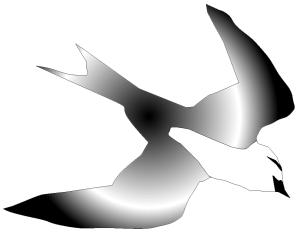

Getting Results Faster...

SwiftX 8051

Board-level Documentation for 8051-Family Targets

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

First edition, February 1998
Printed 7/14/98

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

CONTENTS

List of Figures

List of Tables

Welcome!

Important Information in This Book!

This book is designed to accompany all SwiftX 8051 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 8051; 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 8051. Refer to Intel's documentation for information about the 8051, 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 CME-0562	Appendix A: Axiom CME-562 Board Instructions	33
Micromint BCC52	Appendix B: BCC52 Board Instructions	39

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 8051 ASSEMBLER

The best-known member of Intel's MCS-51 family of microprocessors is the 8051, and its architecture is supported by numerous additional manufacturers, including Philips, Siemens, Dallas, and others. There are many variants but, in general, all execute the same instructions and vary only by the amount of internal memory and which I/O devices they support. For convenience, we will refer to the 8051 only.

This section supplements, but does not replace, the CPU manufacturer's manuals. Departures from Intel's usage are noted here; nonetheless, you should use the manufacturer's manuals for a detailed description of specific instructions.

Where **`boldface`** type is used here, it distinguishes Forth words (such as register names) from Intel names. Usually these are the same; the name **ADDC** can be used as a Forth word and as Intel's name. Where boldface is *not* used, the name refers to Intel's usage or 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 8051 processor. The mnemonics for the 8051 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 10 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>  RET  END-CODE
```

For example:

```
CODE DEPTH ( -- n)   ' DUP CALL \ Save previous TOS
   'S0 # A MOV   R0 A SUB       \ Calculate depth
   C CLR   A RRC   A R3 MOV     \ Convert from cells to count
   0 # R2 MOV   RET   END-CODE
```

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

```
WAIT JMP
```

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 **RET**) at the end of code that initiates an I/O operation, where the interrupt that signals completion of the I/O will be used to wake up the task.

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

References Interrupt handling, Section 3.3
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 4

2.3 REGISTERS

The 8051's Special Function Registers (SFRs) are defined as constants for use by the SwiftX assembler, using the published Intel names. **DPH** and **DPL** may be referred to collectively as the 16-bit register **DPTR**. In addition, the eight registers **R0** through **R7** can be accessed by certain instructions. These are in banks selected by two bank-select bits in the **PSW**.

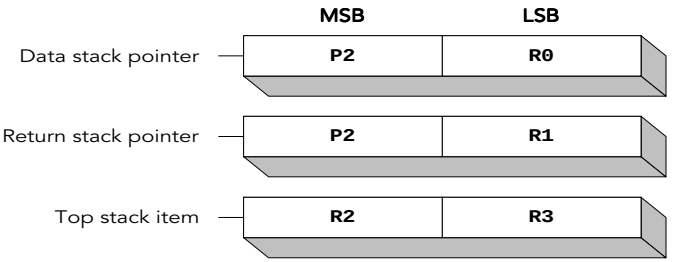

Figure 1. Register usage in 8051 SwiftX

Register **R0** is Forth's data stack pointer, and **R1** is the return stack pointer. **P2** holds the upper half (page) of the stack addresses. In addition, the top stack item is kept in **R2:R3** for convenience. You may not use **P2** and **R0**–**R3** for any purpose other than these dedicated functions without saving and restoring them. **R4** through **R7** are designated as scratch registers that can be used without saving and restoring.

The 8051's processor stack pointer **SP** is used for subroutine calls. Since SwiftX is a *subroutine-threaded* implementation, actual returns are handled through the processor stack, and return addresses are not actually kept on Forth's return stack.

2.4 ADDRESSING MODES

The notation for specifying addressing modes differs from Intel's notation, in that the mode specifiers are operands that precede the mnemonics. In the

SwiftX assembler, these mnemonics are words that actually assemble the instruction (using parameters left on the stack by the mode operands). Note

Table 2: Addressing modes

Mode	Example	Description
Immediate	**5 # B MOV**	Move 5 (immediate) to **B**
Direct	**B R1 MOV**	Move **B** to **R1**
Accumulator	**R1 A MOV**	Move **R1** to **A**
Indirect	**@R1 A MOV**	Move **R1** (indirect) to **A**
External	**A @DPTR MOVX**	Move **A** to the address in **DPTR**
Relative1	**@A+DPTR A MOVC**	Move data from the address in **DPTR** plus **A** to **A**
Relative2	**@A+PC A MOVC**	Move a code byte from the PC plus **A** to **A**
Boolean	**ACC .5 C MOV**	Move bit 5 from **ACC** to **C** flag (see below for the syntactic distinction between **ACC** and **A**)

that the syntax is consistently <source> <destination> <opcode>.

The accumulator may be referenced by two names, **ACC** or **A**. You should only use **ACC** when you are referring to it directly as a Special Function Register (which is bit-addressable). Use **A** to specify the register as source or destination in instructions that use it inherently. For example, you must say:

```
R1 A MOV
```

but:

```
ACC 7 LDB
```

Often there are two forms, and either syntax will work. In those cases, the **A** form usually assembles an instruction that is smaller and quicker.

2.5 MNEMONICS

The mnemonics of the various 8051 opcodes have been defined as words which, when executed, assemble the corresponding opcode at the next location in the program space. As with other Forth words, the operands (e.g., register numbers or names, ports, immediate data, and modifiers) must precede the mnemonic.

The mnemonic **MOV** requires two addresses as operands: a source followed by a destination. These may be registers or other addressing modes. Thus:

```
R1 A MOV
```

transfers the contents of the byte register **R1** to the accumulator. Similarly,

```
@R1 A MOV
```

moves the internal RAM data pointed to by **R1** to the accumulator.

2.6 MACROS

The macros in Table 3 have been defined in order to simplify assembler coding.

Table 3: Simple macros

Command	Action
SUB	Works exactly as **SUBB**, but clears the carry flag first.
ROL, **ROR**	Rotate left or right, respectively, through the carry. These work in the same way as **RLC** and **RRC**, but the operand is an internal address rather than the accumulator.
PSH	Push a specified register onto the data stack.
PUL	Pop the data stack into a specified register.
PSHR	Push a specified register onto the return stack.
PULR	Pop the return stack into a specified register.
TPUSH	Push the top stack item (TOS) onto the data stack (equivalent to **DUP**).

Table 3: Simple macros *(continued)*

Command	Action
TPOP	Pop the top stack item (TOS) from the data stack (equivalent to **DROP**).
TPUSHR	Push the top stack item onto the return stack.
TPOPR	Pop the top stack item from the return stack.

2.7 Renamed Mnemonics

In most cases, SwiftX uses Intel mnemonics and notation, but in postfix order. This section documents the few exceptions to this practice.

To specify a bit in instructions that access individual bits, SwiftX uses the syntax:

```
<reg> <bit> <opcode>
```

where *reg* is a bit-addressable register or SFR, and *bit* is **.0** through **.7**.

Table 4: Bit specifiers in SwiftX

Intel	SwiftX Assembler	Remarks
CPL P1.2	**P1 .2 CPL**	Note the space before the bit designation.
ANL C,/P1.0	**P1 .0 NOT C ANL**	**NOT** used instead of / notation.

A few of Intel'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. Table 5 summarizes these equivalencies; all SwiftX assembler condition codes are presumed to be followed by **IF**, **UNTIL**, or **WHILE**.

The **NOT** used in Table 5 inverts a condition code, but it *cannot* be used with **=**. If you need **= NOT IF** you must use **= IF ELSE** instead.

In addition, the mnemonics **JMP** and **CALL** are provided. These use the short form (**AJMP**, **SJMP**, or **ACALL**) when possible, and **LJMP** or **LCALL** otherwise.

Table 5: Conditional jump and branch equivalencies

Intel	SwiftX Assembler	Description
JC	**CS NOT**	Branch if the carry bit is set.
JNC	**CS**	Branch if the carry bit is not set.
JNZ	**0=**	Branch if A is non-zero.
JZ	**0= NOT**	Branch if A equal zero.
CJNE	**=**	Branch if operands not equal.
SJMP	**NEVER**	Unconditional branch (equivalent to **AGAIN**).
JB	<reg> <bit> **NOT IF**	Branch if *bit* is not set.
JNB	<reg> <bit> **IF**	Branch if *bit* is set.
JBC	<reg> <bit> **TAC IF**	Test and clear *bit*; branch if it was not set.

2.8 Assembler Structures

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

Forth assemblers in general, and 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 13. 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:

```
@DPTR A MOVX   0= NOT IF
```

executes the true branch of the **IF** structure if accumulator **A** is non-zero.

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. Those 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).

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 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 non-zero.

CS *(— cc)*

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

NEVER *(— cc)*

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

NOT *(cc_1 — cc_2)*

Invert the condition code cc_1 to give cc_2.

2.9 DIRECT TRANSFERS

In Forth, most 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 labels, arbitrary 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, there are times when direct transfers are useful, particularly when compactness of the compiled code overrides all other criteria. **CALL** and **JMP** are defined as described in Intel documentation, except that at compile time they automatically choose the smallest and fastest form that is compatible with the given addresses.

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, you will get a `Range error` message when the code compiles. Control structures that span routines are not recommended, because they make the source code harder to understand and harder to modify.

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

```
LABEL <name>
```

described in Section 2.2. **LABEL** does not effect the target dictionary. Invoking *name* returns the address identified by the label, which may be used as a destination for a **JMP** or a **CALL**.

For example, in the code for the serial XTL, we find this sequence:

```
LABEL OUT  \ output char from A with serial interrupt off
   A SBUF MOV            \ Output character
   BEGIN  SCON .1 UNTIL  \ Wait for TI
   SCON .1 CLR           \ Clear TI
   RET   END-CODE
```

This is invoked whenever the code needs to output one character, using:

```
OUT CALL
```

2.10 Assembly Language Macros

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

A SwiftX assembly language macro is a colon definition defined in the **ASSEMBLER** *scope, whose contents include assembler commands.*

The only complication lies in the fact that SwiftX assembler commands are not normally available in the **TARGET** scope (see "Compiler Scoping," *SwiftX Reference Manual,* Section 3.6.2). This is necessary because there are assembler versions of **IF**, **WHILE**, and other words that have very different meanings in high-level Forth. When you use **CODE** or **LABEL** to start a code definition, those words automatically select the assembler search order, and **END-CODE**

restores the previous search order. However, to make macros, you will need to manipulate search orders more directly.

The relevant commands for manipulating vocabularies for assembler macros are given in the glossary at the end of this section.

Example 1: Push a register onto the stack

```
ASSEMBLER
: PSH ( r)    \ Push A onto the data stack
 DUP A - [+HOST] IF [PREVIOUS] \ Is the register A?
     DUP A MOV                 \ If not, put it in A
  [+HOST] THEN DROP [PREVIOUS]
  R0 DEC   A @R0 MOVX ;        \ Push stack
```

This is one of several macros included with SwiftX to facilitate stack operations. Here it was necessary to invoke the **HOST** versions of **IF** and **THEN**, rather than the versions we would have gotten in the **ASSEMBLER** scope.

Example 2: Higher-level macro

```
: TPUSH                   \ Push top-of-stack
   R2 PSH   R3 PSH ;
```

This example uses the macro from Example 1 twice, to push the top stack item kept in register pair **R2:R3** onto the external data stack, making room for a new top stack item (or a **DUP**, if you stop here!).

3. IMPLEMENTATION ISSUES

This section covers specific implementation issues involving various versions of the 8051 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. The 8051's subroutine stack, instead of Forth's return stack, is used by target primitives to contain return addresses.

If you depend on the return stack to be separate from the subroutine stack, your code will not be portable to systems which combine 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 8051 optimization strategies by decompiling some simple definitions. For example, the source definition for **2OVER** is:

```
: 2OVER ( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)
   2>R 2DUP  2R> 2SWAP ;
```

but if you decompile it, you get:

```
021B   2>R ACALL                    31C2
021D   2DUP ACALL                   315E
021F   2R> ACALL                    31DB
0221   2SWAP AJMP                   2175 ok
```

This example shows the subroutine calls in this implementation.

When the last thing in a definition is a subroutine reference, the compiler automatically uses **SJMP**, **AJMP**, or **LJMP** (depending upon the destination of the branch) to save the subroutine return.

If you would like to study these implementation strategies, we encourage you to look at the file **Core.f**.

3.1.2 Data Format and Memory Access

The 8051 is an eight-bit processor, implemented with a 16-bit cell size, meaning that all addresses, stack items, and single-precision numbers are 16 bits wide. The 8051 is a *Harvard architecture* machine, meaning that it has potentially 64K bytes each of directly addressable code and data space. The appendix concerning the board that accompanies your system includes a memory map showing memory usage.

The general registers **R0**–**R7** can be mapped to 0–7, 8–F_H, 10–17_H, or 18–$1F_H$. SwiftX sets the select bits in **PSW** to 0 at power up, but the code does not rely on **R0**–**R7** being mapped to 0–7; registers should always be referenced by name.

Register memory from 20–$3F_H$ is all bit addressable, and SwiftX leaves it all available for application use. The rest of directly addressable internal RAM is 40–$7F_H$, and this is allocated for internal variables.

Indirectly addressable internal RAM (which hides behind the SFRs from 80–FF_H) is used for the subroutine stack on 8052s and compatible devices. If you don't have such a processor, you can modify **Config.f** to use less space for **INT-DATA**, and allow at least 32 bytes for data stack.

Other aspects of memory organization may be found in the appendix for each board supported.

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 separate from the CPU's subroutine stack, which carries return addresses for nested calls. A program may use the return stack for temporary storage during the execution of a definition, however for compatibility with implementation in which return addresses *are* on the return stack, the following restrictions should be respected:

- 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**.
- While 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 8051 supports an adequate SwiftOS implementation. 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.

Whenever a task is suspended, that task's subroutine stack is copied to its return stack. When a task gets control, the subroutine stack is restored from the saved copy. The user areas are in data space, so the contents of **STATUS** is merely a flag which is tested by the code in the inner loop of **PAUSE**.

The three-byte **STATUS** area contains either a **WAKE** (1) or **SLEEP** (0) in the first byte. The remainder is the address of the next task in the round robin.

The task's **STATUS** byte controls task behavior. For example, **PAUSE** sets it to **WAKE** and suspends 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 awakened, its **STATUS**

is set to **SLEEP** as part of the start-up process.

You can find the code for this SwiftOS multitasker in the file **Swiftx\Src\8051\Tasker.f**.

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

3.2 Special Function Registers

The 8051's Special Function Registers (SFRs) vary somewhat with each microcontroller version. Refer to the manufacturer's data sheet for your particular MCU for details regarding the use of these registers.

SwiftX defines names for the registers, corresponding to their Intel designations, in a file for each supported MCU. (These files have names that take the form **Swiftx\Src\8051\Reg_<mcu>.f**.) An example for the 8051/52 is shown in Table 6. These registers may be referenced by these names in code. Most are not used by the SwiftX kernel, and are available for program use.

Table 6: Special Function Registers on the 8051 and 8052

Address	Name	Bit addr	Description
080	**P0**	Y	Port 0
081	**SP**		Stack pointer
082	**DPL**		Data pointer, low byte
083	**DPH**		Data pointer, high byte
087	**PCON**		Power control
088	**TCON**	Y	Timer/counter control
089	**TMOD**		Timer/counter mode control
08A	**TL0**		Timer/counter 0, low byte
08B	**TL1**		Timer/counter 1, low byte
08C	**TH0**		Timer/counter 0, high byte
08D	**TH1**		Timer/counter 1, high byte
090	**P1**	Y	Port 1

Table 6: Special Function Registers on the 8051 and 8052 *(continued)*

Address	Name	Bit addr	Description
098	**SCON**	Y	Serial control
099	**SBUF**		Serial data buffer
0A0	**P2**	Y	Port 2
0A8	**IE**	Y	Interrupt enable
0B0	**P3**	Y	Port 3
0B8	**IP**	Y	Interrupt priority control
0C8	**T2CON**	Y	Timer/counter 2 control
0CA	**RCAP2L**		T/C 2 capture register, low byte
0CB	**RCAP2H**		T/C 2 capture register, high byte
0CC	**TL2**		Timer/counter 2, low byte
0CD	**TH2**		Timer/counter 2, high byte
0D0	**PSW**	Y	Program status word
0E0	**ACC**	Y	Accumulator (A register)
0F0	**B**	Y	B register

The fact that SFRs can only be referenced in code can make interactive debugging difficult. Fortunately it is easy to make simple code words that provide high-level access to an SFR of interest, following these examples:

```
CODE @P1 ( -- char )  \ Read char from P1
   TPUSH  0 # R2 MOV   P1 R3 MOV
   RET  END-CODE

CODE !P1 ( char -- )  \ Store char in P1
   R3 P1 MOV   ' DROP JMP
   END-CODE
```

3.3 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 pro-

cessor interrupt.

The handler itself is written in code. The usual form begins with **LABEL** <name> and ends with an **IRET** (Interrupt Return) and **END-CODE**. (**CODE** is not needed, because such routines are not invoked as subroutines.)

To attach the code to the handler, use the word **INTERRUPT**, which takes an address for the handler and a vector assignment ($3–7B_H$), and links them such that when the interrupt occurs, the accumulator **A** and **PSW** will be saved, and the handler will be invoked. The **IRET** at the end of the handler is a macro that restores the saved registers and returns from the interrupt.

The vector assignments vary for the different members of the 8051 family. The file **\8051\Reg_**<mcu>**.f** provides names for the versions supported by SwiftX as shipped. These should be used in preference to literal numbers, to improve maintainability. Consult your MPU reference manual for the vector assignments for your processor.

Except for saving and restoring the two registers, no other 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.

An example of an interrupt routine is given in Section 4.2.

The definition of the vectors and their associated initialization can be found in **\Swiftx\Src\8051\Vectors.f**. This file must be loaded first in the kernel, as the vectors must go at the start of the program image (i.e., location 0). The system power-up code is in the word **POWER-UP**, found in the file **\8051**<platform>**\Start.f**. **POWER-UP** is assigned as the destination of the **LJMP** in the first vector (**ENTRY**, in **Vectors.f**).

Glossary

INTERRUPT ($addr_1$ $addr_2$ —)

Install $addr_1$ in the interrupt vector at $addr_2$.

IRET *(—)*

Macro, used at the end of an interrupt handler, that assembles code to pop the registers pushed by the code in the vector table and return from the interrupt.

3.4 TIMERS

The system millisecond timer and system date/time are maintained using the timer 0 overflow (**TF0**) interrupt.

See Intel's *Technical Summary* for details about timer 0. The number of milliseconds is accumulated by the **<TF0>** 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
```

References Clock and timing libraries, *SwiftX Reference Manual*, Section 5.2
Timer 0 interrupt routine analyzed, Section 4.2

3.5 SERIAL CHANNEL

The 8051 built-in asynchronous serial port is used as the SwiftX Cross-Target Link (XTL), described in Section 3.9 of the *SwiftX Reference Manual*.

The driver for this port may be found in **Src\8051\Serial.f**, and may be used as an example of an application using the serial port.

References Terminal tasks, *SwiftX Reference Manual*, Section 4.7
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 the specific device, efficiently reusing the common code.

   The on-board registers for your microcontroller are named in files whose names are **\Swiftx\Src\8051\Reg_**<mcu>, where *mcu* is the particular 8051 variant (e.g., **Reg_51.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 learn 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!) Unfortunately, you can only read and write SFRs from code; see the suggestions in Section 3.2 for providing high-level access to SFRs of interest.

   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 0 overflow (TF0) interrupt to provide basic clock services. This provides a good example of a simple interrupt routine. The timer source may be found in **`Swiftx\Src\8051\Timer0.f`**, and the additional routines to maintain time-of-day and calendar support are in **`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. However, we have developed a simple scheme, described below, for keeping time in directly usable units without significant run-time overhead.

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).

