one for the Patriot Scientific PSC1000 microcontroller, and one for its associated I/O pro-

cessor, or IOP; the commands **MPU** and **IOP** select the appropriate versions. These must be used carefully, as some mnemonics are defined in both assemblers. By convention, **MPU** is the default; if you select **IOP**, you are responsible for re-asserting **MPU** when you are finished. **CODE** and **:** automatically select **MPU**. **IOP** routines must be defined using **LABEL** (see Section 2.2). In the status bar of SwiftX's command window, the panel second from the right shows the current assembler mode (MPU or IOP).

The mnemonics for the PSC1000 MPU and IOP opcodes have been defined as Forth words which, when executed, assemble the corresponding opcode at the next location in code space.

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

Some of the manufacturer's mnemonics have been replaced by different names, plus options to describe operations. The principal use of this strategy is in connection with conditional branches. SwiftX constructs conditional branches by using a condition code specifier followed by **IF**, **UNTIL**, or **WHILE**. Table 4 on page 12 summarizes the relationship between SwiftX condition codes used with one of these words and the corresponding Patriot Scientific mnemonic.

In accordance with Patriot Scientific's conventions, assembler mnemonics, addressing mode specifiers, and related commands appear in lower case in SwiftX. ANS Forth words and most SwiftX commands appear in upper case. SwiftX is normally case-insensitive; however, we recommend that you maintain these conventions consistently in order to avoid confusion.

References Assemblers in Forth, *Forth Programmer's Handbook*, Sections 1.3 and 4.0
Case-sensitivity option in SwiftX, *SwiftX Reference Manual*, Section 2.3.4.

2.2 CODE DEFINITIONS

Code definitions normally have the following syntax:

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

For example:

```
CODE DEPTH ( -- n)   \ Return current stack depth
   -2 # push   scache   pop   sdepth
   S0 U)   [] ld   sa push   sub
   1 # shr   1 # shr   add   ret   END-CODE
```

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

As an alternative to the normal **ret**, whose behavior is to execute the next word, the phrase:

```
WAIT br
```

may be used before **END-CODE** to terminate a routine by an unconditional jump through the SwiftOS multitasking executive, leaving the current task idle and passing control to the next task.

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

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

This creates a definition that returns the address of the next code space location, in effect naming it. You may use such a location as the destination of a branch or call, for example. The code fragments used as interrupt handlers are constructed in this way, and the named locations are then passed to **INTERRUPT**, which connects the code address to a specified interrupt vector.

The critical distinctions between **LABEL** and **CODE** are:

- If you reference a **CODE** definition inside a colon definition, the cross-compiler will assemble a call to it; if you invoke it interpretively while connected to a target, it will be executed.
- Reference to a **LABEL** under any circumstance will return the *address* of the labeled code.
- **CODE** routines are for the MPU only; **LABEL** routines may be either for the MPU or IOP.

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

Glossary

CODE <name> *(—)*
Start a new assembler definition, *name*. If the definition is referenced inside a target colon definition, it will be called; if the definition is referenced interpretively while connected to a target, it will be executed.

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

WAIT *(— addr)*
Return the address of the multitasker entry point that deactivates the current task. Used as a code ending (instead of **ret**) at the end of code that initiates an I/O operation, where the interrupt that signals completion of the I/O will be used to wake up the task.

END-CODE *(—)*
Terminate an assembler sequence started by **CODE** or **LABEL**.

IOP *(—)*
Selects IOP assembler mode. IOP code is restricted to **LABEL** definitions.

MPU *(—)*
Selects MPU assembler mode. Set automatically by **CODE** and **:** .

References Interrupt handling, Section 3.2.2
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 4

2.3 REGISTERS

Instead of a conventional register architecture, the PSC1000 MPU has two hardware stacks, an operand stack and a local-register stack; plus a number of registers, as shown in Figure 1.

The operand stack contains eighteen registers and operates as a push-down stack,with direct access to the top three registers (**s0**–**s2**). Arithmetic, logical,

and data-movement operations as well as intermediate result processing are performed on the operand stack. Parameters are passed to procedures and results are returned from procedures on the stack, without the requirement of building a stack frame or necessarily moving data between other registers and the frame. As a true stack, registers are allocated only as needed for efficient use of available storage.

The local-register stack contains 16 registers, and operates as a push-down stack with direct access to the first 15 registers (**r0**–**r14**). These registers plus the remaining register, **r15**, operate together as a stack cache. As a stack, they are used to hold subroutine return addresses, and hence functions as the return stack for the SwiftX kernel.

Both cached stacks automatically spill to memory and refill from memory, and can be arbitrarily deep. Additionally, **s0** and **r0** can be used for memory access.

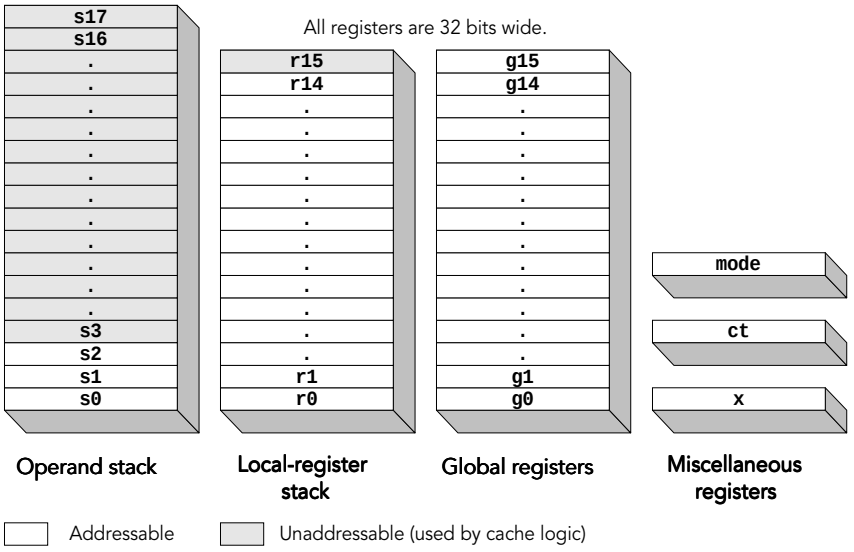

Figure 1. PSC1000 Registers

In addition to the stacks, there are 16 global registers and three miscellaneous registers. The global registers (**g0**–**g15**) are used for data storage, as operand storage for the MPU multiply and divide instructions (**g0**), and for the IOP.

