Getting Results Faster...

SwiftX HC11

Board-level Documentation for 68HC11 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

Second edition, July 1999
Printed 7/9/99

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 68HC11 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 68HC11; 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 68HC11. Refer to Motorola's documentation for information about the 68HC11, 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
New Micros NMIX-0022	Appendix A: NMIX-0022 Board Instructions	39
Axiom CMD 11A8	Appendix B: Axiom CMD11A8 Board Instructions	45

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.

   If you need to uninstall SwiftX, use the "Add/Remove Programs" utility available under Start > Settings > Control Panel.

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 68HC11 ASSEMBLER

The M68HC11 CPU is optimized for low power consumption and high-performance operation at bus frequencies up to 4 MHz. Features include:

- Two 8-bit or one 16-bit accumulator
- Two 16-bit index registers
- Powerful bit-manipulation instructions
- Six powerful addressing modes
  (Immediate, Extended, Indexed, Inherent, and Relative)
- Power-saving STOP and WAIT modes
- Memory-mapped I/O and special functions
- 16 × 16 integer and fractional divides
- 8 × 8 multiply

The architecture of the M68HC11 CPU causes all peripheral, I/O, and memory locations to be treated identically as locations in the 64 Kbyte memory map. Thus, there are no special instructions for I/O that are separate from those used for memory. This technique is sometimes called *memory-mapped I/O*. In addition, there is no execution-time penalty for accessing an operand from an external memory location, as opposed to a location within the MCU.

Throughout this book, we assume you understand the hardware and functional characteristics of the 68HC11 as described in the *Motorola M68HC11 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 **LDD** 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 cross-compiler provides an assembler for the 68HC11 processor. The mnemonics for the 68HC11 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 3 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 )   TPUSH     \ Push current TOS
   U LDY   S0 ,U LDD   0 ,X STX\ Get S0 from user area
   0 ,X SUBD   1 # SUBD   LSRD  \ Subtract current stack addr
   RTS   END-CODE               \ Return depth in TOS
```

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 4.

2.3 REGISTERS

68HC11 registers, shown in Figure 1, are an integral part of the CPU and, unlike I/O registers, are not addressed as if they were memory locations.

- **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 an 8-bit offset, which is included as part of the instruction, to form the effective address of the operand to be used in the instruction. The exchange instructions **XGDX** and **XGDY** offer a very simple way to load an index value into the 16-bit accumulator D.
- **Stack pointer (SP)** points to the last stack location used. The 68HC11 supports an automatic program stack that is used to save system context during subroutine calls and interrupts; it can also be used for temporary storage of data.

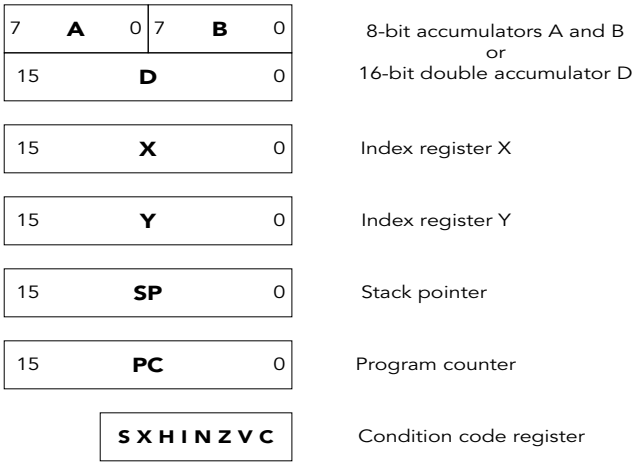

Figure 1. 68HC11 registers

- **Program counter** is a 16-bit register that holds the address of the next instruction to be executed.
- **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 **X** is used as the Forth data stack pointer, and **SP** is the return stack pointer. Index register **Y** is always available as a scratch register, without saving or restoring.

  Register **D** contains the top-of-stack (TOS) value, which is convenient since it's so often needed in a register. The actual top stack location addressed by **0 ,X** is always vacant, so if you need **D** for another purpose it's easily saved. For example, the code generated by **DUP** is:

