Getting Results Faster...

SwiftX HC12

Board-level Documentation for 68HC12 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

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 68HC12 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 68HC12; 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 68HC12. Refer to Motorola's documentation for information about the 68HC12, 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
Axiom CMD12A4	Appendix A: Axiom CMD12A4 Instructions	43
Axiom CME12B32	Appendix B: CME12B32 Board Instructions	49
MC68HC912B32 EVB	Appendix C: Motorola EVB Instructions	55

   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. (The uninstall option is provided here in Windows 3.1 only; if you are running Windows 95, you can use the Remove button on the Start > Settings > Taskbar... > Start Menu Programs dialog.)

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 68HC12 Assembler

The CPU12 is a high-speed, 16-bit processing unit. It has full 16-bit data paths, with wider internal registers (up to 20 bits) for high-speed extended math instructions. The instruction set is a superset of the M68HC12 instruction set. The CPU12 allows instructions with odd byte counts, including many single-byte instructions; this provides efficient use of ROM space. An instruction queue buffers program information so the CPU always has immediate access to at least three bytes of machine code at the start of every instruction. The CPU12 also offers an extensive set of indexed addressing capabilities.

Throughout this book, we assume you understand the hardware and functional characteristics of the 68HC12 and its CPU12 processor core as described in the *Motorola CPU12 Reference Manual*. We also assume you are familiar with the basic principles of Forth.

This section supplements, but does not replace, the CPU 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 Motorola names. Usually these are the same; for example, the name **`LDAA`** can be used as a Forth word and as a Motorola mnemonic. Where boldface is *not* used, the name refers to the manufacturer's usage or to hardware issues that are not particular to SwiftX or Forth.

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 68HC12 cross-compiler provides an assembler for the Motorola CPU12. The mnemonics for the CPU12 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 jumps. SwiftX constructs conditional jumps by using a condition code specifier followed by **IF**, **UNTIL**, or **WHILE**. Table 5 on page 11 summarizes the relationship between SwiftX condition codes used with one of these words and the corresponding Motorola mnemonic.

References Assemblers in Forth, *Forth Programmer's Handbook*, Sections 1.3 and 4.0

2.2 Code Definitions

Code definitions normally have the following syntax:

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

For example:

```
CODE DEPTH ( -- n)      \ Return the current stack depth
   2 ,-Y STY   U LDX            \ Push current S
   S0 ,U LDD   0 ,Y SUBD   LSRD \ Subtract initial S
   0 ,Y STD   RTS               \ Convert to cell count
END-CODE
```

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

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

```
    WAIT BRA
```

may be used before **END-CODE** to terminate a routine. It returns 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 **EXCEPTION**, which connects the code address to a specified exception vector.

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

- 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 labelled code.

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 **RTS**) 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**.

References Interrupt handling, Section 2.7
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 5

2.3 Registers

CPU12 registers, shown in Figure 1, are an integral part of the CPU and are not addressed as if they were memory locations.

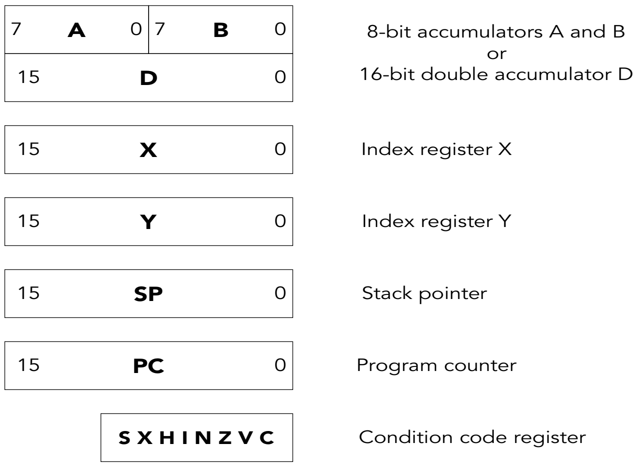

Figure 1. CPU12 Registers

- **Accumulators A and B** are general-purpose, 8-bit accumulators used to hold operands and results of arithmetic calculations or data manipulations. Some instructions treat these two 8-bit accumulators as the single 16-bit double accumulator D.

- **Index registers X and Y** are used for indexed addressing mode, in which the contents of a 16-bit index register are added to 5-bit, 9-bit, or 16-bit constants or the contents of an accumulator to form the effective address of the operand to be used in the instruction.
- **Stack pointer (SP)** points to the last stack location used. The CPU12 supports an automatic program stack that is used to save system context during subroutine calls and interrupts, and can also be used for temporary storage of data. The stack pointer can also be used in all indexed addressing modes.
- **Program counter** is a 16-bit register that holds the address of the next instruction to be executed. The program counter can be used in all indexed addressing modes except autoincrement/decrement.
- **Condition code register (CCR)** contains five status indicators, two interrupt masking bits, and a STOP disable bit. The five flags are half carry (H), negative (N), zero (Z), overflow (V), and carry/borrow (C). The half-carry flag is used only for BCD arithmetic operations. The N, Z, V, and C status bits allow for branching based on the results of a previous operation.

Index register **`Y`** is used as the Forth data stack pointer and **`SP`** is the return stack pointer; if you need to use either of these for another purpose, its contents must be saved and restored. Registers **`A`** and **`B`** (known as **`D`** when used as a 16-bit accumulator) and index register **`X`** are always available for scratch use within code routines.

Additional internal I/O registers are defined as constants for use by the SwiftX assembler, using the published manufacturer names. These internal registers (such as **`PORTA`**) can be referred to by name inside both code and high-level Forth definitions. They are also available to the host command-line interpreter during interactive debugging. The actual register list varies somewhat with the various members of the CPU12 family (see Section 3.3).

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 CPU12 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>`.

Table 2 lists the addressing modes of the CPU12 and offers examples which show the difference between Motorola and SwiftX assembler notation. Note that SwiftX assembler modes and operands are separated by spaces.

Table 2: Addressing modes

Mode	Motorola	SwiftX
Inherent	ABA	`ABA`
Immediate	LDX #1234	`1234 # LDX`
Direct	LDAA PORTA	`PORTA LDAA`
Extended	LDX 1234	`1234 LDX`
Relative	BRA FOO	`FOO BRA`
Indexed (fixed offset)	LDD 0,X	`0 ,X LDD`
Indexed (pre-decrement)	STD 2,-Y	`2 ,-Y STD`
Indexed (pre-increment)	STD 2,+Y	`2 ,+Y STD`
Indexed (post-decrement)	STD 2,Y-	`2 ,Y- STD`
Indexed (post-increment)	STD 2,Y+	`2 ,Y+ STD`
Indexed (accumulator offset)	LDD D,X	`D,X LDD`
Indexed-Indirect (fixed offset)	LDD [1234,X]	`1234 [,X] LDD`
Indexed-Indirect (accumulator offset)	JMP [D,X]	`[D,X] JMP`

A few CPU12 instructions take two operands that may use more than one addressing mode. These are the move and bit-manipulation instructions. Table 3 and Table 4 have examples of Motorola and SwiftX assembler formats for the move and bit-manipulation instructions.

Table 3: Multiple-mode move instructions

Operation	Motorola	SwiftX
Immediate → Extended	MOVW #1234,5678	`1234 # 5678 MOVW`
Immediate → Indexed	MOVW #1234,0,X	`1234 # 0 ,X MOVW`
Extended → Extended	MOVW 1234,5678	`1234 5678 MOVW`
Extended → Indexed	MOVW 1234,0,X	`1234 0 ,X MOVW`

Table 3: Multiple-mode move instructions *(continued)*

Operation	Motorola	SwiftX
Indexed → Extended	MOVW 0,X,1234	`0 ,X 1234 MOVW`
Indexed → Indexed	MOVW 0,X,2,X	`0 ,X 2 ,X MOVW`

Table 4: Multiple-mode bit-manipulation instructions

Mode	Motorola	SwiftX
Direct	BSET PORTA,#1	`PORTA 1 # BSET`
Extended	BSET 1234,#1	`1234 1 # BSET`
Indexed (fixed offset)	BSET 0,X,#1	`0 ,X 1 # BSET`

2.5 Direct Transfers

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

However, direct transfers are useful at times, particularly when compactness of the compiled code overrides all other criteria. **`JMP`** and **`JSR`** are defined as described in Motorola documentation, except they automatically choose the smallest, fastest form compatible with the given addresses (i.e., **`BRA`** may be substituted for **`JMP`**, or **`BSR`** for **`JSR`**).

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 **`JMP`** or a **`JSR`**.

For example, in the target code for the serial cross-target link, we find:

```
LABEL <C@>   1 # LDAB   (C@) JMP   END-CODE
```

This puts the value 1 in accumulator **B** and branches to a routine named **(C@)**, which was also defined by **LABEL**.

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> <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 13. The combination of a condition code and a structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional branch instruction *Bcc*, where *cc* is the condition code. The other components of the structures—**BEGIN**, **REPEAT**, **ELSE**, and **THEN**—enable the assembler to provide an appropriate destination address for the branch.