Since these registers are shared, the MPU and the IOP can also communicate through them. Global register assignments are shown in Table 2.

Table 2: Global register assignments

Registers	Description
g0	Scratch; used by some MPU instructions
g1–g3	Scratch
g4–g7	Used by IOP for DMA transfers
g8	Reserved for C cross-development system
g9–g10	Scratch
g11	SwiftX user pointer (**U**)
g12–g15	Reserved for IOP use

Miscellaneous registers include:

- **mode**, which contains mode and status bits;
- **x**, an indirect memory accessg11 register (in addition to **s0** and **r0**);
- **ct**, which is a loop counter and also participates in floating-point operations

lstack is used to specify the local-register stack. For example,

```
lstack push
```

pops one cell from the local-register stack and pushes it onto the operand stack.

To set or obtain the addresses of the memory regions that the operand stack and local-register stack spill into, you may use **sa** and **la**, respectively.

References Register usage in SwiftX implementation, Section 3.1.1

2.4 Addressing Modes

The notation for specifying addressing modes in SwiftX differs from common assembler notation, in that the mode specifiers are operands that precede the

mnemonics. In this PSC1000 assembler, instruction mnemonics are words that actually assemble opcodes using parameters left on the stack by the mode operands. Note that the syntax is `<operand(s)> <opcode>`. In the one case of two operands (for the IOP) the order of operands is `<source> <destination>`. SwiftX assembler modes and operands are separated by spaces.

Since the PSC1000 is a stack-oriented computer, most of its instructions simply operate on the top item or items on the operand stack, and require no further operand notation. For example, **`sub`** subtracts the top item from the next, leaving the difference on top of the stack, and the net stack size is reduced by one. However, in some cases an operand is not on the operand stack; it may be on the local-register stack, for example, or in memory.

2.4.1 General Addressing Modes

Table 3 lists the addressing mode specifiers available for the PSC1000. Most are applicable only to **`ld`** and **`st`** instructions, although **`[]`** may also be used with the branches and some other instructions. Refer to the *PSC1000 Reference Manual* for details.

Note that **`[]`** pops its address operand from the operand stack.

Table 3: Addressing mode specifiers

Specifier	Description
`[]`	Indirect; uses the address in the top stack item. Stack effect is *(addr —).*
`[r0]`	Uses the cell-aligned address in **`r0`**.
`[x]`	Uses the cell-aligned address in **`x`**.
`[--r0]`	Uses the cell-aligned address in **`r0`**, which is pre-decremented by four (one cell).
`[--x]`	Uses the cell-aligned address in **`x`**, which is pre-decremented by four (one cell).
`[r0++]`	Uses the cell-aligned address in **`r0`**, which is post-incremented by four (one cell).
`[x++]`	Uses the cell-aligned address in **`x`**, which is post-incremented by four (one cell).

2.4.2 Literals

Literal operands are preceded by a **#** indicator. The SwiftX assembler will automatically assemble the appropriate form (**push.n** for 4 bits, **push.b** for 8 bits, or **push.l** for 32-bits) depending on the size of the actual number.

For example, here is one example of each form:

```
0 # push            \ Push zero on the operand stack
14 # push           \ Push local-regs on operand stack
GIVE # push         \ Push the addr GIVE on the stack
```

The explicit forms **push.n**, **push.b**, and **push.l** are also available, and perform appropriate range checks on immediate values.

2.5 Direct Transfers

In Forth, most control transfers are performed using structures (such as those described in Section 2.6) and code endings (described in Section 2.2). Good Forth programming style involves many short, self-contained definitions (either code or high level), without the unstructured branching and long code sequences that are characteristic of conventional assembly language. The Forth approach is also consistent with principles of structured programming, which favor small, simple modules with one entry point, one exit point, and simple internal structures. However, direct transfers are useful at times, particularly when compactness of the compiled code overrides all other criteria.

SwiftX supports the **br**, **bz**, **call**, and **dbr** instructions by automatically assembling the appropriate form depending upon the length of the branch. Note that the destination address must always be cell-aligned.

The **skip_** instructions are used around a **br** to form conditionals. However, the structured programming words in Section 2.6 are the preferred way of handling conditionals.

To create a named label for a target location in the host dictionary, use the form:

LABEL <name>

described in Section 2.2. Invoking *name* returns the address identified by the label, which may be used as a destination for a branch or call.

2.6 Assembler Structures

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

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

The program structures supported in this assembler are:

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

In the sequences above, *cc* represents condition codes, which are listed in a glossary beginning on page 14. The combination of a condition code and a structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional skip/branch instruction sequence. The other components of the structures—**BEGIN**, **REPEAT**, **ELSE**, and **THEN**—enable the assembler to provide an appropriate destination address for the branch. The **BEGIN** ... **LOOP** structure uses **ct** as the loop counter. **LOOP** will assemble either **dbr** or **mloop**, depending upon the distance of the branch.

SwiftX's IOP assembler supports **BEGIN … AGAIN** loops, as well as all other IOP mnemonics.

Since the destination of any transfer (**br**, **dbr**, etc.) must be cell-aligned, **BEGIN** always aligns to a group boundary.

Most conditional branches pop a value from the stack and test it, although the carry condition is a status bit set by the previous operation. Thus:

```
xor  0= NOT IF
```

executes the true branch of the **IF** structure if the result of the **xor** of the top two stack items is non-zero.

In high-level Forth words, control structures must be complete within a single definition. In **CODE**, this is relaxed. However, control structures that span routines are not recommended—they make the source code harder to understand and harder to modify.

Table 4 shows the conditions generated by a SwiftX condition specifier. These may be used with **IF**, **WHILE** and **UNTIL**. See the glossary below for details. Refer to your processor manual for details about the condition bits.

Note that the standard Forth syntax for sequences such as **0< IF** implies *no branch in the true case*. Therefore, the combination of the conditional skip and branch instructions assembled by **IF**, etc., branch on the *opposite* condition (i.e., ≥ 0 in this case). If **NOT** is used following a condition, it inverts the condition.