```
0 ,X STD    DEX    DEX
```

  This stores the contents of **D** (the top stack item) in the location referenced by **X**, and decrements **X** by two bytes (one cell). Since the value remains in **D**, the

effect is to have two copies of it. Had the value not already been in **D**, it would have been necessary to fetch it, costing one more instruction.

The 68HC11 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 colon definitions.

References Internal I/O registers, Section 3.3

2.4 Addressing Modes

The notation for specifying addressing modes differs from common assembler notation, in that the mode specifiers are operands that precede the mnemonics. In this assembler (as in most Forth assemblers), 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 principle addressing modes of the 68HC11, with examples.

Table 2: Addressing modes

Mode	Example	Description
Immediate	`-1 # LDD`	Load register **D** with -1.
Direct	`SCSR TST`	Test internal register **SCSR**.
Extended	`MSECS LDD`	Load the cell at **MSECS** into Register **D**.
Indexed by **X**	`2 ,X LDY`	Load the second stack item into **Y**.
Indexed by **Y**	`4 ,Y LDD`	Load **Y**+4 (2 cells) into Register **D**.
Relative	`FOO BRA`	Branch to location **FOO**.
Inherent	`ABA`	Add accumulator B to accumulator A.

2.5 Direct Transfers

In Forth, most direct transfers are performed using structures (such as those described in Section 2.6) and **LABEL**s. 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**). The mnemonics **EJMP** and **EJSR** may be used when the extended forms are required (such as in building assembler routine dispatch tables).

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>  THEN
<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 12. The combination of a condition code and a structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional branch instruction **B**cc, 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:

```
TSTA   0< IF
```

executes the true branch of the **IF** structure contents if accumulator **A** is negative.

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 3 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 3: 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.
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 glossaries below, the stack notation *cc* refers to a condition code. Available condition codes are listed in the glossary that begins on page 12.

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.

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 the *SwiftX Reference Manual*'s Section 4.

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 <RTI>
   MSECS CELL+ LDD   TC2 # ADDD   MSECS CELL+ STD
   MSECS LDD   TC1 # ADCB   TC0 # ADCA   MSECS STD
   $40 # LDAA   TFLG2 STAA   RTI   END-CODE

<TICK> V-RTI EXCEPTION
```

<RTI> 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 **TFLG2**.

Table 4 lists the interrupt vectors that are named in SwiftX (see **Swiftx\Src\68hc11\Ev-ram.f**) and their respective offsets from the start of the vector area.

Table 4: Named interrupt vectors

Offset (hex)	Name	Description
00FE	**V-RESET**	Reset
00FC	**V-CLOCK**	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-TIC1**	Timer input capture 1 interrupt
00EC	**V-TIC2**	Timer input capture 2 interrupt
00EA	**V-TIC3**	Timer input capture 3 interrupt
00E8	**V-TOC1**	Timer output capture 1 interrupt
00E6	**V-TOC2**	Timer output capture 2 interrupt
00E4	**V-TOC3**	Timer output capture 3 interrupt
00E2	**V-TOC4**	Timer output capture 4 interrupt

Table 4: Named interrupt vectors *(continued)*

Offset (hex)	Name	Description
00E0	**V-TIC4/OC5**	Timer IC4/OC5 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-SCI**	SCI 0 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\68hc11**<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 68HC11 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, both for 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 68HC11'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 3.1.2.

You may see examples of SwiftX 68HC11 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:

```
BFE6   >R JSR                          BDB8E3
BFE9   SWAP JSR                        BDB868
BFEC   R@ JSR                          BDB90A
BFEF   + JSR                           BDB970
BFF2   SWAP JSR                        BDB868
BFF5   R> JSR                          BDB8EE
BFF8   - JMP                           7EB975 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** or **BSR** (depending upon the length of the branch) 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:

```
B888   SWAP BSR                        8DDE
B88A   OVER BRA                        20E4 ok
```

Although **SWAP** is called with a **BSR**, **OVER** is called with a **BRA**, because it will use the **RTS** at the end of **OVER** to return to its caller.

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 the definition of **DIGIT**, which converts a small binary number to a digit:

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

With the optimizer turned off, you would get:

```
SEE DIGIT
D292   0 ,X STD                        ED00
D294   DEX                             09
D295   DEX                             09
D296   0 ,X STD                        ED00
D298   DEX                             09
```