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

```
2 ,Y+ LDX   0= NOT IF
```

executes the true branch of the **IF** structure if the top stack item (popped into **X** by the **LDX** instruction, which set the condition bits) is non-zero. (The word

NOT following a condition code inverts its sense.)

In high-level Forth words, control structures must be complete within a single definition. In **CODE**, this is relaxed: the limitation is that **IF** or **WHILE** cannot branch forward more than 127 bytes in the object code. If it does, the assembler displays the `Range error` message and terminates. Control structures that span routines are not recommended—they make the source code harder to understand and harder to modify.

Table 5 shows the instructions generated by a SwiftX conditional phrase. These examples use **IF**. **WHILE** and **UNTIL** generate the same instructions, but satisfy the branch destinations slightly differently. See the glossary below for details. Refer to your processor manual for details about the condition bits.

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

Table 5: Instructions generated by SwiftX conditional structure words

Phrase	Instruction assembled	Description
0< IF	BPL	Branch if the N bit is not set.
0< NOT IF	BMI	Branch if the N bit is set.
0= IF	BNE	Branch if the Z bit is not set.
0= NOT IF	BEQ	Branch if the Z bit is set.
0> IF	BLE	Greater-than-zero; branch if the Z bit is set or if N and V differ from each other.
0> NOT IF	BGT	Less-than-or-equal; branch if the Z bit is clear and N and V are both set or both clear.
S< IF	BGE	Signed less-than; branch if the N and V bits are both set or both clear.
S< NOT IF	BLT	Signed greater-or-equal; branch if the Z bit is set, or if the N and V bits differ from each other.
CS IF	BCC	Branch if the carry bit is not set.
CS NOT IF	BCS	Branch if the carry bit is set.

Table 5: Instructions generated by SwiftX conditional structure words

Phrase	Instruction assembled	Description
?SET IF	BRCLR	Branch if bit is not set.
?CLR IF	BRSET	Branch if bit is set.
NEVER IF	BRA	Unconditional branch (equivalent to **AGAIN**).

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 13.

Glossary Branch Macros

BEGIN *(— addr)*

Leave the current 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**).

IF *(cc — addr)*

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

ELSE *($addr_1$ — $addr_2$)*

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

THEN *(addr —)*

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

Condition Codes

0< *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on positive (N bit not set).

0= *(— cc)*

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

0> *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on zero or negative (Z bit set or N and V differ).

S< *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch on greater-than-or-equal (N and V bits are the same).

CS *(— cc)*

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

?CLR *(x m — x m cc)*

Given an address parameter *x* and immediate mask *m*, return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a branch if the bits in the location referenced by *x* masked by *m* are all set (1). For example:

`DATA 7 # ?CLR IF`	Branch if low-order three bits of **`DATA`** are non-zero.
`0 ,X 1 # ?CLR IF`	Branch if low-order bit of the byte at **`0 ,X`** is set.

The addressing modes for *x* may be direct, extended, or indexed.

`?SET` *(x m — x m cc)*

Given an address parameter *x* and immediate mask *m*, return the condition code that—used with **`IF`**, **`WHILE`**, or **`UNTIL`**—will generate a branch if the bits in the location referenced by *x* masked by *m* are all clear (0). For example:

`DATA 7 # ?SET IF`	Branch if low-order three bits of **`DATA`** are zero.
`0 ,X 1 # ?SET IF`	Branch if low-order bit of the byte at **`0 ,X`** is clear.

The addressing modes for *x* may be direct, extended, or indexed.

`NEVER` *(— cc)*

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

`NOT` *(cc_1 — cc_2)*

Invert the condition code cc_1 to give cc_2.

2.7 Interrupt Handling

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 **`RTI`** (Return from Interrupt) and **`END-CODE`**. (**`CODE`** is not needed, as such routines are not invoked as subroutines.)

To attach the code to the handler, use the word **`EXCEPTION`**, which takes an address for the trap handler and an interrupt number, and links them such that when the interrupt occurs, it will be vectored directly to the code. No overhead is imposed by SwiftX, and 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 Section

5 of the *SwiftX Reference Manual*.

One example of the use of interrupts is given in Section 4.2. Another example of a simple interrupt handler is the one provided for the real-time interrupt in **`\Swiftx\Src\68hc12\Rti.f`**. It looks like this:

```
LABEL <TICK>   MSECS CELL+ LDD   TC2 # ADDD   MSECS CELL+ STD
   MSECS LDD   TC1 # ADCB   TC0 # ADCA   MSECS STD
   RTIFLG $80 # BSET   RTI   END-CODE

<TICK> V-RTI EXCEPTION
```

`<TICK>` is the real-time interrupt handler, and **`V-RTI`** is the Real-Time Interrupt Vector. It accumulates a millisecond count in the 32-bit counter **`MSECS`** and clears the interrupt flag in **`RTIFLG`**.

Table 6 lists the interrupt vectors that are named in SwiftX (see **`Swiftx\src\68hc11\Ev-ram.f`**).

Table 6: Named interrupt vectors

Vector (hex)	Name	Description
00FE	**`V-RESET`**	Reset
00FC	**`V-COPCLOCK`**	COP clock monitor fail reset
00FA	**`V-COPFAIL`**	COP fail reset
00F8	**`V-TRAP`**	Unimplemented instruction trap
00F6	**`V-SWI`**	Software interrupt
00F4	**`V-XIRQ`**	XIRQ interrupt
00F2	**`V-IRQ`**	IRQ interrupt
00F0	**`V-RTI`**	Real-time interrupt
00EE	**`V-TIMER0`**	Timer channel 0 interrupt
00EC	**`V-TIMER1`**	Timer channel 1 interrupt
00EA	**`V-TIMER2`**	Timer channel 2 interrupt
00E8	**`V-TIMER3`**	Timer channel 3 interrupt
00E6	**`V-TIMER4`**	Timer channel 4 interrupt
00E4	**`V-TIMER5`**	Timer channel 5 interrupt

Table 6: Named interrupt vectors *(continued)*

Vector (hex)	Name	Description
00E2	**`V-TIMER6`**	Timer channel 6 interrupt
00E0	**`V-TIMER7`**	Timer channel 7 interrupt
00DE	**`V-TOF`**	Timer overflow
00DC	**`V-PAOF`**	Pulse accumulator overflow
00DA	**`V-PAIE`**	Pulse accumulator input edge
00D8	**`V-SPI`**	SPI serial transfer complete
00D6	**`V-SCI0`**	SCI 0 interrupt
00D4	**`V-SCI1`**	SCI 1 interrupt
00D2	**`V-ATD`**	ATD interrupt
00D0	**`V-BDLC`**	BDLC interrupt

Power-up initialization for any of these vectors that are to be used in the target should be done by the word **`START`**, which can be found in **`\Swiftx\Src\68hc12\`**`<platform>`**`\Start.f`**.

Glossary

`EXCEPTION` *(addr n —)*

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

3. Implementation Issues

This section covers specific implementation issues involving various versions of the 68HC12 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 68HC12's subroutine stack is used by target primitives as Forth's return stack, and may contain return addresses.

If you depend on the return stack to be identical with the subroutine stack, your code will not be portable to systems which separate these stacks. Do not use the return stack except under the specific rules given in Section 3.1.3.

You may see examples of SwiftX 68HC12 optimization strategies by decompiling some simple definitions. For example, the source definition for **/STRING** is:

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

but if you decompile it, you get:

```
SEE /STRING
8442   2 ,Y+ 2 ,-SP MOVW                180271AE
8446   SWAP JSR                         16808E
8449   0 ,SP 2 ,-Y MOVW                 1802806E
844D   + JSR                            168160
8450   SWAP JSR                         16808E
8453   2 ,SP+ 2 ,-Y MOVW                1802B16E
8457   - JMP                            068167 ok
```

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

When the last thing in a definition is a subroutine reference, the compiler automatically substitutes a **JMP** for the **BSR**, to save the subroutine return. Consider **TUCK**, defined as:

```
 : TUCK ( x1 x2 -- x2 x1 x2)   SWAP OVER ;
```

If you type **SEE TUCK**, you will get:

```
80A4   SWAP BSR                         07E8
80A6   2 ,Y 2 ,-Y MOVW                  1802426E
80AA   RTS                              3D ok
```

However, if you make a new definition:

```
 : -TUCK ( x1 x2 -- x1 x1 x2)   OVER SWAP ;
```

and type **SEE -TUCK**, you get:

```
9844   2 ,Y 2 ,-Y MOVW                  1802426E
9848   SWAP JMP                         06808E ok
```

In both cases, the **MOVW** instruction is the code substituted for **OVER**, but **SWAP** is called with a **JMP** in the second case, because it is the last thing in the definition and *its* **RTS** will handle the return, saving the need for one in **-TUCK**.