Table 4: Conditions available for SwiftX conditional structure words

Phrase	Description
0<	Branch if greater than or equal to zero.
0< NOT	Branch if less than zero.
0=	Branch if not equal to zero.
0= NOT	Branch if equal to zero.
CS	Branch if the carry bit is not set.
CS NOT	Branch if the carry bit is set.
NEVER	Unconditional branch.

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

In the glossary entries below, the stack notation *cc* refers to a condition code. Available condition codes are listed beginning on page 14.

Glossary **Branch Macros**

BEGIN *(— addr)*

Leave the current code address *addr* on the stack. Doesn't assemble anything.

AGAIN *(addr —)*

Assemble an unconditional branch to *addr*.

UNTIL *(addr cc —)*

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

WHILE *($addr_1$ cc — $addr_2$ $addr_1$)*

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

REPEAT *($addr_2$ $addr_1$ —)*

Set the destination address of the branch that is at $addr_1$ (presumably having been left by **WHILE**) to point to the next location in code space, which is outside the loop. Assemble an unconditional branch to the location $addr_2$ (presumably left by a preceding **BEGIN**).

LOOP *(addr —)*

Assemble a conditional branch to *addr*. Depending upon the distance of the branch, assembles an **mloop** or **dbr** instruction.

IF *(cc — addr)*

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

ELSE *($addr_1$ — $addr_2$)*

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

THEN *(addr —)*

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

Condition Specifiers

0< *(— cc)*

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

0= *(— cc)*

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

CS *(— cc)*

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

NEVER *(— cc)*

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

NOT *(cc_1 — cc_2)*

Inverts the condition code cc_1 to give cc_2.

2.7 Assembly Language Macros

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

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

The only complication lies in the fact that SwiftX assembler commands are not

normally accessible in colon definitions (see "scoping," *SwiftX Reference Manual*, Section 3.6.2). This is necessary because there are assembler versions **IF**, **WHILE**, and other words that have very different meanings in high-level Forth. When you use **CODE** or **LABEL** to start a code definition, those words automatically select the assembler search order, and **END-CODE** restores the previous search order. However, to make macros, you will need to manipulate search orders more directly.

The relevant commands for manipulating vocabularies for assembler macros are given in the *SwiftX Reference Manual*, Section 3.6.2. Here are a few simple examples.

Example 1: U) (User variable referencing)

```
ASSEMBLER
: U) ( addr -- )  \ Address relative to U
    g11 push  [+HOST]  TR0 @ -  ?DUP IF
   [PREVIOUS]  # push  add  [+HOST]  THEN ;
```

This is a simple macro that is commonly used to obtain the address of a user variable by adding an offset to the current user pointer **U**, which is kept in **g11**. For example:

```
S0 U) [] ld
```

...will fetch the contents of **S0** (the empty stack pointer) for the current user.

This mechanism is also used to provide the code for the words that are expanded in place, as shown in the next example.

Example 2: Direct code compilation

```
COMPILER
: 2DUP   [+ASSEMBLER]  s1 push  s1 push ;

TARGET
: 2DUP ( x -- x x)   2DUP ;
```

The SwiftX cross-compiler's version of **2DUP** is executed whenever it encounters a **2DUP** in a **TARGET** colon definition. It will expand the **2DUP** macro in place, assembling the instructions:

```
s1 push
s1 push
```

In the compiler's definition of **2DUP**, it was necessary to add the **ASSEMBLER** scope temporarily to the search order to get access to **s1** and **push**.

The second definition of **2DUP** provides an executable version that you can type during debugging, and whose execution token is returned by **'** and **[']**. It is an essential feature of SwiftX that most target words are available for interactive execution from the command window during a debugging session. Providing these additional definitions adds a tiny amount of space in the target, but the entire strategy provides significant performance improvement. It is cost-effective whenever the overhead of a **call** and **ret** combination is large compared to the actual content of the word itself. Most small, frequently-used Forth words are defined this way.

The glossary below lists the only general-purpose macro in this system.

Glossary

U) *(addr — u)*

Given a user variable address *addr*, return its offset into the user area.

References Register names and usage in SwiftX, Section 2.3

Scoping and search orders (**ASSEMBLER**, **TARGET**, **[+HOST]**, etc.), *SwiftX Reference Manual*, Section 3.6.2.

3. Implementation Issues

This section covers specific implementation issues involving various versions of the PSC1000 processor. For board-specific details, see the relevant appendix.

3.1 Implementation Strategy

A variety of options are available when implementing a Forth kernel on a particular processor. In SwiftX, we attempt to optimize, as much as possible, for both execution speed and object compactness. This section describes the implementation choices made in this system.

3.1.1 Execution Model

The execution model is a subroutine-threaded scheme, with in-line code substitution for simple primitives.

The PSC1000's local-register stack is used as Forth's return stack, and may contain return addresses. Global register **g11** is used as the pointer to the current task or user, often called **U**.

You may see examples of SwiftX PSC1000 optimization strategies by decompiling some simple definitions. For example, the source definition for **DABS** is:

```
: DABS ( d1 -- d2)   DUP 0< IF  DNEGATE  THEN ;
```

but if you decompile it, you get:

```
SEE DABS
15B0   push  0 # push.n  mxm  pop       9220DFB3
15B4   eqz  eqz  skip  skip             E5E53030
15B8   15C0 bz                         10000002
15BC   DNEGATE call                    0FFFFFE6
15C0   ret  skip  skip  skip           6E303030 ok
```

This example clearly shows the combination of in-line code and subroutine calls in this implementation. Note the **skip**s inserted to force alignment.

When a colon definition ends with a call to an external routine, the compiler automatically assembles it as a **br** rather than a **call**, to avoid a redundant **ret**. You may see an example of this by typing **SEE PAUSE**.

If you are an experienced Forth programmer and would like to study these implementation strategies, we encourage you to look at the file **Core.f**, noticing particularly the sections marked **COMPILER**.

3.1.2 Data Format and Memory Access

Because the PSC1000 is a 32-bit processor, it can theoretically address 4 Gb of memory. However, by convention I/O devices are memory-mapped, and the uppermost address bits are used to select I/O device addresses. So the effective contiguous memory addressability is limited to 2 Gb. This address space is, in turn, divided into four "memory groups," selected by bits 26-27 in an address. Refer to the section on the board you are using for further implementation details.