```
D299   DEX                          09
D29A   9 # LDD                      CC0009
D29D   > JSR                        BDB9E7
D2A0   (IF) JSR                     BDB80A
D2A3   D2A8 BNE                     2603
D2A5   D2B2 JMP                     7ED2B2
D2A8   0 ,X STD                     ED00
D2AA   DEX                          09
D2AB   DEX                          09
D2AC   7 # LDD                      CC0007
D2AF   + JSR                        BDB970
D2B2   0 ,X STD                     ED00
D2B4   DEX                          09
D2B5   DEX                          09
D2B6   30 # LDD                     CC0030
D2B9   + JMP                        7EB970 ok
```

But the optimized version in your kernel is:

```
SEE DIGIT
C1B6   0 ,X STD                     ED00
C1B8   DEX                          09
C1B9   DEX                          09
C1BA   0 ,X STD                     ED00
C1BC   DEX                          09
C1BD   DEX                          09
C1BE   9 # LDD                      CC0009
C1C1   > JSR                        BDB9E7
C1C4   (IF) JSR                     BDB80A
C1C7   C1CC BNE                     2603
C1C9   C1CF JMP                     7EC1CF
C1CC   7 # ADDD                     C30007
C1CF   30 # ADDD                    C30030
C1D2   RTS                          39 ok
```

This code is not only smaller (28 bytes vs. 39), but it's significantly faster, since it's optimized the conditional and handling of literals, eliminating the two calls to **+**. The compiler will attempt to optimize where it can do so without increasing program size.

To experiment with this further, follow this procedure:

1. Turn off the optimizer, by typing:

   ```
   -OPTIMIZER
   ```

2. Type in a definition.
3. Decompile it, using **SEE** <name>.
4. Turn the optimizer back on, by typing:

   ```
   +OPTIMIZER
   ```

5. Re-enter your definition.

Tip: You can re-enter the previous definition by pressing your up-arrow key until you see the desired line, then press Enter to re-enter it.

6. Decompile it and compare.

You can see the rules it uses by typing:

```
HOST .RULES TARGET
```

3.1.2 Data Format and Memory Access

Because the 68HC11 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, if any.

The high byte of a 16-bit cell on the 68HC11 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 from high RAM. The return stack is the CPU's subroutine stack, and 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 68HC11 supports an efficient SwiftOS implementation, with seven instructions required to deactivate a task and ten to activate one. The subroutine-threaded implementation means there is no **I** register (see address interpreter, Section 4.3 of the *SwiftX Reference Manual*) to be saved and restored, only the return and data stack pointers.

The four-byte **STATUS** area contains either a **NOP** (01) or an indexed **JSR** (AD_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** ($7E_H$) opcode, **WAKE** decodes to **7E , 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 5: SwiftOS user status instructions

Name	Value (hex)	Instruction	Description
WAKE	AD7E	**JSR**	Call to wake-up code pointed to by **X**.
SLEEP	017E	**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\68hc11\Tasker.f**.

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

3.2 Internal EEPROM Access

SwiftX supplies simple primitives for access to the internal EEPROM at $B600_H$–$B7FF_H$. These words are described in the glossary below, and may be found in the file **..\68HC11\Eeprom.f**. Each of the "**!**" operators is analogous to the normal memory access operator, except that it performs the necessary actions to program this EEPROM. This memory may be read with normal fetch operators (**@**, **C@**, **MOVE**, etc.)

As shipped, SwiftX makes no use of this EEPROM space, leaving it available for application use.

Glossary

EEC! *(b c-addr —)*

Store a byte at *c-addr* in EEPROM. The byte is erased only if necessary.

EE! *(x a-addr —)*

Store a cell at a-*addr* in EEPROM. The cell is erased only if necessary. Note that *a-addr* must be cell-aligned (even).

EES! *(c-addr$_1$ c-addr$_2$ u —)*

Write the string at *c-addr$_1$*, of length *u*, to *c-addr$_2$* in EEPROM.

EE-BULK-ERASE *(—)*

Erases the entire contents of the internal EEPROM.

3.3 I/O Registers

The 68HC11'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\68hc11\Reg_<mcu>.f`**.) An example for the MC68HC11A8 is shown in Table 6, where the offsets shown are with respect to **REGBASE**, defined in the file **`Config.f`**. 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 6: Internal registers in the 68HC11A8

Offset	Name	Description
00	**PORTA**	Port A data register
02	**PIOC**	Parallel I/O control register
03	**PORTC**	Port C data register
04	**PORTB**	Port B data register

Table 6: Internal registers in the 68HC11A8 *(continued)*

Offset	Name	Description
05	**PORTCL**	Port C latched data register
07	**DDRC**	Port C data direction register
08	**PORTD**	Port D data register
09	**DDRD**	Port D data direction register
0A	**PORTE**	Port E data register
0B	**CFORC**	Timer compare force register
0C	**OC1M**	Output compare 1 mask register
0D	**OC1D**	Output compare 1 data register
0E	**TCNT**	Timer counter register
10	**TIC1**	Timer input capture register 1
12	**TIC2**	Timer input capture register 2
14	**TIC3**	Timer input capture register 3
16	**TOC1**	Timer output compare register 1
18	**TOC2**	Timer output compare register 2
1A	**TOC3**	Timer output compare register 3
1C	**TOC4**	Timer output compare register 4
1E	**TOC5**	Timer output compare register 5
20	**TCTL1**	Timer control register 1
21	**TCTL2**	Timer control register 2
22	**TMSK1**	Main timer interrupt mask register 1
23	**TFLG1**	Main timer interrupt flag register 1
24	**TMSK2**	Misc. timer interrupt mask register 2
25	**TFLG2**	Misc. timer interrupt flag register 2
26	**PACTL**	Pulse accumulator control register
27	**PACNT**	Pulse accumulator count register
28	**SPCR**	SPI control register
29	**SPSR**	SPI status register
2A	**SPDR**	SPI data register
2B	**BAUD**	Baud rate control register

Table 6: Internal registers in the 68HC11A8 *(continued)*

Offset	Name	Description
2C	**SCCR1**	SCI control register 1
2D	**SCCR2**	SCI control register 2
2E	**SCSR**	SCI status register
2F	**SCDR**	SCI data register
30	**ADCTL**	A/D control/status register
31	**ADR1**	A/D result register 1
32	**ADR2**	A/D result register 2
33	**ADR3**	A/D result register 3
34	**ADR4**	A/D result register 4
39	**OPTION**	System configuration options
3A	**COPRST**	Arm/reset COP timer circuitry
3B	**PPROG**	EEPROM programming register
3C	**HPRIO**	Highest priority interrupt and misc.
3D	**INIT**	RAM and I/O mapping register
3E	**TEST1**	Factory test register
3F	**CONFIG**	Configuration control register

3.4 TIMERS

The system millisecond timer is implemented using the 68HC11 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 Motorola's *Technical Summary* for details about the RTI. The number of milliseconds is accumulated by the **<RTI>** interrupt handler in the variable **MSECS**.

COUNTER returns the current value of a 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 5.2

3.5 SERIAL CHANNEL

The Multiple Serial Interface (MSI) is a component of the 68HC11 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 68HC11 SwiftX system provides support for the SCI. The SCI is a full-duplex, universal asynchronous receiver transmitter (UART) interface, described in Section 9 of Motorola's *M68HC11 Reference Manual*, 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, M68HC12, and M68HC05 families.

The SCI is used for the Cross-Target Link (XTL), whose control is described in Section 3.9 of the *SwiftX Reference Manual*.

References ACIA serial driver on CMD11A8 board, Section B.4.2
Terminal tasks, *SwiftX Reference Manual*, Section 4.7
Multitasking demo, *SwiftX Reference Manual*, Section 4.8
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\68hc11\Reg_**<mcu>, where *mcu* is the particular variant of 68HC11 controller (e.g., **Reg_a8.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 find out 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. The occurrence of 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\68HC11\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 from 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 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 <TOF>   \ 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
   $80 # LDAA   TFLG2 STAA        \ Set TOF bit in register
   RTI  END-CODE                  \ End

\ Attach <TOF> to the timer overflow exception:
   <TOF>  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 CMD11A8 board described in Appendix B.

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:

```
$B5F0 EQU LCD-CW        \ LCD command write
$B5F1 EQU LCD-DW        \ LCD data write
$B5F0 EQU LCD-CR        \ LCD command read
$B5F1 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: 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 that contains **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 on the Axiom board. You may find it interesting to compare them.