More extensive optimization is provided by a powerful rule-based optimizer than can optimize a number of common high-level phrases. This optimizer is normally running, but can be turned off for debugging or comparison purposes. For example, consider this test definition:

```
  : TRY   4 CELLS + 7 AND ;
```

With the optimizer turned off, you would get:

```
SEE TRY
98D1   4 # 2 ,-Y MOVW                         18006E0004
98D6   CELLS JSR                              168196
98D9   + JSR                                  168160
98DC   7 # 2 ,-Y MOVW                         18006E0007
98E1   AND JMP                                068140 ok
```

But with it turned on, you would get:

```
SEE TRY
98C1   8 # 2 ,-Y MOVW                         18006E0008
98C6   + JSR                                  168160
98C9   0 ,Y LDD                               EC40
98CB   CLRA                                   87
98CC   7 # ANDB                               C407
98CE   0 ,Y STD                               6C40
98D0   RTS                                    3D ok
```

This code is only one byte smaller, but it's significantly faster since it has pre-applied **CELLS** to the literal 4, and "in-lined" the instruction for **AND**.

3.1.2 Data Format and Memory Access

Because the 68HC12 is a 16-bit processor, its directly addressable memory space is limited to 64K. Operators for accessing this memory are discussed in Section 3.2. The appendix concerning the board that accompanies your system includes a memory map showing the available extended memory.

The high byte of a 16-bit cell on the 68HC12 is the lowest address—i.e., this is a *big-endian* machine.

3.1.3 Stack Implementation and Rules of Use

The Forth virtual machine has two stacks with 16-bit items, located in RAM. Stacks grow downward in address space. The return stack is the CPU's subroutine stack, and it functions analogously to the traditional Forth return stack (i.e., carries return addresses for nested calls). A program may use the return stack for temporary storage during the execution of a definition, subject to the following restrictions:

- A program shall not access values on the return stack (using **`R@`**, **`R>`**, **`2R@`**, or **`2R>`**) that it did not place there using **`>R`** or **`2>R`**.
- When within a **`DO`** loop, a program shall not access values that were placed on the return stack before the loop was entered.
- All values placed on the return stack within a **`DO`** loop shall be removed before **`I`**, **`J`**, **`LOOP`**, **`+LOOP`**, or **`LEAVE`** is executed.
- 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 CPU12 CPU family supports a very efficient SwiftOS implementation, with six instructions required to deactivate a task and eight to activate one. The subroutine-threaded implementation means there is no **`I`** register (see address interpreter, Section 5.3 of the *SwiftX Reference Manual*) to be saved and restored, only the return and data stack pointers.

The four-byte **`STATUS`** contains a **`NOP`** ($A7_H$) or an indexed **`JSR`** (15_H) opcode in the first byte. The remainder is a **`JMP`** to the next task in the round robin.

Because the **`JSR`** in the active task's **`STATUS`** is always followed by a **`JMP`** (06) opcode, **`WAKE`** decodes to **`6 ,X JSR`**. The appropriate destination address for the task wake-up code is placed in **X** when any task gives up control of the CPU.

Table 7: SwiftOS user status instructions

Name	Value (hex)	Instruction	Description
WAKE	1506	**JSR**	Call to wake-up code pointed to by **X**.
SLEEP	A706	**JMP**	Jump to next task (address in next cell).

These instructions are stored in a task's **STATUS** to control task behavior. For example, **PAUSE** sets it to **WAKE** and deactivates the task; it will wake up after exactly one circuit through the round robin. You may also store **WAKE** in the **STATUS** of a task in an interrupt routine to set it to wake up. When a task is being started, its **STATUS** is set to **SLEEP** as part of the start-up process. Because the return stack is also the CPU's subroutine stack, it is also used to pass information during a task swap. That is, the **JSR** to the wake-up code in the awakening task's **STATUS** passes the task address on this stack.

If you wish to review the simple code for this SwiftOS multitasker, you will find it in the file **Swiftx\Src\68hc12\Tasker.f**.

References SwiftOS task activation/deactivation, *SwiftX Reference Manual*, Section 5.2.1

3.2 Extended Memory Access

Some members of the 68HC11 family provide for access to extended program memory through the **PPAGE** (Program Page) register, and to extended data memory through the **DPAGE** (Data Page) register. Both registers provide additional address lines that are appended to a base address. The **PPAGE** register provides A14-A21, thus extending a 14-bit base address to 22 bits, while the **DPAGE** register provides A12-A19, extending a 12-bit base address to 20 bits.

This section describes SwiftX's support for these features.

3.2.1 Extended Data Memory

SwiftX supplies simple primitives for access to paged data memory. These words may be found in the file **`..\68hc12\Extmem.f`**, and are described in the glossary below.

Paged data memory has a base address of 7000 to 7FFF$_H$. SwiftX accesses this memory using a paged extended address, which consists of the page number (0–FF$_H$) and the offset within the page (0–FFF$_H$). The offset is on top of the stack, allowing the use of simple address arithmetic. This type of address is referred in the stack comments using the notation *e-addr*.

You may define uData sections in paged data memory using the offset in the low half of a 32-bit host cell, and the page number in the upper half. For example:

```
10000 10FFF UDATA SECTION BANK1
20000 20FFF UDATA SECTION BANK2
```

Thereafter, you can define data structures there and access them using the words in the glossary below. For example:

```
BANK1
8 EBUFFER: ITEM-COUNT
: SHOW   8 0 DO  ITEM-COUNT I + EC@ .  LOOP ;
```

Each of the "E" operators is analogous to the standard memory access operator, except it takes an *e-addr* instead of a linear address. **`EDUMP`** is used primarily for diagnostic purposes.

Glossary

`EC@` *(e-addr — b)*

Fetch a byte from *e-addr* in extended memory.

`E@` *(e-addr — x)*

Fetch a cell (16-bits) from *e-addr* in extended memory.

`EC!` *(b e-addr —)*

Store a byte at *e-addr* in extended memory.

E! *(x e-addr —)*

Store a cell at *e-addr* in extended memory.

EBUFFER: `<name>` *(n —)*

Defines a data region *n* bytes in length in extended memory. Use of *name* returns the *e-addr* of the start of the region.

EDUMP *(e-addr n —)*

Dump *n* bytes starting at *e-addr* in extended memory.

3.2.2 Extended Program Memory

Extended program memory support is found in the file **`..\68hc12\Bridge.f`**, which must be loaded by your **`Kernel.f`** file. This extends the compiler to support code in extended program pages, and to provide for inter-page calls.

Warning: Do not include **`Bridge.f`** if you are not using paged program memory, because it will inhibit access to the base addresses in that region.

Paged program memory has base addresses in the range 8000-BFFF$_H$. During compilation, the host's 32-bit representation of target address space contains the **`PPAGE`** value in the upper 16 bits, and the base address in the lower 16 bits. Calls to the SwiftX kernel (which must *not* be in paged memory) and calls within a page are direct, but inter-page calls will automatically be compiled with additional instructions to manage the **`PPAGE`** register appropriately. Because of this overhead, you should take care to organize your program to place any time-critical code in the kernel page and generally minimize inter-page calls

To define code sections in extended program memory, simply add a leading digit to addresses in the range 8000-BFFF$_H$ for **`SECTION`**. Here's an example for a 64K PROM with the upper 16K page always accessible in the address space from C000$_H$ to FFFF$_H$ and the lower three pages mapped in the program page memory address space:

```
08000 0BFFF CDATA SECTION PAGE0
18000 1BFFF CDATA SECTION PAGE1
```

```
28000 2BFFF CDATA SECTION PAGE2
0C000 0FFFF CDATA SECTION PROG
```

PROG will contain the kernel, and **PAGE0** through **PAGE2** will reside in three program pages. Control of what code goes in which page is achieved by explicit use of the page name to select a current cData section, as the compiler always works in the current cData section.

After compiling the program, you must save your code pages explicitly, to the files corresponding to each physical target address region. For example:

```
PAGE0  SAVE-CODE TARGET0.S19    \ 0000-3FFF
PAGE1  SAVE-CODE TARGET1.S19    \ 4000-7FFF
PAGE2  SAVE-CODE TARGET2.S19    \ 8000-BFFF
PROG   SAVE-CODE TARGET.S19     \ C000-FFFF
```

3.3 I/O Registers

The 68HC12's I/O registers vary somewhat with each microcontroller version. Refer to Motorola's *Technical Summary* for your particular MCU for details regarding the use of these registers.

SwiftX defines names for the registers, corresponding to their Motorola designations, in a file for each supported MCU (these files have names that take the form **Swiftx\Src\68hc12\Reg_**<mcu>**.f**). An example for the MC68HC-812A4 is shown in Table 8. 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. Most are not used by the SwiftX kernel, and are available for program use.