Low memory is occupied by interrupt vectors, up to $14F_H$. Patriot Scientific uses the space from 150_H to $3FF_H$ in its monitor program, so SwiftX as shipped starts its code space at 400_H. You may reconfigure this if you wish.

References PSC Evaluation Card memory map, Figure 2

3.1.3 Stack Implementation and Rules of Use

The Forth virtual machine has two stacks with 32-bit items. The operand stack is Forth's "data stack," and the return stack is the CPU's local-register stack, carrying return addresses for nested calls. A program may use the return stack for temporary storage during the execution of a definition, subject to the following restrictions:

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

3.1.4 SwiftOS Multitasker Implementation

The PSC1000's implementation of the SwiftOS multitasker is somewhat larger than on some chips, with twenty instructions required to suspend a task, but only nine for it to resume. This is because a task's operand and local-register stacks are saved in memory while it is inactive. However, it is reasonably fast.

The subroutine-threaded implementation means there is no **I** register (see address interpreter, 4.3 of the *SwiftX Reference Manual*) to be saved and restored, only the return and data stacks. The address of the saved return stack is pushed onto the data stack, so only one stack pointer needs to be saved in the task's user area. This makes the SwiftOS multitasker slightly faster than the example given in the *PSC1000 Reference Manual.*

The single-cell user variable **STATUS** contains either the **SLEEP** or **WAKE** code (one instruction-group each), and is followed by the **FOLLOWER** cell containing the address of the next task in the round-robin. **SLEEP** passes control to the **FOLLOWER** task, which will execute whichever routine is contained in *its* **STATUS** cell. **WAKE** returns control to the task, restoring its context to the state in which it left off when it called **STOP** or **PAUSE**. The address for the actual code

to perform the wakeup operation is on the stack while SwiftOS is searching for the next task to be activated. The code executed by **SLEEP** and **WAKE** is:

```
SLEEP:     4 # inc  [] ld  push  [] br
WAKE:      g11 pop  [] br
```

If you wish to review the simple code for this SwiftOS multitasker, you will find it in the file **Swiftx\Src\PSC1000\Tasker.f**.

References SwiftOS task activation/deactivation, *SwiftX Reference Manual*, 4.2.1

3.1.5 XTL Implementation

The PSC1000 SwiftX implementation for the Patriot Scientific Evaluation Card (see Appendix A) uses port 0 (lower connector) of the 16C552A chip on that card for the Cross-Target Link (XTL). The XTL's serial protocol is described in 3.9 of the *SwiftX Reference Manual*.

3.2 INTERRUPTS

SwiftX typically uses interrupts for I/O. Interrupts handle I/O with minimal system overhead, and this strategy meshes well with SwiftOS multitasking. This section describes interrupt handling on the PSC1000.

3.2.1 Interrupt Vectors

The PSC1000's interrupts have been given SwiftX names that are the same as those used in Patriot's **sysmem.def** file. SwiftX's definitions are in the file **Swiftx\Src\Psc1000\Vectors.f**. These names are shown in Table 5. The vectors may be referenced by name in conjunction with **INTERRUPT** for attaching service code to interrupts.

Table 5: Named interrupt vectors

Address	Name	Description
IOP software reset vector		
010	`iop_resetv`	
External/DMA interrupt executable vector locations		
100	`int0_trapv`	
104	`int1_trapv`	
108	`int2_trapv`	
10C	`int3_trapv`	
110	`int4_trapv`	
114	`int5_trapv`	
118	`int6_trapv`	
11C	`int7_trapv`	
Floating point trap executable vector locations		
120	`fpexp_trapv`	Exponent trap
124	`fpunf_trapv`	Underflow trap
128	`fpovf_trapv`	Overflow trap
12C	`fpnorm_trapv`	Normalize trap
130	`fprnd_trapv`	Round trap
Debugging trap executable vector locations		
134	`bkpt_trapv`	Breakpoint trap
138	`step_trapv`	Single step trap
Memory fault executable vector location		
13C	`memflt_trapv`	Memory fault trap
Stack overflow and underflow		
140	`lrsovf_trapv`	Local register stack overflow trap

Table 5: Named interrupt vectors

Address	Name	Description
144	**lrsunf_trapv**	Local register stack underflow trap
148	**osovf_trapv**	Operand stack overflow trap
14C	**osunf_trapv**	Operand stack underflow trap

3.2.2 Interrupt Handlers

The procedure for defining an interrupt handler in SwiftX involves two steps: defining the actual interrupt-handling code, and attaching that code to a processor interrupt or trap number.

The handler itself is written in code. The usual form begins with **LABEL** <name> and ends with an **reti** (Return from Interrupt) and **END-CODE**.

To attach the code to the handler, use the word **INTERRUPT**, which takes an address for the handler code and an interrupt number, and links them such that when the interrupt occurs, it will be vectored directly to the code through a simple dispatcher described below. No task needs to be directly involved in interrupt handling. If a task is performing additional high-level processing (for example, calibrating data acquired by interrupt code), the convention in SwiftX is that the handler code performs only the most time-critical processing, and notifies a task of the event by modifying a variable or by setting the task to wake up. Further information on task control may be found in 4. of the *SwiftX Reference Manual*.

The PSC1000's interrupt vector are located in code space addresses 100_H–$14F_H$. Each unused vector in the table is initialized to **reti**.

Any registers used by the handler must be preserved, usually with **push** and **pop** instructions. Interrupts should be left disabled while a handler is executing. If you desire nested interrupt operation, the handler may explicitly enable interrupts, and nesting will then occur.

When a handler returns, execution resumes where it was when the interrupt occurred.

An example of a simple interrupt handler is the one provided for the IOP-generated MPU interrupt in **\Swiftx\Src\PSC1000\Evc\Iopclock.f**. It looks like this:

```
LABEL <TIMER>          \ Clock interrupt
   mode push  x push  di \ Save mode & x, disable interrupts
   MSECS 3 CELLS + # push   x pop      \ Get date/time
   [--x] ld   1 # push   add   [x] st
   [--x] ld   0 # push   addc  [x] st  \ Increment,
   [--x] ld   1 # inc   [x] st         \   handle carry
   x pop   mode pop   reti   END-CODE
```

<TIMER> is the MPU interrupt handler. This code accumulates a millisecond count and date/time information in **MSECS**.