They may be found in the files **Swiftx\Src\68HC11\Scixtl.f** and **..\Cmd11a8.ACIATerm.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 Named Registers

The Axiom's second serial port is on an R65C51 ACIA chip on the CMD11A8 board. It has several registers which, like other 68HC11 I/O, are memory mapped. These registers start at $B5F8_H$. To improve code readability, we will give them names, as follows:

```
TARGET

$B5F8 EQU 'ACIA

INTERPRETER

: ACIA: ( n -- )   'ACIA + EQU ;

TARGET

0 ACIA: A-RDR      \ ACIA Receiver Data Register
0 ACIA: A-TDR      \ ACIA Transmit Data Register
1 ACIA: A-SR       \ ACIA Status Register
2 ACIA: A-CMD      \ ACIA Command Register
3 ACIA: A-CTRL     \ ACIA Control Register
```

This is a good example of the use of the cross-compiler scopes described in the *SwiftX Reference Manual*. The interpreter word **`ACIA:`** is a defining word that builds named addresses, given an offset from the base address provided by **`'ACIA`**. The defined register names return their address; you can read and write them using **`C@`** and **`C!`** (as well as code instructions).

4.4.2 Polled Serial Output

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

We can test the port easily, by writing a very simple word to output a single character:

```
CODE SPIT ( char -- )   A-TDR STAB   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 the ACIA (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
      A-SR LDAA  $10 # ANDA   \ Check for port ready
   0= NOT UNTIL               \ Repeat till ready
   A-TDR STAB                 \ Output character
   ' DROP JMP                 \ Discard character
   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. Because 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 (A-EMIT) ( char -- )     \ Output chars
   BEGIN   ' PAUSE JSR        \ PAUSE for multitasker
      A-SR LDAA  $10 # ANDA   \ Check for port ready
   0= NOT UNTIL               \ Repeat till ready
   A-TDR STAB                 \ Output character
   ' DROP JMP                 \ Discard character
   END-CODE
```

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

```
: (A-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

4.4.3 Interrupt-driven Queued Serial Input

Input is somewhat more complex than output. Rather than reproduce the entire code here, refer to the file **..\68HC11\Cmd11a8\ACIAterm.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 256-byte buffer, plus four pointers used to manage the process. The buffer is called **ACIA-RQ** (ACIA Receive Queue). The pointers are defined as offsets into the buffer; the actual data begins at **ACIA-RQ** + 4. The layout is given in Table 7.

Table 7: 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 **<ACIA>** (just as the convention of names in parentheses is used to indicate the low-level components of high-level functions, the con-

vention of names in angled brackets is used to indicate interrupt routines). The phrase:

```
<ACIA> V-IRQ EXCEPTION
```

attaches the code to the exception vector for the IRQ exception handler.

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 in task-level code. There are three code primitives, described in the glossary at the end of this section.

Although **(A-KEY?)** is sufficient as a primitive for **KEY?**, both **(A-AWAIT)** and **(A-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 **(A-KEY)**, then, is:

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

The task will be suspended in **(A-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.

Glossary

(A-KEY?) *(— flag)*
Return a flag that is *true* if a character has been received. The primitive for **KEY?**.

(A-AWAIT) *(—)*
Set **RTASK** to the task executing this word, and check the queue. If there are no characters, suspend the task. If there is at least one character, set **RTASK** to **NOTASK** and return.

(A-READ) *(— char)*
Return the next character from the queue.

4.4.4 Port and Task Initialization

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

The word **/ACIA** in the file **Swiftx\Src\68hc11\Cmd11a8\Aciaterm.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\68hc11\Cmd11a8\Start.f**. **/ACIA** initializes the buffer pointers and sets a default baud rate and port control bits.

ACIA-TERMINAL in **Swiftx\Src\68hc11\Cmd11a8\Aciaterm** is intended to be executed by a terminal task to set its vectored serial I/O functions to the device-layer versions for ACIA. 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\68hc11**<board>**\Debug.f**:

```
256 TERMINAL CONSOLE          \ Define a task

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

APPENDIX A: NMIX-0022 BOARD INSTRUCTIONS

This section provides information pertaining to the NMIX-0022, which is supported by SwiftX for the 68HC11. 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 NMIX-0022 is a fully configured development system for the Motorola MC68HC11A8 microcontroller. The system is supplied with the SwiftX kernel in the on-board PROM, 8K RAM, a DB9 serial cable, 9v 200ma wall plug, and a printed hardware manual.