Table 8: Internal registers in the 68HC812A4

Address	Name	Description
PARALLEL I/O PORT		
00	**PORTA**	Port A Data Register
01	**PORTB**	Port B Data Register
02	**DDRA**	Port A Data Direction Register

Table 8: Internal registers in the 68HC812A4 *(continued)*

Address	Name	Description
03	`DDRB`	Port B Data Direction Register
04	`PORTC`	Port C Data Register
06	`DDRC`	Port C Data Direction Register
08	`PORTE`	Port E Data Register
09	`DDRE`	Port E Data Direction Register
0A	`PEAR`	Port E Assignment Register
0B	`MODE`	Mode Register
0C	`PUCR`	Pull-up Control Register
0D	`RDRIV`	Reduced Drive of I/O Lines Register
10	`INITRM`	Initialization of Internal RAM Position Register
11	`INITRG`	Initialization of Internal Register Position Register
12	`INITEE`	Initialization of Internal EEPROM Position Register
13	`MISC`	Miscellaneous Mapping Control Register
REAL-TIME CLOCK		
14	`RTICTL`	Real Time Interrupt Control Register
15	`RTIFLG`	Real Time Interrupt Flag Register
COP WATCHDOG TIMER		
16	`COPCTL`	COP Control Register
17	`COPRST`	Arm/Reset COP Timer register
18	`ITST0`	
19	`ITST1`	
1A	`ITST2`	
1B	`ITST3`	
RESETS AND INTERRUPTS		
1E	`INTCR`	Interrupt Control Register
1F	`HPRIO`	Highest Priority Interrupt Register
KEY WAKEUP		
05	`PORTD`	Port D Data Register
07	`DDRD`	Port D Data Direction Register

Table 8: Internal registers in the 68HC812A4 *(continued)*

Address	Name	Description
20	`KWIED`	Key Wakeup Port D Interrupt Enable Register
21	`KWIFD`	Key Wakeup Port D Flag Register
24	`PORTH`	Port H Data Register
25	`DDRH`	Port H Data Direction Register
26	`KWIEH`	Key Wakeup Port H Interrupt Enable Register
27	`KWIFH`	Key Wakeup Port H Flag Register
28	`PORTJ`	Port J Data Register
29	`DDRJ`	Port J Data Direction Register
2A	`KWIEJ`	Key Wakeup Port J Interrupt Enable Register
2B	`KWIFJ`	Key Wakeup Port J Flag Register
2C	`KPOLJ`	Key Wakeup Port J Polarity Register
2D	`PUPSJ`	Key Wakeup Port J Pull-up/Pull-down Select Register
2E	`PULEJ`	Key Wakeup Port J Pull-up/Pull-down Enable Register
MEMORYEXPANSION		
30	`PORTF`	Port F Data Register
31	`PORTG`	Port G Data Register
32	`DDRF`	Port F Data Direction Register
33	`DDRG`	Port G Data Direction Register
34	`DPAGE`	Data Page Register
35	`PPAGE`	Program Page Register
36	`EPAGE`	Extra Page Register
37	`WINDEF`	Window Definition Register
38	`MXAR`	Memory Expansion Assignment Register
CHIP SELECT		
3C	`CSCTL0`	Chip Select Control Register 0
3D	`CSCTL1`	Chip Select Control Register 1
3E	`CSSTR0`	Chip Select Stretch Register 0

Table 8: Internal registers in the 68HC812A4 *(continued)*

Address	Name	Description
3F	`CSSTR1`	Chip Select Stretch Register 1
PHASE LOCK LOOP		
40	`LDVH`	Loop Divider Register High
41	`LDVL`	Loop Divider Register Low
42	`RDVH`	Reference Divider Register High
43	`RVDL`	Reference Divider Register Low
47	`CLKCTL`	Clock Control Register
ANALOG TO DIGITAL CONVERTER		
60	`ATDCTL0`	Reserved
61	`ATDCTL1`	Reserved
62	`ATDCTL2`	ATD Control Register 2
63	`ATDCTL3`	ATD Control Register 3
64	`ATDCTL4`	ATD Control Register 4
65	`ATDCTL5`	ATD Control Register 5
66	`ATDSTATH`	
67	`ATDSTATL`	
68	`ATDTESTH`	
69	`ATDTESTL`	
6F	`PORTAD`	Port AD Data Input Register
70	`ADR0H`	A/D Converter Result Register 0
72	`ADR1H`	A/D Converter Result Register 1
74	`ADR2H`	A/D Converter Result Register 2
76	`ADR3H`	A/D Converter Result Register 3
78	`ADR4H`	A/D Converter Result Register 4
7A	`ADR5H`	A/D Converter Result Register 5
7C	`ADR6H`	A/D Converter Result Register 6
7E	`ADR7H`	A/D Converter Result Register 7
STANDARD TIMER MODULE		
80	`TIOS`	Timer Input Capture/Output Compare Select

Table 8: Internal registers in the 68HC812A4 *(continued)*

Address	Name	Description
81	`CFORC`	Timer Compare Force Register
82	`OC7M`	Output Compare 7 Mask Register
83	`OC7D`	Output Compare 7 Data Register
84	`TCNTH`	Timer Counter Register Hi
85	`TCNTL`	Timer Counter Register Lo
86	`TSCR`	Timer System Control Register
87	`TQCR`	Reserved
88	`TCTL1`	Timer Control Register 1
89	`TCTL2`	Timer Control Register 2
8A	`TCTL3`	Timer Control Register 3
8B	`TCTL4`	Timer Control Register 4
8C	`TMSK1`	Timer Interrupt Mask Register 1
8D	`TMSK2`	Timer Interrupt Mask Register 2
8E	`TFLG1`	Timer Interrupt Flag Register 1
8F	`TFLG2`	Timer Interrupt Flag Register 2
90	`TC0H`	Timer Input Capture/Output Compare Reg. 0 Hi
91	`TC0L`	Timer Input Capture/Output Compare Reg. 0 Lo
92	`TC1H`	Timer Input Capture/Output Compare Reg. 1 Hi
93	`TC1L`	Timer Input Capture/Output Compare Reg. 1 Lo
94	`TC2H`	Timer Input Capture/Output Compare Reg. 2 Hi
95	`TC2L`	Timer Input Capture/Output Compare Reg. 2 Lo
96	`TC3H`	Timer Input Capture/Output Compare Reg. 3 Hi
97	`TC3L`	Timer Input Capture/Output Compare Reg. 3 Lo
98	`TC4H`	Timer Input Capture/Output Compare Reg. 4 Hi
99	`TC4L`	Timer Input Capture/Output Compare Reg. 4 Lo
9A	`TC5H`	Timer Input Capture/Output Compare Reg. 5 Hi
9B	`TC5L`	Timer Input Capture/Output Compare Reg. 5 Lo
9C	`TC6H`	Timer Input Capture/Output Compare Reg. 6 Hi
9D	`TC6L`	Timer Input Capture/Output Compare Reg. 6 Lo

Table 8: Internal registers in the 68HC812A4 *(continued)*

Address	Name	Description
9E	`TC7H`	Timer Input Capture/Output Compare Reg. 7 Hi
9F	`TC7L`	Timer Input Capture/Output Compare Reg. 7 Lo
A0	`PACTL`	Pulse Accumulator Control Register
A1	`PAFLG`	Pulse Accumulator Flag Register
A2	`PACNTH`	Pulse Accumulator Count Register High
A3	`PACNTL`	Pulse Accumulator Counter Register Low
AD	`TIMTST`	Timer Test Register
AE	`PORTT`	Timer Port T Data Register
AF	`DDRT`	Timer Port T Data Direction Register
SERIAL COMMUNICATION (SCI & SPI)		
C0	`SC0BDH`	SCI 0 Baud Rate Register High
C1	`SC0BDL`	SCI 0 Baud Rate Register Low
C2	`SC0CR1`	SCI 0 Control Register 1
C3	`SC0CR2`	SCI 0 Control Register 2
C4	`SC0SR1`	SCI 0 Status Register 1
C5	`SC0SR2`	SCI 0 Status Register 2
C6	`SC0DRH`	SCI 0 Data Register High
C7	`SC0DRL`	SCI 0 Data Register Low
C8	`SC1BDH`	SCI 1 Baud Rate Register High
C9	`SC1BDL`	SCI 1 Baud Rate Register Low
CA	`SC1CR1`	SCI 1 Control Register 1
CB	`SC1CR2`	SCI 1 Control Register 2
CC	`SC1SR1`	SCI 1 Status Register 1
CD	`SC1SR2`	SCI 1 Status Register 2
CE	`SC1DRH`	SCI 1 Data Register High
CF	`SC1DRL`	SCI 1 Data Register Low
D0	`SP0CR1`	SPI 0 Control Register 1
D1	`SP0CR2`	SPI 0 Control Register 2
D2	`SP0BR`	SPI 0 Baud Rate Register

Table 8: Internal registers in the 68HC812A4 *(continued)*

Address	Name	Description
D3	`SP0SR`	SPI 0 Status Register
D5	`SP0DR`	SPI 0 Data Register
D6	`PORTS`	Port S Data Register
D7	`DDRS`	Port S Data Direction Register
EEPROM		
F0	`EEMCR`	EEPROM Module Configuration Register
F1	`EEPROT`	EEPROM Block Protect
F2	`EETST`	EEPROM Test Register
F3	`EEPROG`	EEPROM Control Register

3.4 Timers

The system millisecond timer is implemented using the 68HC12 Real Time Interrupt (RTI) which is programmed here to use the CPU Eclock divided by 8192. For an 8 MHz EClock, this results in a rate of 1024 interrupts per second.