Power-up initialization for all interrupts is done by the word **/VECTORS**, which should be included in the word **START**, found in **\Swiftx\Src\PSC1000**<platform>**\Start.f**.

/VECTORS must be called *before* any other initialization functions that enable specific interrupts.

Further details on the implementation and management of interrupt vectors may be found in **\Swiftx\Src\PSC1000\Vectors.f**

Glossary

INTERRUPT *(addr n —)*

Store address *addr* into the interrupt table entry for vector *n*. Two versions are supplied: the **INTERPRETER** version is used to set the code image vector, while the **TARGET** version sets the vector at run time in the target.

/VECTORS *(—)*

Initializes all interrupt vectors to branch to their dispatcher routines.

References Clock, calendar, and timing libraries, *SwiftX Reference Manual*, 5.3
IOP clock, 4.2

4. WRITING I/O DRIVERS

The purpose of this section is to describe approaches to writing drivers for your SwiftX system. The general approach is not significantly different from writing drivers in assembly language or C: you must study the documentation for the device in question, determine how to control the device, decide how you want to use the device for your application, and then write the code.

However, a few suggestions may help you take advantage of SwiftX's interactive character and Forth's ease of interfacing to various devices. We will discuss these in this chapter, with examples from some common devices.

We shall assume you have some experience writing drivers in other languages or for other hardware.

4.1 GENERAL GUIDELINES

Here we offer some general guidelines that will make writing and testing drivers easier.

1. **Name your device registers**, usually by defining them as **CONSTANT**s or **EQU**s. This will help make your code more readable. It will also help "parameterize" your driver: for example, if you have several devices that are similar except for their hardware addresses, you can write the common control code and pass it a port or register address to indicate a specific device, efficiently reusing the common code.
2. **Test the device** before writing a lot of code for it. It may not work; it may not be connected properly; it may not work exactly like the documentation says it should. It's best to discover these things before you've written a lot of code based on incorrect information, or have gotten frustrated because your code isn't behaving as you believe it should!

If you've named your registers and have your target board connected, you can use the XTL to test your device. You can simply read and write memory-mapped ports, using **@** and **!** (or **C@** and **C!**, as appropriate), and use the **.** ("dot") command to display the results. (Usually you want the numeric base set to **HEX** when doing this!).

Try reading and writing registers; send some commands and see if you get the results you expect. In this way, you can explore the device until you really understand it and have verified that it is at least minimally functional.

3. **Design your basic strategy for the device.** For example, if it's an input device, will you need a buffer, or are you only reading single, occasional values? Will you be using it in a multitasked application? If so, will more than one task be using this device? In a multitasked environment, it's often advisable to use interrupt-driven drivers so I/O can proceed while the task awaiting it is asleep, and other tasks can run. An interrupt (or expiration of a count of values read, etc.) can wake the task. If the device is used by just a single task, you can build in the identity of the task to be awakened; if multiple tasks will be using it, you can use a facility variable to control access to the device and to identify which task to awaken. See the section on the SwiftOS multitasker in the *SwiftX Reference Manual* for a discussion of these features.
4. **Keep your interrupt handlers simple!** If you're using interrupts, the recommended strategy is to do only the most time-critical functions (e.g., reading an incoming value and storing it in a buffer or temporary location) and then wake the task responsible for the device. High-level processing can be done by the task after it wakes up.
5. **Respect the SwiftOS multitasking convention** that a task must relinquish the CPU when performing I/O, to allow other tasks to run. This means you should **PAUSE** in the I/O routine, or **STOP** after setting up streamed I/O that will take place in interrupt code.

4.2 EXAMPLE: SYSTEM CLOCK

SwiftX programs the IOP to refresh DRAM and interrupt the MPU at 1 ms intervals. The MPU uses this to provide basic time-of-day services. This provides a good example of a simple interrupt routine, as well as the interplay between the IOP and MPU. The complete source may be found in **Swiftx\Src\PSC1000\Evc\Iopclock.f**.

We would like to be able to set a time of day, and have it updated automatically. Our first design decision is the units to store: milliseconds, seconds, or just a count of clock ticks. Storing clock ticks provides the simplest interrupt code, but requires the routine that delivers the current time of day to convert clock ticks to time units. This is the best approach, as it minimizes the low-level code. But since the IOP's timing is based on the time to execute a microloop, which is, in turn, dependent on cycle time, this isn't straightforward. A set of calculations performed at compile time generates the constants needed to do this.

There are actually two processors involved: the IOP executes a simple loop, performing refreshes and generating an MPU interrupt every millisecond. Its code looks like this:

```
IOP

LABEL IOP_CLOCK
  ILOOP_DELAY # g7 ld   \ Inner loop delay count
  OLOOP_DELAY # g6 ld   \ Outer loop delay count
  BEGIN   1 int  g6 delay \ Interrupt MPU, outer loop delay
     REFRESH_COUNT # g15 ld  nop \ Establish loop count
     refresh  refresh   \ Two refresh cycles
     g7 delay  g15 mloop \ Inner loop delay
  AGAIN   END-CODE      \ Infinite loop

MPU
```

This feature has no multitasking impact. No task owns the clock, nor does any task directly interface to it. Instead, the clock interrupt code on the MPU just runs along, incrementing its counter, and any task that wants the time can read it by calling the word **COUNTER** (or one of the higher-level words that calls it).

Our counter will be a **VARIABLE** named **TICKS**. The interrupt routine looks like this:

```
LABEL <TIMER>          \ Clock interrupt
   mode push  x push  di \ Save mode & x, disable interrupts
   MSECS 3 CELLS + # push   x pop      \ Get date/time
   [--x] ld   1 # push   add   [x] st
   [--x] ld   0 # push   addc  [x] st  \ Increment,
   [--x] ld   1 # inc   [x] st         \   handle carry
   x pop   mode pop   reti   END-CODE
```

The interrupt vector is set in the startup code, along with similar initialization functions, by the phrase:

```
<TIMER> int1_trapr INTERRUPT
```

Many systems handle the midnight rollover in clock interrupt code. However, it's inefficient to check so frequently for something that happens so infrequently! In this example, we check in the word **@NOW** (see *SwiftX Reference Manual*, Section 5.3), which is the command for fetching the system date and time. Providing your application code always uses **@NOW** (or the higher-level words that call it) to fetch the time rather than reading **SECONDS** directly, you need never worry about midnight rollover, and the overall cost to the system is minimized.