Features include:

- 5 parallel ports
- 1 asynchronous serial channel RS-232,422, or 485
- 1 synchronous serial channel, TTL
- 8-channel, 8-bit A/D
- 8-bit counter
- 16-bit timer
- 3 input captures
- 5 output compares
- .5K EEPROM
- 8K RAM chip included
- 64K address space
- Three 28-pin JEDEC memory sockets

- Flexible address decoding, socket assignments
- On-board Power Supply circuits: 7-18VAC input
- Battery backup circuits for memory
- 34-pin JEDSTACK™ Vertical Stacking Connector (VSC)

A.2 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 power supply to the terminal strip on the NMIX-0022.
2. Connect a communications cable between the serial port on your PC and the serial port on the NMIX-0022. A serial cable that plugs directly into a 25-pin PC COM port is supplied by New Micros. A standard 25-to-9 pin adapter may be required if your COM ports have DB-9 connectors.
3. Apply power to the NMIX-0022.

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

A.3 Development Procedures

On the NMIX-0022, the SwiftX kernel resides in PROM. As a result, procedures for preparing and installing new kernels may differ from those described in the generic SwiftX manual and from other 68HC11 systems.

A.3.1 Starting a Debugging 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 target system by selecting Project > Debug from the menu. This completely compiles the kernel and compares it to the target's system in PROM.

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 PROM 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 system 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 system matches the PROM's system, 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 EPROM

Your NMIX-0022 board is shipped with a SwiftX kernel installed in its onboard EPROM. In order for SwiftX's Cross-Target Link (XTL) to work prop-

erly, the object generated by your kernel source must exactly match the installed kernel. Therefore, if you make any changes to your kernel, you must replace this kernel.


Build

To install a new kernel,

1. Select the Project > Build menu item or Toolbar button—this generates a new **`Target.s19`** object file. Exit from SwiftX.


Run

2. Use a PROM programmer utility of your choice to burn a new PROM using this file. You may use the Tools > Run menu item or Toolbar button to do this.
3. Turn off power to the board. Change the PROM, then restore power to the board.
4. Launch SwiftX and continue as described in Section A.3.1.

A.3.3 Running the Demo Application

As shipped, the interactive testing program **`Debug.f`** 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 only, as the NMIX-0022 does not provide a second serial port.


Debug

To run this program, launch SwiftX as described in Section A.3.1 and select Project > Debug to bring up the target program. Type **`CALCULATE`** to start the application using the XTL.

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

A.4 HARDWARE CONFIGURATION

This sections describes features of the SwiftX implementation that are specific to the NMIX-0022.

The memory layout of the NMIX-0022 board is shown in Figure 2.

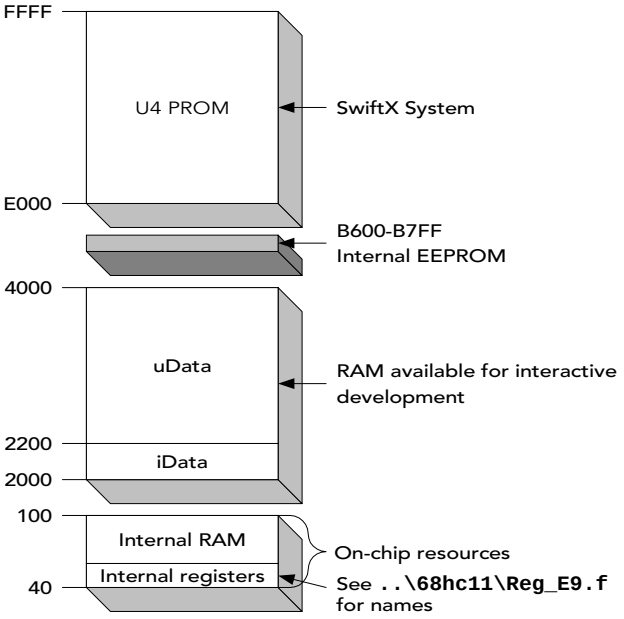

Figure 2. Memory organization in the NMIX-0022

The SwiftX system runs in the EPROM at $E000_H$. As shipped, it occupies about 6K of this space. If the libraries this kernel is configured for are not all needed, they may be removed. Some of the uData space is reserved as cData for application development.