See Section 11.3 of the Motorola *MC68HC812A4 Technical Summary* (or the equivalent for other MCUs) for details about the RTI. The number of milliseconds is accumulated by the **`<RTI>`** interrupt handler in the variable **`MSECS`**.

`COUNTER` returns the low-order 16 bits of the current 32-bit free-running counter of clock interrupts. **`TIMER`**, always used after **`COUNTER`**, obtains a second count, subtracts the value left on the stack by **`COUNTER`**, then displays the elapsed time (in milliseconds) since **`COUNTER`**. **`COUNTER`** and **`TIMER`** may be used to time processes or the execution of commands. The usage is:

```
COUNTER <process or command to be timed> TIMER
```

The timer overflow (TOF) interrupt is used to accumulate the current date and time.

References Clock and timing libraries, *SwiftX Reference Manual*, Section 6.2

3.5 Serial Channel

The Multiple Serial Interface (MSI) is a component of the 68HC12 microcontroller. It provides high-speed communication to other devices. The MSI contains two sub-modules: the serial communications interface (SCI) and the serial peripheral interface (SPI).

The 68HC12 SwiftX system provides support for the SCI. The SCI is a full-duplex, universal asynchronous receiver transmitter (UART) interface, described in Section 14.2 of Motorola's *MC68HC812A4 Technical Summary*, and in the equivalent documents for other MCUs in this family. It is fully compatible with the SCI systems found in other Motorola MCUs, such as those of the M68K, M68HC11, and M68HC05 families.

The SCI provides two serial ports. SCI 0 is used for the Cross-Target Link (XTL), whose control is described in Section 4.9 of the *SwiftX Reference Manual*. SCI 1 is available for application use.

SwiftX support for a terminal task on SCI 1 is provided in the file **`..\68hc12\Sciterm.f`**, which includes **`EMIT`**, **`TYPE`**, **`KEY?`**, **`KEY`**, and **`ACCEPT`** behaviors for a task associated with the SCI 1 serial port. Rules for use:

1. Define a task for the serial port, as described in Section 5.4.1 of the *SwiftX Reference Manual*.
2. In your one-time initialization, **`CONSTRUCT`** the task and call **`/SCI1`** to initialize the port. Change the definition of **`K1(BAUD)`** for the desired initial baud rate.
3. In the task's start-up word, assign the **`SCI1-TERMINAL`** vectors and a terminal type (such as the "dumb terminal" in **`SwiftX\Src\Dumb.f`**).

If you write your code using the standard Forth versions of words like **`EMIT`**, **`TYPE`**, **`KEY?`**, **`KEY`**, and **`ACCEPT`**, they will be executed using the versions provided for the task executing them. That is, a word that does a **`TYPE`** will output to the host's SwiftX command window if it is executed by the XTL task, or will use the SCI 1 output words if it is executed by a task that was configured for that port by using the procedure outlined above.

An example is provided for running a multitasking version of the demo program included with SwiftX:

```
256 TERMINAL CONSOLE           \ Define a task

: DEMO   CONSOLE ACTIVATE      \ Start task
   DUMB SCI1-TERMINAL          \ Initialize task vectors
   BEGIN  CALCULATE  AGAIN ;   \ Run CALCULATE indefinitely
```

This defines a terminal task named **CONSOLE**. **DEMO** causes **CONSOLE** to initialize itself with the **DUMB** terminal type (providing versions of the vectored words **CR**, **TAB**, etc., for a simple ANSI terminal), sets the SCI 1 driver routines for **TYPE**, **KEY**, etc., and then leaves it in an infinite loop running the conical pile calculator demo program.

References SCI 1 serial driver, Section 4.3
Terminal tasks, *SwiftX Reference Manual*, Section 5.4
Multitasking demo, *SwiftX Reference Manual*, Section 5.5
Running the demo program, *SwiftX Reference Manual*, Section 1.4.3
SwiftOS multitasker implementation, Section 3.1.4

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 must 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.

   The on-board registers for your microcontroller are named in files whose names are **`\Swiftx\Src\68HC12\Reg_`**`<mcu>`, where *mcu* indicates your variant of the HC12 controller (e.g., **`Reg_a4.f`**). Special registers associated with other devices may be named at the beginning of the file containing the driver.

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. Memory-mapped registers can be read or written using **C@**, **C!**, **@**, **!**, etc. (depending on the width of the register), and the **.** ("dot") command can be used to display the results. (Usually you want the numeric base set to **HEX** when doing this!) For example, to look at the Port A data register, you could type:

   ```
   PORTA C@ .
   ```

   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 both 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 uses the timer overflow (TOF) interrupt to provide basic clock services. This provides a good example of a simple interrupt routine. The complete source may be found in **`Swiftx\Src\68HC12\Clock.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. Returning time of day is never as time-critical as servicing frequent clock ticks!

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

Our counter will be two cells, or 32 bits. With a millisecond tick rate, it will roll over every 49 days. This means it can be used to time intervals of up to 49 days, in addition to returning the time of day. The interrupt routine has to pick up the low-order cell and increment it; if it overflows, the high-order part must be incremented. The interrupt routine looks like this:

```
2VARIABLE TOCKS \ Holds the 32-bit TOF tick interrupt count.

LABEL <CLOCK>   \ TOF interrupt handler that increments TOCKS.
   TOCKS CELL+ LDX INX
   TOCKS CELL+ STX                 \ Increment low-order cell
   0= IF                           \ On overflow,
      TOCKS LDX INX TOCKS STX      \ ..increment high-order cell
   THEN
   TFLG2 $80 # BSET  RTI  END-CODE\ Set bit in Flag Register 2

\ Attach <CLOCK> to the timer overflow exception:
   <CLOCK>  V-TOF  EXCEPTION
```

4.3 Example: Terminal I/O to an LCD Display

Terminal I/O provides a somewhat 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.

Our mission in writing a serial driver is to provide the *device-layer versions* of these standard routines. As shipped, this version of SwiftX includes the output portion of a serial driver that communicates to an LCD display attached to the Axiom CMD12A4 board described in Appendix A.

Since Axiom provides a standard LCD display interface on most of the boards in their product line (many of which are supported by SwiftX), we have factored this driver into two files. Both are named **Display.f**. The layer that is generic to all Axiom boards contains no CPU-specific code or other features, and so is placed at the highest level of the **\SwiftX\Src** directory. The layer that is device-specific is in the lowest-level directory, for the board itself. Please refer to these files during the balance of this discussion.

Since writing to an LCD display is hardly a time-critical activity, we designed these layers to keep as much of the logic in the high-level file as possible.

The standard Axiom LCD display has (on all boards) a two-byte interface, consisting of a command register and a data register. Both are memory-mapped, but to addresses that depend on the target CPU and board design. So the high-level file requires just four functions from the low-level file: functions to read each register and write each register. These are called **@LCD-CMD**, **@LCD-DAT**, **!LCD-CMD**, and **!LCD-DAT**, respectively.

Using the command register functions, we define three control functions:

- **LCD-WAIT** waits until the busy bit is clear.
- **!LCD-WAIT** outputs a command and waits until not busy.
- **/LCD** initializes the LCD.

Then using these plus the data register read and write functions, we can define device-specific equivalents for **EMIT**, **TYPE**, **PAGE**, **AT-XY**, and **CR**. For example:

```
: (D-EMIT) ( char -- )   !LCD-DAT LCD-WAIT ;

: (D-TYPE) ( c-addr len -- )
   0 ?DO COUNT  (D-EMIT)  LOOP DROP ;
```

Since the device is interfaced through memory-mapped registers, even the low-level portion of the driver is extremely simple. All you have to do is name the registers, for readability, and then use **C@** and **C!** to read and write them. The entire code for this is:

```
$3F0 EQU LCD-CW        \ LCD command write
$3F1 EQU LCD-DW        \ LCD data write
$3F2 EQU LCD-CR        \ LCD command read
$3F3 EQU LCD-DR        \ LCD data read

: !LCD-CMD ( char -- )   LCD-CW C! ;
: !LCD-DAT ( char -- )   LCD-DW C! ;

: @LCD-CMD ( -- char)   LCD-CR C@ ;
: @LCD-DAT ( -- char)   LCD-DR C@ ;
```

4.4 Example: General Serial I/O

General serial I/O provides a still more complex example, including a full implementation of the Forth serial I/O API.

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.

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) directly connected to a serial port. You may find it interesting to compare them. They may be found in **Swiftx\Src\68HC12\Sci0xtl.f** and **..\Sci1term.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.