4.3 EXAMPLE: GENERAL SERIAL I/O

General serial I/O provides a more complex example. The Forth language provides a standard Application Programming Interface (API) for serial I/O, with commands for single-character input and output (**KEY** and **EMIT**, respectively), as well as for stream input and output (**ACCEPT** and **TYPE**, respectively). Basic principles of serial I/O in Forth are described in the *Forth Programmer's Handbook*, Section 3.8.

In SwiftOS, we assume that a terminal task may have a serial port attached to it. The serial I/O commands are vectored in such a way that a definition containing **TYPE**, for example, will output its string to the port attached to the task, executing the definition using that task's vectored version of **TYPE**. Thus, you can write a definition that produces some kind of display and, if a task attached to a CRT executes it, the output will go on the screen; but if a task controlling a printer executes it, the text will be printed.

Our mission in writing a serial driver is to provide the *device-layer versions* of these standard routines.

As shipped, SwiftX includes two serial drivers: one uses the XTL to provide access to the host's keyboard and screen as a virtual terminal; the other communicates to a "dumb terminal" (or terminal emulator) connected to a serial port. You may find it interesting to compare them. They may be found in **Swiftx\Src\Psc100\Evc\Serial0.f** and **..\Serial1.f**, respectively.

There are two basic approaches to implementing serial drivers in Forth, which differ depending on whether the primitive layer is single-character I/O or streamed I/O. In the first case, the primitives support **KEY** and **EMIT**; **ACCEPT** then consists of **KEY** inside a loop, and **TYPE** consists of **EMIT** inside a loop. In the second case, **KEY** and **EMIT** call **ACCEPT** and **TYPE**, respectively, with a count of 1. Both approaches are valid: single-character I/O is simpler to implement, but streamed I/O is optimal for systems on which many tasks may be performing serial I/O concurrently. The driver discussed here uses single-character I/O.

The first basic decision to make is whether the I/O will be interrupt driven or polled. A polled driver checks the device status to see whether an event has occurred (e.g., a character has been received or is ready for output), whereas an interrupt-driven approach relies on an interrupt to signal that an event has occurred. Interrupt-driven drivers tend to have less overhead, but polled drivers are easier to implement and test.

In addition, the nature of the device has some bearing. For example, incoming keystrokes from a keyboard or pad are relatively infrequent (occurring at human, rather than computer, speeds); if polling were used, the routine would check many, many times before the next character arrives, thus creating needless overhead. On the other hand, when sending a string of characters, the next character is ready as soon as the last one is gone (which will be quickly). For such situations, we would use polled output and interrupt-driven input.

In the sections that follow, we'll show how we would write and test the device-layer serial I/O words, and then show how to connect them to the high-level words in the serial API.

4.3.1 I/O Registers

The Patriot EVC includes a 16C552, which provides two serial channels and a parallel port. SwiftX defines names for its memory-mapped registers in the file **Swiftx\Src\Psc100\Evc\Reg_Evc.f**), as shown in Table 6. These registers may be referenced by these names in code or in high-level definitions; they may also be interrogated interactively if your target is connected.

Table 6: Control registers for the EVC's 16C552 serial ports

Address	Name	Description
Serial Port 0, $98000000		
0	**THR0**	Transmitter holding register
0	**RBR0**	Receiver buffer register
0	**DLL0**	Divisor Latch LSB
1	**DLM0**	Divisor Latch MSB
1	**IER0**	Interrupt enable register
2	**ISR0**	Interrupt Status Reg
2	**FCR0**	FIFO Control Register
3	**LCR0**	Line Control Register
4	**MCR0**	Modem Control Register
5	**LSR0**	Line Status Register
6	**MSR0**	Modem Status Register
Serial Port 1, $A8000000		
0	**THR1**	Transmitter holding register
0	**RBR1**	Receiver buffer register
0	**DLL1**	Divisor Latch LSB
1	**DLM1**	Divisor Latch MSB
1	**IER1**	Interrupt enable register
2	**ISR1**	Interrupt Status Reg
2	**FCR1**	FIFO Control Register
3	**LCR1**	Line Control Register
4	**MCR1**	Modem Control Register
5	**LSR1**	Line Status Register
6	**MSR1**	Modem Status Register

4.3.2 Polled Serial Output

To output a single character, all you have to do is verify that the port is ready, and then write the character to the port.

We can test the port easily, by writing a single character directly in its transmitter register:

```
41 THR1 !
```

You could make this into a definition and edit it into a file, but it may be just as easy to type it at the keyboard, if your XTL is active.

This should send an A character to whatever is connected to Port 1 (e.g., a terminal emulator). If this works, we then must take care of the fact that, if we're calling it repeatedly in a loop (for streaming output), the port may not always be ready. That can be handled this way:

```
: (EMIT) ( char -- )
   BEGIN   LSR1 @         \ Read line status register
      $20 AND  UNTIL      \ Repeat till ready
   THR1 ! ;               \ Output character
```

You can test this by putting it in a loop:

```
: TRY   ( addr n -- )     \ Output n chars from addr
   0 DO  DUP C@ (EMIT)  1+ LOOP  DROP ;

PAD 50 65 FILL            (puts 50 A characters at PAD)
PAD 50 TRY                (should output 50 A characters)
```

All that remains is to fulfill the requirement that I/O words should relinquish control of the CPU for the multitasker. Since the time when other tasks potentially could run is in the polling loop, while we are waiting for input, that is where we should give other tasks the opportunity to run. The final primitive word, then, becomes:

```
: (S1-EMIT) ( char -- )
   BEGIN   PAUSE          \ PAUSE for multitasker
      LSR1 @              \ Read line status register
      $20 AND  UNTIL      \ Repeat till ready
   THR1 ! ;               \ Output character
```

The low-level word for **TYPE** is simply the single-character **EMIT** behavior, **(S1-EMIT)** in this case, in a loop:

```
: (S1-TYPE) ( addr u -- )    \ Output u chars from addr
   0 ?DO                     \ Repeat for u chars
      COUNT (S1-EMIT)        \ Output next char
   LOOP DROP ;               \ Done; discard addr
```

Note that the use of **COUNT** here takes advantage of the literal behavior of the word: it takes an address, and returns a byte from that address plus the address of the next byte. Although **COUNT** is designed to return the length and address of a counted string (whose length is in its first byte), it is also perfect for running through a string, as in this case.