SwiftX uses only 6 bytes of internal RAM, leaving the remainder for application use.

APPENDIX B: AXIOM CMD11A8 BOARD INSTRUCTIONS

This section provides information pertaining to the Axiom CMD11A8, which is supported by SwiftX for the 68HC11. 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.

B.1 BOARD DESCRIPTION

The Axiom CMD11A8 is a fully configured development system for the Motorola MC68HC11A8 microcontroller. The system is supplied with the SwiftX kernel in the on-board EEPROM, a blank 32K EEPROM, 32K RAM, a DB9 serial cable, 9v 200ma wall plug, printed hardware manual, and the UTL11 utility package of support software.

Features include:

- MC68HC11A8 CPU
- 512-byte EEPROM
- 16 BIT timer with 3 capture, 5 compare
- Pulse counter
- 8 channel 8 BIT AID
- SPI serial port
- SCI serial port
- Two serial ports with DB9 connectors:
  - COMl HC11 SCI extensions
  - COM2 65C51 ACIA (UART with RS232/485/422)
- Six I/0 ports:

  - HC11 port A, D, E
  - 82C55 AUX ports A, B, C
  - 44 I/0 total, 29 configurable, 11 inputs, 4 outputs
- Three configurable memory sockets:
  - 2-pin JEDEC 8K/32K byte RAM, EPROM, EEPROM
- Programmable clock — output 0.5 Hz to 16 kHz
- Keypad interface
- Keyboard/SPI interface
- LCD module interface with contrast adjust
- Regulated power supply
- Bus expansion port with 7 chip selects
- All I/O connectorized
- 120 × 60mm prototype area with connections
- Optional features:
  - Real-time clock RF5C15
  - Battery backup circuit
  - Alarm output

Specifications:

- Board size 120 × 160mm (5" x 6")
- Power input: +7 to +15V standard
- Current consumption: 100ma standard, 30ma special

B.2 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 pro-

ceed 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 *CMD11A8 Development Board User's Manual*. Do not attempt to perform the "Terminal" tests mentioned in that book, because the Buffalo Monitor program has been replaced by the SwiftX kernel in the CMD11A8 board as supplied by FORTH, Inc.
2. Of jumper positions JP1 through JP13, only these should be installed: JP3, JP4, JP6, and JP7. Jumper RX_SEL should be in position 1-2 for RS232 operation of COM2. See the manual if your application requires RS422 or RS485 operation.
3. The CMD11A8 has two serial communication ports, COM1 and COM2. SwiftX uses COM1 for the XTL, attached to the host, and COM2 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 B.3.3).

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

B.3 Development Procedures

On the Axiom CMD11A8, the SwiftX kernel resides in EEPROM. As a result, procedures for preparing and installing new kernels may differ from those described in the generic SwiftX manual and from those for other 68HC11 systems.

B.3.1 Starting a Debugging 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 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 install it in the board's EEPROM as described in Section B.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*.

B.3.2 Installing a New Kernel in EEPROM

Your Axiom CMD11A8 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 this kernel.

Remember to enable the EEPROM write line by installing JP6 and JP10 before programming the EEPROMs. You should remove these jumpers to protect the EEPROMs while you are developing and testing code.


Build

To install a new kernel:

1. Select the Project > Build menu item or Toolbar button. This generates a new **`Target.s19`** object file.


Run

2. Launch the Axiom Ax11 program (installed as part of the UTL11 utilities package; see Section B.2). You may use the Tools > Run menu item or Toolbar button to do this.
3. Use the Ax11 "Program Code Memory" command to install **`Target.s19`** in the target EEPROMs. (Consult the *CMD11A8 Development Board User's Manual* for detailed instructions.) Ax11 will prompt you for the filename to download. Enter the full path to your **`Target.s19`** (or use the Directory button to browse for it). Follow the directions as you proceed to download the system to the board's EEPROM.
4. Restore jumpers as directed by Ax11 when the code memory has been programmed. Press the reset button on the board, and exit the Ax11 utility.
5. Return to your SwiftX command window. You will have to reload Debug to resume interactive testing.