4.4.1 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 very simple word to output a single character:

```
CODE SPIT ( char -- )  2 ,Y+ SC1DRH MOVW  RTS  END-CODE
```

Such a word could be edited into a file, but it may be just as easy to type it at the keyboard, if your XTL is active. You can try it immediately:

```
 65 SPIT
```

This should send an A character to whatever is connected to SCI 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:

```
CODE SPITS ( char -- )
   BEGIN   SC1SR1 TST     \ Check for port ready
      0< UNTIL            \ Repeat till ready
   2 ,Y+ SC1DRH MOVW      \ Output character
   RTS   END-CODE
```

You can test this by putting it in a loop:

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

```
PAD 50 65 FILL              (puts 50 A characters at PAD)
PAD 50 GO                   (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:

```
CODE (S1-EMIT) ( char -- )      \ Output chars
   BEGIN   ' PAUSE JSR          \ PAUSE for multitasker
      SC1SR1 TST                \ Check for port ready
   0< UNTIL                     \ Repeat till ready
   2 ,Y+ SC1DRH MOVW            \ Output character
   RTS   END-CODE
```

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

4.4.2 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\68HC12\Sci1term.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 on the interface. So the input side of this driver will have a 100-byte buffer, plus four pointers used to manage the process. The buffer is called **SCI1-RQ** (SCI 1 Receive Queue). The pointers are defined as offsets into the buffer; the actual data begins at **SCI1-RQ** + 4. The layout is given in Table 9.

Table 9: Input queue pointers

Offset name	Value	Description
RIN	0	Offset for next received byte
ROUT	1	Offset to next byte to remove
RTASK	2	Task to be awakened
RDATA	4	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 **<SCI1>** (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:

```
<SCI1> V-SCI1 EXCEPTION
```

attaches the code to the exception vector for SCI 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 three code primitives, described in Table 10.

Table 10: 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-AWAIT)	*(—)*	Sets **RTASK** to the task executing this word, and checks the queue. If there are no characters, suspends the task. If there is at least one character, sets **RTASK** to **NOTASK** and returns.
(S1-READ)	*(— char)*	Returns the next character from the queue.

Although **(S1-KEY?)** is sufficient as a primitive for **KEY?**, both **(S1-AWAIT)** and **(S1-READ)** are required for **KEY**. This is because **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. The definition of **(S1-KEY)**, then, is:

```
: (S1-KEY) ( -- char)   (S1-AWAIT)  PAUSE  (S1-READ) ;
```

The task will be suspended in **(S1-AWAIT)** until a key is available, at which time it will **PAUSE** (ensuring that there is at least one **PAUSE**, even if a key was already in the queue) and then read the key.

4.4.3 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 **/SCI1** in the file **Swiftx\Src\68hc12\Sci1term.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\68hc12\**<board>**\Start.f**. **/SCI1** initializes the buffer pointers and sets a default baud rate and port control bits.

SCI1-TERMINAL in **Swiftx\Src\68hc12\Sci1term.f** is intended to be executed by a terminal task to set its vectored serial I/O functions to the device-layer versions for SCI 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\68hc12\**<board>**\Try.f**:

```
256 TERMINAL CONSOLE          \ Define a task

: DEMO   CONSOLE ACTIVATE     \ Start task
   DUMB SCI1-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 **SCI-TERMINAL**, and then enters an infinite loop performing the **CALCULATE** demo routine.

Appendix A: Axiom CMD12A4 Instructions

This section provides information pertaining to the Axiom CMD12A4 Instructions, which is supported by SwiftX for the 68HC12. 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

The CMD12A4 is a fully configured development system for the Motorola MC68HC812A4 microcontroller. The system is supplied with the PB68HC12A4 controller module, SwiftX kernel in the on-board EEPROM, 16K EEPROM, 16K RAM, serial cable, 9v 300ma wall plug, printed hardware manual, and the UTL12 software disk with assembler and support software.

Features include:

- PB68HC12A4 Controller Module (installed)
- One RS232 SCI with DB9 connection (COM 1)
- One RS232/485 SCI Port with DB9 connection (COM 2)
- 8- or 16-bit bus support with automatic bus size switching.
- LCD interface port, memory mapped (80 character maximum)
- Keypad Interface Port, 16 key
- SPI/Simple Serial Port
- Flexible I/O configuration to maximize use
- Four configurable 32-pin memory sockets for 32K to 2Mb ROM, and 32K to 512Kb RAM

- Bus expansion ports with control signals
- Large 2 × 7.5 inch prototyping area
- Efficient 6 to 30V DC input power supply
- 16MHz oscillator/8MHz bus
- Operating power: 120mA @ 5V

A.2 Demo Board Connections

We strongly advise you to set up and use the 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.

Installing SwiftX on this board requires only a small amount of effort, but proceed with care, because a mistake could damage the board.

Caution—Do not put the board on a conducting surface when power is applied to it.

1. Connect the serial cable and power supply as directed in the "Getting Started" section of the *CMD12A4 Development Board User's Manual*. Do not attempt to perform the "Terminal Window" tests mentioned in that book, because the Buffalo Monitor program has been replaced by the SwiftX kernel in the CMD12A4 board as supplied by FORTH, Inc.
2. Ensure that all MODE switches on the CMD12A4 board are set to OFF (except for switch 3, which should be ON if you require extended data space access).
3. The controller in the CMD12A4 has two on-board serial communication ports, SC0 and SC1. SwiftX uses SC0 for the XTL, attached to the host, and SC1 is available as a generic serial port. You may attach a terminal emulator window to this port, and test it using the "Conical Pile Calculator" demo supplied with your system (see Section A.3.3).

You are now ready to run your SwiftX software, as described in the following sections.

A.3 Development Procedures

On the Axiom CMD12A4 Instructions, the SwiftX kernel resides in EPROM. As a result, procedures for preparing and installing new kernels differ from those described in the generic SwiftX manual and from those for other 68HC12 systems that may be RAM-based.

A.3.1 Starting a Development Session

Debug

Launch SwiftX 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 selecting the Project > Debug menu item or Toolbar button. This completely compiles the kernel and compares it to the target's kernel in EEPROM.

If the source has changed, you will get the message, `Kernel Mismatch`, in which case you must generate a new code image and install it in the board's EEPROM as described in Section A.3.2.

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 and select Project > Debug again.

If the connection was successfully established and the host version of the kernel matches the EEPROM'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.3.2 Installing a New Kernel in EEPROM

Your Axiom CMD12A4 Instructions board is shipped with a SwiftX kernel installed in its on-board EEPROM. 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 the kernel in EEPROM.

Remember to enable the EEPROM write line by installing jumper position 10 of the ROM_SEL jumper block before programming the EEPROMs. You should remove this jumper to protect the EEPROMs while you are developing and testing code.

To install a new kernel:

Build

Run

1. Select the Project > Build menu item or Toolbar button. This generates a new, downloadable **`Target.s19`** object file.
2. Launch the Axiom Ax12 program, which may be found in Axiom's UTL12 utilities package, which is located in the **`Swiftx\Bin`** directory. You may use the Tools > Run menu item or toolbar button to do this.
3. Use the "Program Code Memory" command in Ax12 to install **`Target.s19`** in the target EEPROMs. (Consult the *CMD12A4 Development Board User's Manual* for detailed instructions.)
4. Disable the EEPROM write line by removing the jumper you installed in preparation for this procedure (see above), thus protecting the EEPROMs while you are developing and testing code.

A.3.3 Running the Demo Application

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

The demo program may use the host keyboard and screen via the XTL (the default configuration), or you may connect the CMD12A4's COM2 serial port to a COM port on your host, and define a separate task to talk to a standard terminal emulator utility in the PC, as described in Section 5.4 of the *SwiftX Reference Manual*.

Debug

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

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

A.4 Board-level Implementation Issues

This section describes features of the SwiftX implementation that are specific to the Axiom CMD12A4 Instructions.

A.4.1 System Hardware Configuration

The Axiom CMD12A4 board has a relatively complex memory layout, as shown in Figure 2 (page 48).

The SwiftX kernel runs in the EEPROM at 8000_H. As shipped, it occupies about 5K of this space. If the libraries this kernel is configured for are not all needed, they may be removed. A simple kernel can be made to fit in the 4K internal EEPROM; alternatively, there is ample space left in the 32K main program space for application development.

The kernel configures iData and uData spaces in internal RAM, in the file **`..\Cmd12a4\Config.f`**. The region between 2000_H and 7000_H is configured for additional iData, uData, and cData for development, in the file **`..\Cmd12a4\Try.f`**. You may adjust any of these to suit your needs.

The CMD12A4 board provides a 4K window into which extended memory may be mapped. Operators for accessing this memory are discussed in Section 3.2.