We will provide two counters, a 32-bit millisecond counter called **`MSECS`** and a 48-bit time-of-day counter called **`SECS`**. With a millisecond tick rate, the millisecond counter will roll over every 49 days. This means it can be used to time intervals of up to 49 days. The time-of-day counter keeps 32-bit seconds and a 16-bit counter for fractions of a second. The two are separate, because the midnight time-of-day rollover needs to reset **`SECS`**, without disturbing any interval that is being timed by **`MSECS`**.

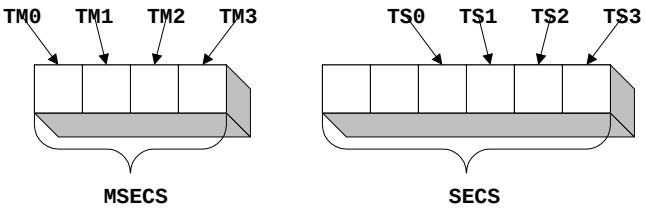

Figure 2. Timer management using incremental addends

At the beginning of the file **`\8051\Timer0.f`**, which contains this code, the actual clock rate is divided down to a succession of four one-byte fractional

increments for the millisecond timer **MSECS**, **TM0** through **TM3**; and a similar set for the low-order parts (fraction and low two bytes of the integer part), **TS0** through **TS1**. Each represents the correct fractional component to add to each byte. When an interrupt occurs, each component is added to its respective byte, and the carries are allowed to propagate. This scheme, shown in Figure 2, retains an accurate timer without costly scaling operations at interrupt time.

The interrupt routine looks like this:

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

LABEL <TF0>
   MSECS 3 + A MOV   TM3 # A ADD   A MSECS 3 + MOV \ Tally
   MSECS 2 + A MOV   TM2 # A ADDC   A MSECS 2 + MOV   \ ms
   MSECS 1 + A MOV   TM1 # A ADDC   A MSECS 1 + MOV   \ counter
   MSECS 0 + A MOV   TM0 # A ADDC   A MSECS 0 + MOV
   SECS 5 + A MOV   TS3 # A ADD   A SECS 5 + MOV   \ Tally
   SECS 4 + A MOV   TS2 # A ADDC   A SECS 4 + MOV     \ seconds
   SECS 3 + A MOV   TS1 # A ADDC   A SECS 3 + MOV     \ counter
   SECS 2 + A MOV   TS0 # A ADDC   A SECS 2 + MOV
   SECS 1+ A MOV   0 # A ADDC   A SECS 1+ MOV      \ Propagate
   SECS A MOV   0 # A ADDC   A SECS MOV              \ carry
   IRET   END-CODE

\ Attach <TF0> to the timer overflow interrupt:
   <TF0> TF0 INTERRUPT
```

Many systems handle the midnight rollover in clock interrupt code. However, it's inefficient to check so frequently for something that happens so infrequently! In this example, we check in the word **@NOW** (see Section 3.4), which is the command for fetching the system date and time. If your application code always uses **@NOW** (or the higher-level words that call it) to fetch the time, rather than reading **SECS** directly, you never need to worry about midnight rollover, and the overall cost to the system is minimized.

4.3 Example: Terminal I/O to an LCD Display

Serial 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**, respec-

tively), as well as for stream input and output (**ACCEPT** and **TYPE**, respectively). Basic principles of serial I/O in Forth are described in the *Forth Programmer's Handbook*, Section 3.8.

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

Our mission when 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 CME-562 board described in Appendix A.

There are two basic approaches to implementing serial drivers in Forth; they 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 next 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 this case (as with most local output), we will use a polled approach.

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**. Please refer to these files during the balance of this discussion.

- The layer generic to all Axiom boards contains no CPU-specific code or other features, so it is placed at the highest level of the **\SwiftX\Src** directory. (Because writing to an LCD display is not a time-critical activity, we designed these layers to keep as much of the logic in the high-level file as possible.)
- The device-specific layer, for the board itself, is in the lowest-level directory.

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**.

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.

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 ;
```

Because 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:

```
$FFF0 EQU LCD-CW        \ LCD command write
$FFF1 EQU LCD-DW        \ LCD data write
$FFF2 EQU LCD-CR        \ LCD command read
$FFF3 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@ ;
```

APPENDIX A: AXIOM CME-562 BOARD INSTRUCTIONS

This section provides information pertaining to the Axiom CME-562, which is supported by SwiftX for the 8051 family. 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 Axiom CME-562 is a fully configured development system for a compatible Philips variant of the popular Intel 8051 microcontroller. The system is supplied with the SwiftX kernel in the on-board EEPROM, 32K SRAM, a DB9 serial cable, 9v 200ma wall plug, printed hardware manual, and the UTL51 utility package of support software.