B.3.3 Running the Demo Application


Debug

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

The demo program may use the host keyboard and screen via the XTL (the default configuration), or you may connect the CMD11A8's COM2 serial port to a COM port on your host. **`Debug.f`** defines a task on the board's COM2 (see Section B.4.2). If you will be using this port, you must ensure that its cable keeps DCD high in order for the board to be able to receive input.

To run the demo program, select Project > Debug to bring up the target program. The demo is automatically started on COM2 when Debug is launched. Type **`CALCULATE`** to start the application if you want it to use the XTL.

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

B.4 Board-level Implementation Issues

This sections describes features of the SwiftX implementation that are specific to the Axiom CMD11A8.

B.4.1 System Hardware Configuration

The memory layout of the Axiom CMD11A8 board is shown in Figure 3.

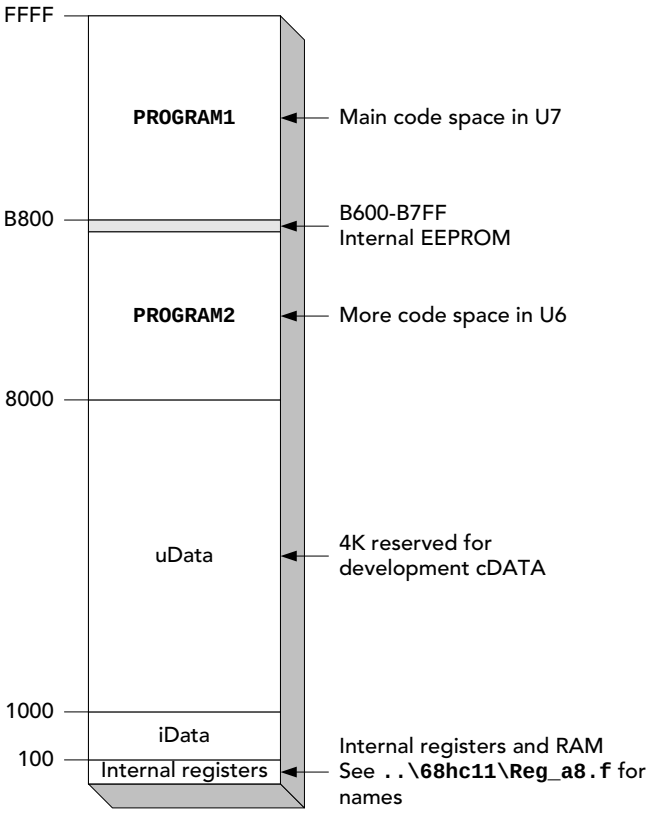

Figure 3. Memory organization in the Axiom CMD11A8

The SwiftX system runs in the EEPROM at $E000_H$. As shipped, it occupies about 5K of this space. There is ample room in the 28K RAM space for application development. SwiftX uses only 6 bytes of internal RAM; the rest is available for application use.

The kernel configures iData in internal RAM (in the **..\Cmd11a8\Config.f** file). The region between 1000_H and $7FFF_H$ is configured for uData, and a cData section is allocated from this space, between 1000_H and $1FFF_H$, for development (in the file **..\Cmd11a8\Debug.f**). You may adjust this, if necessary, to suit your needs.

B.4.2 ACIA Port Support

SwiftX support for a terminal task on COM2 (a 65C51 ACIA chip) is provided in the file **..\68hc11\Aciaterm.f**, which includes **EMIT**, **TYPE**, **KEY?**, **KEY**, and **ACCEPT** behaviors for a task associated with this port. Rules for use:

1. Define a task for the serial port, as described in Section 4.7.1 of the *SwiftX Reference Manual*.
2. In your one-time initialization, **CONSTRUCT** the task and call **/ACIA** to initialize the port. Change the definition of **(BAUD)** for the desired initial baud rate.
3. In the task's start-up word, assign the **ACIA-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 using the host's SwiftX command window if it is executed by the XTL task, and will use the ACIA output words if it is executed by a task that was configured for that port 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 ACIA-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 ACIA driver routines for **TYPE**, **KEY**, etc., and then leaves it in an infinite loop running the conical pile calculator demo program.

GENERAL INDEX