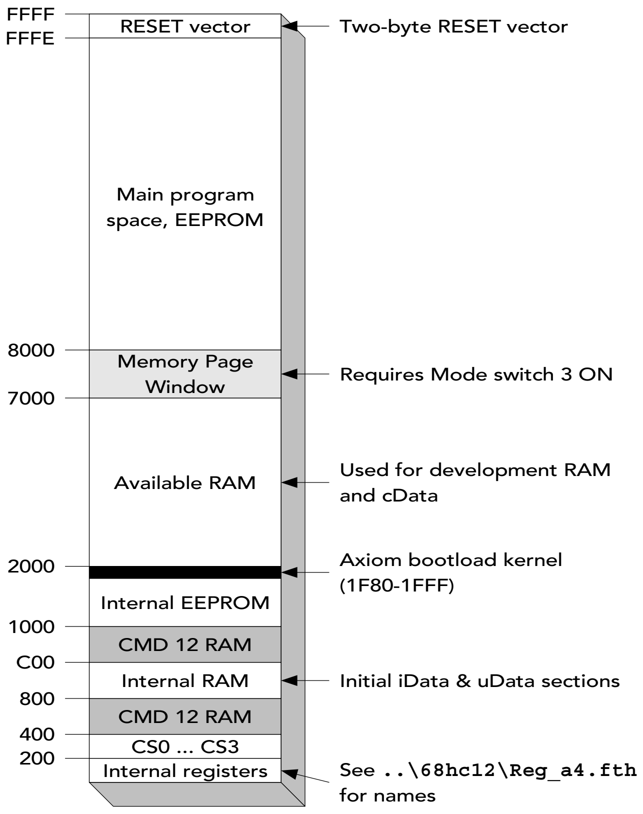
FFFF
FFFE
RESET vector
Two-byte RESET vector
Main program space, EEPROM
8000
Memory Page Window
Requires Mode switch 3 ON
7000
Available RAM
Used for development RAM and cData
2000
Axiom bootload kernel (1F80-1FFF)
Internal EEPROM
1000
CMD 12 RAM
C00
Internal RAM
Initial iData & uData sections
800
CMD 12 RAM
400
CS0 ... CS3
200
Internal registers
See ..\68hc12\Reg_a4.fth for names

Figure 2. Memory organization in the CMD12A4

APPENDIX B: CME12B32 BOARD INSTRUCTIONS

This section provides information pertaining to the Axiom CME12B32 board that is supported by SwiftX for the 68HC12. It includes instructions for setting up your board and connecting it to your host PC, as well as additional information pertinent to the implementation of the SwiftX kernel and SwiftOS on this board.

B.1 BOARD DESCRIPTION

The CME12B32 is a low-cost board provided by Axiom Manufacturing for the M68HC12B32 MCU, a variant of the 68HC12 family. The board provides the platform for interface and power connections to the M68HC12B32 MCU.

Features include:

- 64K Byte SRAM
- RS232 SCI with DB9 connection (SCI0)
- LCD Interface Port
- Keypad Interface Port, 16 Key
- SPI/Simple Serial Port
- Four configurable 32-pin memory sockets
- Bus expansion port, 16-bit data, with control
- All HC12 I/O to headers (MCU_PORT1 and 2)
- 16MHZ oscillator/8MHZ Bus
- 6 pin debug header
- RESET generator <4.5V
- Bootload Firmware - works with AX12

- Expanded or single chip operation w/MODA/MODB jumpers
- Operating Power: 120ma @ 5V w/on board regulator
- Board size 5.5 X 6.5 inches with Proto Area

B.2 MCU Specifications

The MC68HC912B32 microcontroller unit (MCU) is a 16-bit device composed of standard on-chip peripherals including a 16-bit central processing unit (CPU12), 32 Kbyte flash EEPROM, 1 Kbyte RAM, 768 byte EEPROM, an asynchronous serial communications interface (SCI), a serial peripheral interface (SPI), an 8-channel timer and 16-bit pulse accumulator, an 8-bit analog-to-digital converter (ADC), a four-channel pulse-width modulator (PWM), and a J1850-compatible byte data link communications module (BDLC). The chip is the first 16-bit microcontroller to include byte-erasable EEPROM and flash EEPROM on the same device. System resource mapping, clock generation, interrupt control and bus interfacing are managed by the Lite Integration Module (LIM). The MC68HC912B32 has full 16-bit data paths throughout, but the multiplexed external bus can operate in an 8-bit narrow mode, so single 8-bit wide memory can be interfaced for lower-cost systems.

Features include:

- 16-bit CPU12
- 20-bit ALU
- 32 Kbyte flash EEPROM with 2 Kbyte erase-protected boot block
- 768 byte EEPROM
- 1 Kbyte RAM with single-cycle access for aligned or misaligned read/write
- 8-channel, 8-bit analog-to-digital converter
- 8-channel timer
- Each channel fully configurable as either input capture or output compare
- Simple PWM mode
- Modulo reset of timer counter
- 16-bit pulse accumulator
- Asynchronous Serial Communications Interface (SCI)

- Synchronous Serial Peripheral Interface (SPI)
- J1850 byte data link communication (BDLC)
- COP watchdog timer, clock monitor, and periodic interrupt timer
- 80-pin QFP package
- Up to 63 general-purpose I/O lines
- 2.7V–5.5V operation at 8 MHz
- Single-wire Background Debug™ Mode (BDM)
- On-chip hardware breakpoints

B.3 Demo Board Connections

We strongly advise you to set up and use the 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.

If you purchased the board with your SwiftX system, it will be shipped with all jumpers set correctly for SwiftX and with SwiftX installed in the flash EEPROM on the board. The following instructions are for reference, or to help if you are porting SwiftX from a different 68HC12 board.

Installing SwiftX on this test board requires only a small amount of effort, but proceed with care, because a mistake could damage the board.

Caution—Do not put the board on a conducting surface when power is applied to it.

1. Connect the serial cable to the COM-1 port. This is used for the SwiftX XTL during development.
2. Set jumpers JP1, JP2, JP3 all *on*, and VPP_EN *off*. The MEM-SEL jumpers 1,3,4 should be *on* (32K EEPROM in U6/7).
3. Connect the power cord and transformer as indicated.

You are now ready to run your SwiftX software, as described in the following sections.

B.4 Development Procedures

On the CME12B32, the SwiftX kernel resides in flash EEPROM. As a result, procedures for preparing and installing new kernels differ somewhat from those described in the generic SwiftX manual and from other 68HC12 systems that may be RAM- or PROM-based.

B.4.1 Starting a Development Session

Debug

Launch SwiftX as described in Section 1.3.1. You may change any options you wish and, when you are ready, you may compile your kernel by selecting Project > Debug from the menu. This completely compiles the kernel and compares it to the target's kernel in EEPROM. If the source has changed, you will get the message, `Kernel Mismatch`, in which case you must generate a new code image and program it into the board's flash EEPROM (see Section B.4.2 below).

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 and select Project > Debug again.

If the connection was successfully established, 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*.

B.4.2 Installing a New Kernel in EEPROM

Your board is shipped with a SwiftX kernel installed in its on-board flash EEPROM. In order for SwiftX's Cross-Target Link (XTL) to work, 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 the kernel in EEPROM.

Remember to enable the EEPROM write line by installing the VPP_EN jumper before programming the EEPROMs. You should remove this jumper to protect the EEPROMs while you are developing and testing code.

To install a new kernel:

Build

Run

1. Select the Project > Build menu item or Toolbar button. This generates a new, downloadable **`Target.s19`** object file.
2. Launch the Axiom Ax12 program, which may be found in Axiom's UTL12 utilities package, which is located in the **`Swiftx\Bin`** directory. You may use the Tools > Run menu item or toolbar button to do this.
3. Use the "Program Code Memory" command in Ax12 to install **`Target.s19`** in the target EEPROMs.
4. Disable the EEPROM write line by removing the jumper you installed in preparation for this procedure (see above), thus protecting the EEPROMs while you are developing and testing code.

B.4.3 Running the Demo Application

As shipped, the interactive testing program **`Debug.f`** (loaded by the Project > Debug menu item) is configured to run the demo application described in the *SwiftX Reference Manual*. The demo program uses the host keyboard and screen via the XTL.

To run this program, follow the instructions given in Section 1.4.3 of your *SwiftX Reference Manual*.

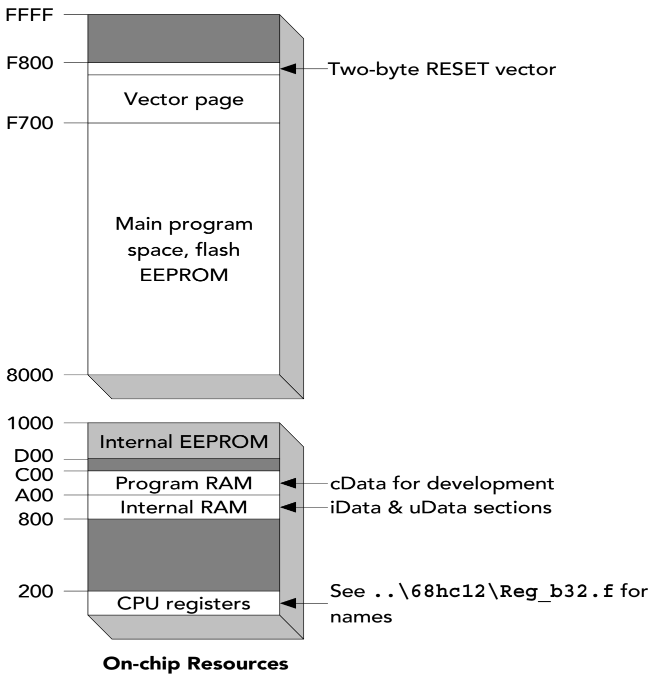

Figure 3. Memory organization in the CME12B32

B.5 System Hardware Configuration

The EVB's memory layout is shown in Figure 3.