References Principles of serial I/O in Forth, *Forth Programmer's Handbook*, Section 3.8
COUNT, *Forth Programmer's Handbook*, Section 2.3.5.2
PAUSE and multitasker requirements, *SwiftX Reference Manual*, Section 4

4.3.3 Interrupt-driven Queued Serial Input

Input is somewhat more complex than output. Rather than reproduce the entire code here, refer to the file **Swiftx\Src\Psc1000\Evc\Serial1.f** as you read these notes.

We usually need to buffer incoming characters. We certainly don't want to miss a character because we were busy when it appeared in the receiver. So the input side of this driver will have a 256-byte buffer, plus four pointers used to manage the process. The buffer itself has no name, but pointers into it are defined; the actual data begins at **RDATA**. The layout is given in Table 7.

Table 7: Input queue pointers

Offset name	Description
RIN	Offset for next received byte
ROUT	Offset to next byte to remove
RTASK	Task to be awakened
RDATA	Start of actual data

As is common in SwiftOS, we separate interrupt-level processing from task-level processing, doing only the bare minimum at interrupt time. In this case,

the interrupt code fetches the character from the input port and puts it in the next input location, indicated by **RIN**, and awakens the task responsible for the port. This is done by **<SERIAL1>** (just as the convention of names in parentheses is used to indicate the low-level components of high-level functions, the convention of names in angled brackets is used to indicate interrupt routines).

The phrase:

```
<SERIAL1> int3_trapv INTERRUPT
```

(found in **/SERIAL1** at the end of the file) attaches the code to the interrupt vector for serial port 1.

A dummy task called **NOTASK** is provided; it is only a variable, not a real task, because it only serves as a place for the interrupt code to store the "wake-up" value if an interrupt occurs when *no* task is asleep awaiting character input from this port.

The balance of the processing of incoming characters is done by task-level code. There are two definitions, described in Table 8.

Table 8: Serial input primitives

Word	Stack	Description
(S1-KEY?)	*(— flag)*	Returns a flag that is true if a character has been received. The primitive for **KEY?**.
(S1-KEY)	*(— char)*	Returns the next character from the queue.

(S1-KEY) must wait until a key is received, and in the SwiftOS multitasking environment the waiting must be done in an inactive state, so other tasks can run. To do this, it **STOP**s until a key is ready, then reads it. The task will be suspended until it's awakened by the keyboard interrupt, at which time it will **PAUSE** once more (ensuring that there is at least one **PAUSE**, even if a key was already in the queue, and that the task's **STATUS** is not set to **WAKE**) and then read the key.

4.3.4 Port and Task Initialization

All that remains is to provide port initialization, and to attach these device-layer functions to an actual SwiftOS task.

The word **/SERIAL1** in the file **Swiftx\Src\Psc1000\Evc\Serial1.f** initializes the port (the naming convention /name is often used for words that initialize a port, device, or function *name*). As usual with initialization routines, it is intended to be called in the high-level **START** routine in the file **Swiftx\Src\Psc1000\Evc\Start.f**. **/SERIAL1** initializes the buffer pointers and sets a default baud rate and port control bits.

S1-TERMINAL in **Swiftx\Src\Psc1000\Evc\Serial1.f** is intended to be executed by a terminal task to set its vectored serial I/O functions to the device-layer versions for Serial Port 1. As an example of how this is done, look at the definition of the terminal task set up to run the demo application at the end of **Swiftx\Src\Psc1000\Evc\Try.f**:

```
256 TERMINAL CONSOLE          \ Define a task

: DEMO   CONSOLE ACTIVATE     \ Start task
   DUMB S1-TERMINAL           \ Initialize task vectors
   BEGIN  CALCULATE  AGAIN ;\ Run CALCULATE indefinitely
```

Here, the command **DEMO** starts the task **CONSOLE** executing the balance of the definition following **ACTIVATE**; it performs the initialization steps **DUMB** and **S1-TERMINAL**, and then enters an infinite loop performing the **CALCULATE** demo routine.

APPENDIX A: PSC1000 EVALUATION CARD II CONFIGURATION AND USE

This section provides information pertaining to the PSC1000 Evaluation Card II (EVC), which is supported by SwiftX for the PSC1000. It includes instructions for setting up your board and connecting it to your host PC, as well as additional information specific to the implementation of the SwiftX kernel and SwiftOS on this board.

A.1 BOARD DESCRIPTION AND CONFIGURATION

The Evaluation Card is a demonstration board provided by Patriot Scientific, whose principal features are shown below.

- PSC1000 Processor
  - User-replaceable oscillator
- Memory
  - On board Fast SRAM 32Kx32 or 64kx32 or 512kx32
  - On board Fast Page DRAM (1Mx32)
  - Small Outline 72 pin DIMM - Fast page or EDO DRAM DIMM 1Mx32 or 4Mx32
  - Flash ROM - 32kx8 to 512kx8 or 32kx32 to 512kx32
- I/O Devices
  - TL16C552A Multi I/O Chip
  - Serial Port 0 (DCE)
  - Serial Port 1 (DTE or DCE)
  - Bi-directional Parallel Port
  - Configuration Register

- Status Register
- User defined push button
- 8 User programmable LED's

- FPGAs (Altera Max7032)
  - Programmable decode for I/O and soDIMM memory
  - Programmable use for PSC1000 input and output pins
- Debug/Expansion
  - Seven HP 1660 Logic Analyzer/Debug/Expansion connectors
  - Reset push button/external reset connector
- Connectors
  - J1 External PC disk drive style power connector
  - J2 External 5.5mmx2.1mm 5V power connector
  - JP1 External /Reset connector
  - J3A DCE Serial port 0 (lower DB9M connector; used for XTL)
  - J3B DTE Serial port 1 (upper DB9M connector; used for 2nd serial port)
  - J4 Bi-directional Parallel Port
  - J12..J6 Expansion/ HP 1660 Logic
- Analyzer/Expansion connectors
  - J6,J9 All 32 bits of the AD bus
  - J7 All PSC1000 Input and Output Signals
  - J8 PSC1000 Memory Control Signals and Spare I/O
- FPGA Signals
  - J10,J11 24 CAS address bit and 8 RAS address bits
  - J12 All outputs of decode FPGA
  - J13 DCE Serial port 1 (2x10 IDC)
  - J14 DTE Serial port 1 (2x10 IDC)

As noted in Section 3.1.2, the PSC1000 uses address bits 26-27 to access different memory groups. On the EVC, these groups are mapped as shown in Table 9.

Table 9: PSC1000 memory group assignments on the EVC

Group	Description
0	Fast SRAM
1	On-board DRAM
2	DIMM DRAM
3	Flash memory (byte-wide)

As shipped, SwiftX is configured for 64K cells (256KB) of SRAM, organized as shown in Figure 2.

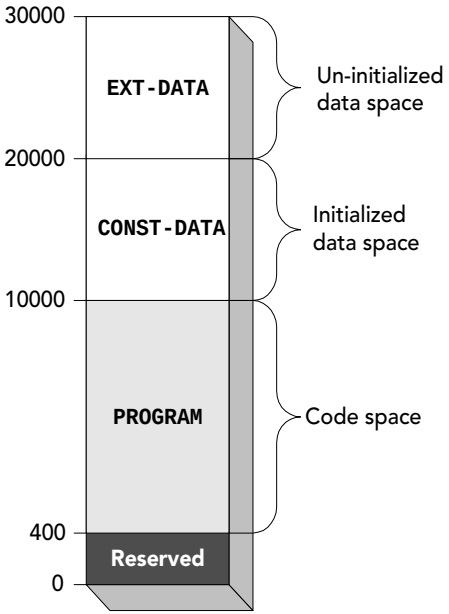

Figure 2. Memory configuration in PSC EVC

A.2 Development Procedures

On the PSC1000 Evaluation Card II, the SwiftX kernel runs in the SRAM memory (group 0). As shipped, this kernel is set up to be downloaded using the

PSC download utility. However, the kernel is configured so that it may also be installed in flash.

A.2.1 Downloading a New Kernel to SRAM

SwiftX installs a copy of Patriot's download utilities in **`SwiftX\Bin\Psc`**. It also provides a **`Download.bat`** file in the **`SwiftX\Src\Psc1000\Evc`** directory, which performs all the necessary steps to download a new SwiftX kernel to SRAM while Patriot's SHMON EPROM is installed in socket U5.

In order for SwiftX's Cross-Target Link (XTL) to work properly, the object generated by your kernel source must exactly match the installed kernel. Therefore, whenever you make any changes to your kernel, you must replace the kernel in SRAM.

To download a new kernel to SRAM (after every power-up or reset):


Build

1. Select the Project > Build menu item. This generates a new **`Target.bin`** object file.
2. Press the Reset button (SW1) on the EVC. The LEDs of DP3 and DP4 should flash briefly, and one will continue to blink.
3. Run the **`Download`** batch file. You may do this using the Run toolbar button or Tools > Run menu option, or launch it from Windows Explorer. This runs the Patriot utility in a DOS box. The download takes several seconds, during which there is no visual feedback. The process is complete when you see the messages:


Run

```
Entry point:0x400
address=0x400
SHMON> !\n
?>
!g\n
```

   After the download, the LEDs will display a continuously changing count.

4. Close the DOS box. The batch file leaves it open in case an error should occur.
5. From SwiftX, continue as described in the next section.

A.2.2 Starting a Development Session


Debug

Launch SwiftX (if you have not already done so) as described in Section 1.3.1 of the *SwiftX Reference Manual*. You may change any options you wish and, when you are ready, you may compile your kernel by using the Debug toolbar button or selecting Project > Debug from the menu. This compiles the kernel and compares it to the target's kernel in SRAM.

If the source has changed since your last download, you will get the message, `Kernel Mismatch`, in which case you must generate a new code image and install it in SRAM.

If the board is not connected properly, you will get the message, `No XTL. Try again? (y/n)`. This means the kernel was properly compiled, but the host failed to establish XTL communication with the target. Check your connections (the XTL uses Serial Port 0, the lower connector) and select Project > Debug again.

If the connection was successfully established and the host version of the kernel matches the PROM's kernel, the target will display the system ID and its creation date. 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 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 of the *SwiftX Reference Manual*.

A.2.3 Running the Demo Application

As shipped, the interactive testing program **`Debug.f`** (loaded by the Project > Debug menu item) is configured to run a demo application described in the *SwiftX Reference Manual*.

The demo program uses the host keyboard and screen via the XTL (the default configuration), or you may connect the EVC's second serial port (upper connector) to a COM port on your host, using a separate task to talk to a standard terminal emulator utility in the PC, as described in Section 4.4 of the *SwiftX Reference Manual*.


Debug

To run the demo program, select Project > Debug to bring up the target program, which will run the demo program as a separate task using the second serial port. Type **`CALCULATE`** to start the application if you want it to use the XTL.

Thereafter, follow the instructions in the *SwiftX Reference Manual*.

A.2.4 Installing a New Kernel in EPROM

You PSC1000 Evaluation Card II board is shipped with a Patriot utility in its on-board PROM. If you will be using SwiftX exclusively with this board, you may install a SwiftX boot and kernel in the PROM. You may also use the PROM boot code as a model for start up code in your ultimate target. The source may be found in **`SwiftX\Src\PSC1000\Boot.f`**.

In order for SwiftX's Cross-Target Link (XTL) to work properly, the object generated by your kernel source must exactly match the installed kernel. Therefore, if you make any changes to your kernel, you must replace this kernel.


Build

To install a new kernel,

1. Select the Project > Build menu item or Toolbar button. This loads the file **`Build.f`**, which generates two binary object files:

   ```
   LOADER   SAVE-CODE TARGET0.BIN
   PROGRAM  SAVE-CODE TARGET.BIN
   ```

   `TARGET0.BIN` contains the boot loader and must be programmed into the

boot ROM device at offset 0. **`TARGET.BIN`** contains the the SwiftX kernel itself which the boot loader moves toRAM for execution. It must be programmed at offset 400_H in the boot ROM device.


Run

2. Use a PROM programmer utility of your choice to burn a new PROM using these files. You may use the Tools > Run menu item or Toolbar button to do this.
3. Turn off power to the board. Change the PROM, then restore power to the board.
4. Continue as described in Section A.2.2.

GENERAL INDEX