The SwiftX kernel runs in the EEPROM at 8000_H. As shipped, it occupies about 4K of this space. If the libraries this kernel is configured for are not all needed, they may be removed. A stripped kernel can be made to fit in the 768-byte internal EEPROM; alternatively, there is ample space left in the 32K main program space for application development. The kernel configures iData and uData spaces in internal RAM, in the file **`..\Config.f`**.

The controller in the EVB has one on-board serial communication port, SC0. SwiftX uses SC0 for XTL communication with the host.

Appendix C: Motorola EVB Instructions

This section provides information pertaining to the Motorola Evaluation Board, M68EVB912B32 (sometimes referred to here simply as the "EVB") that is supported by SwiftX for the 68HC12. It includes instructions for setting up your board and connecting it to your host PC, as well as additional information pertinent to the implementation of the SwiftX kernel and SwiftOS on this board.

C.1 Board Description

The M68EVB912B32 is a low-cost evaluation board provided by Motorola for the M68HC12B32 MCU, a variant of the 68HC12 family. The board consists of a 5.15 × 3.4 inch (13.1 × 8.64 cm) double-sided printed circuit board that provides the platform for interface and power connections to the M68HC12B32 MCU.

Features include:

- Double-sided PCB
- Single-supply +3 to +5 Vdc power input (P5)
- One RS232 interface
- Header footprints for access to all MCU pins
- 16 MHz crystal for 8 MHz bus operation
- Headers for jumper selection of, and connection to, hardware options
- Four 2 × 20 header connectors for access to the MCU I/O lines (P2, P3, P4, and P6)
- Prototype expansion area for customized interfacing with the MCU
- Low-profile reset push-button switch
- Low-voltage inhibit protection

C.2 MCU Specifications

The MC68HC912B32 microcontroller unit (MCU) is a 16-bit device composed of standard on-chip peripherals including a 16-bit central processing unit (CPU12), 32 Kbyte flash EEPROM, 1 Kbyte RAM, 768 byte EEPROM, an asynchronous serial communications interface (SCI), a serial peripheral interface (SPI), an 8-channel timer and 16-bit pulse accumulator, an eight-bit analog-to-digital converter (ADC), a four-channel pulse-width modulator (PWM), and a J1850-compatible byte data link communications module (BDLC). The chip is the first 16-bit microcontroller to include byte-erasable EEPROM and flash EEPROM on the same device. System resource mapping, clock generation, interrupt control and bus interfacing are managed by the Lite integration module (LIM). The MC68HC912B32 has full 16-bit data paths throughout, but the multiplexed external bus can operate in an eight-bit narrow mode, so single eight-bit wide memory can be interfaced for lower-cost systems.

Features include:

- 16-bit CPU12
- 20-bit ALU
- 32 Kbyte flash EEPROM with 2 Kbyte erase-protected boot block
- 768 byte EEPROM
- 1 Kbyte RAM with single-cycle access for aligned or misaligned read/write
- 8-channel, 8-bit analog-to-digital converter
- 8-channel timer
- Each channel fully configurable as either input capture or output compare
- Simple PWM mode
- Modulo reset of timer counter
- 16-bit pulse accumulator
- Asynchronous Serial Communications Interface (SCI)
- Synchronous Serial Peripheral Interface (SPI)
- J1850 byte data link communication (BDLC)
- COP watchdog timer, clock monitor, and periodic interrupt timer
- 80-pin QFP package
- Up to 63 general-purpose I/O lines

- 2.7V–5.5V operation at 8 MHz
- Single-wire Background Debug™ Mode (BDM)
- On-chip hardware breakpoints

C.3 Demo Board Connections

We strongly advise you to set up and use the 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.

Installing SwiftX on this test board requires only a small amount of effort, but proceed with care, because a mistake could damage the board.

Caution—Do not put the board on a conducting surface when power is applied to it.

1. Use the serial cable supplied with the board to connect the EVB port P1 to your PC's serial port.
2. Ensure that jumpers W3 and W4 on the EVB are set to the 0 position.
3. Apply power (ground and +5V) to P5.

You are now ready to run your SwiftX software, as described in the following sections.

C.4 Development Procedures

On the EVB, the SwiftX kernel resides in flash EEPROM. As a result, procedures for preparing and installing new kernels differ somewhat from those described in the generic SwiftX manual and from other 68HC12 systems that may be RAM-based.

C.4.1 Starting a Development Session

Debug

Launch SwiftX as described in Section 1.3.1. You may change any options you wish and, when you are ready, you may compile your kernel by selecting the Project > Debug menu item or Toolbar button. This completely compiles the kernel and compares it to the target's kernel in EEPROM. If the source has changed, you will get the message, `Kernel Mismatch`, in which case you must generate a new code image and program it into the board's flash EEPROM (see Section C.4.2 below).

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 and select Project > Debug again.

If the connection was successfully established, 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*.

C.4.2 Installing a New Kernel in EEPROM

Your EVB is shipped with a SwiftX kernel installed in its on-board flash EEPROM. In order for SwiftX's Cross-Target Link (XTL) to work, 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 the kernel in EEPROM.

To install a new kernel:

Build

1. Select the Project > Build menu item or Toolbar button. This generates a new, downloadable **`Target.s19`** object file. At that point, exit SwiftX.

Run

2. Launch a terminal emulator program such as Microsoft Terminal or HyperTerminal. (Sample Terminal and HyperTerminal configuration files are provided in the **`...\68HC12\B32EVB`** directory.) Line settings should be 9600 baud, 8 bits, no parity. You may use the Tools > Run menu item or Toolbar button to do this.

Be sure that +12 volts is supplied to the board via the Vpp pin on W8. Move both jumpers W3 and W4 to the ″1″ position and press the reset button to enter the bootload mode. The board should display the following command prompt:

```
(E)rase, (P)rogram or (L)oadEE:
```

3. Type `E` to erase the flash EEPROM. The board sends the message `Erased` followed by the command prompt when it is ready.
4. In the terminal program, specify a short delay (500 milliseconds) for each line sent to the board, or pace transmission on the receipt of the `*` (asterisk) character, to allow each record to be programmed into flash EEPROM.
5. Type `P` to program the flash memory, and download the new **`Target.s19`** file to the board as a text file. The board echoes one `*` after each record is programmed. When it is done, the board says `Programmed` and once again displays the command prompt.
6. Move W3 and W4 back to the “0” position and press reset.
7. Exit the terminal emulator before launching SwiftX for testing.

C.4.3 Running the Demo Application

Debug

As shipped, the interactive testing program **`Debug.f`** (loaded by the Project > Debug menu item or Toolbar button) is configured to run the demo application described in the *SwiftX Reference Manual*. The demo program uses the host keyboard and screen via the XTL.

To run this program, see Section 1.4.3 of the *SwiftX Reference Manual*.

C.5 System Hardware Configuration

The EVB's memory layout is shown in Figure 4.

The SwiftX kernel runs in the EEPROM at 8000_H. As shipped, it occupies about 4K of this space. If the libraries this kernel is configured for are not all needed, they may be removed. A stripped kernel can be made to fit in the 768-byte internal EEPROM; alternatively, there is ample space left in the 32K main program space for application development.

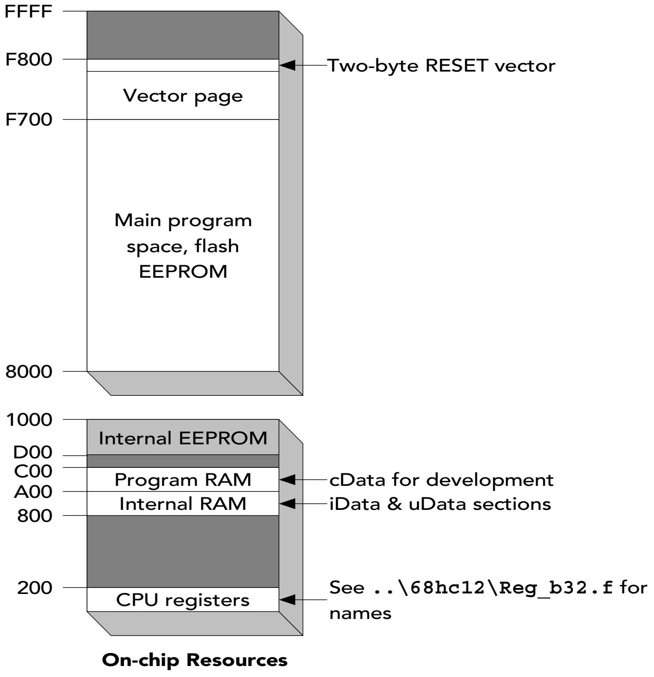

Figure 4. Memory organization in the M68EVB912B32

The kernel configures iData and uData spaces in internal RAM, in the file `..\Config.f`.

The controller in the EVB has one on-board serial communication port, SC0. SwiftX uses SC0 for XTL communication with the host.

General Index