Features include:

- Philips S80C562-12 CPU
- Three configurable memory sockets, containing:
  - 32K EEPROM
  - 32K SRAM
- Two 16-bit timer/counters
- Additional 16-bit timer with four capture, three compare
- Two eight-bit PWM outputs
- Eight-channel eight-bit A/D
- SCI serial port
- RS232 serial port with DB9 connector
- Three I/0 ports:

  - Port 1, 3, and 4 (20 lines)
  - One input-only port
  - Port 5 (A/D) (eight lines)
- Keypad/serial accessory interface
- Keyboard/SPI interface
- LCD module interface
- Bus expansion port with seven chip selects
- 5.5" × 1.5" prototype area

Specifications:

- Board size 5.5" × 4.5"
- Power input: +7 to +12V standard
- Current consumption at 11 MHz: 100ma standard, 30ma optional

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 serial cable and power supply as directed in the "Getting Started" section of the *CME-562 Development Board User's Manual*. Do not attempt to perform the "Terminal" tests mentioned in that book, because the Axiom Monitor program has been replaced by the SwiftX kernel on the CME-562 board as supplied by FORTH, Inc.
2. Jumpers should be set as shown in Table 7.

3. The CME-562 has one serial communication port, COM1. SwiftX uses this port for the XTL, attached to the host on its selected serial port.

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

Table 7: Axiom CME-562 jumper settings

Jumper	State	Jumper	State
JP1	Open	JP4	2-3
JP2	Open	JP5	Open
JP3	Closed	JP6	2-3
JP7	Closed	JP8	2-3
JP9	Closed	JP10	1-2
JP11	Closed		
JP12	Closed		

A.3 Development Procedures

On the Axiom CME-562, 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 8051 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 kernel by selecting Project > Debug from the menu or its corresponding 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 respond to the prompt by pressing *y* to try again—in which case you will choose Project > Debug next—or *n* to abort.

When the connection is successfully established and the host version of the kernel matches the EEPROM's kernel, the target will display the system ID. 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 CME-562 board is shipped with a SwiftX kernel installed in its onboard 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.

To install a new kernel:


Build

1. Select the Project > Build menu item or its toolbar button—this generates a new **Target.hex** object file.
2. Turn off power to the board, and remove the EEPROM.
3. Use a PROM programmer utility of your choice to burn a new EEPROM for


Run

U5 using this file. You may use the Tools > Run menu item or toolbar button for this. Many people find it convenient to handle this with a batch file in their project directory.

4. Install the new EEPROM and apply power to the board.
5. Continue using Project > Debug as described in Section A.3.1.

A.3.3 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 a demo application described in the *SwiftX Reference Manual*.

The demo program must use the host keyboard and screen via the XTL (the default configuration).

To run the demo program, 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 Board-level Implementation Issues

This section describes features of the SwiftX implementation that are specific to the Axiom CME-562. The memory layout of the Axiom CME-562 board is shown in Figure 3.

The SwiftX system runs in the EEPROM code space at 0000_H. As shipped, it occupies about 5K of this space. There is ample room in the 16K RAM code space for application development. SwiftX uses only 13 bytes of internal RAM; the rest is available for application use.

The kernel configures a small amount of uData in internal RAM (in the **`..\Cme562\Config.f`** file). This internal RAM may only be accessed from within **`CODE`** words. The region from $C000_H$ to $DFFF_H$ is configured for iData, and a uData section is allocated between $E000_H$ and $FF7F_H$. You may adjust any of these allocations, if necessary, to suit your needs.

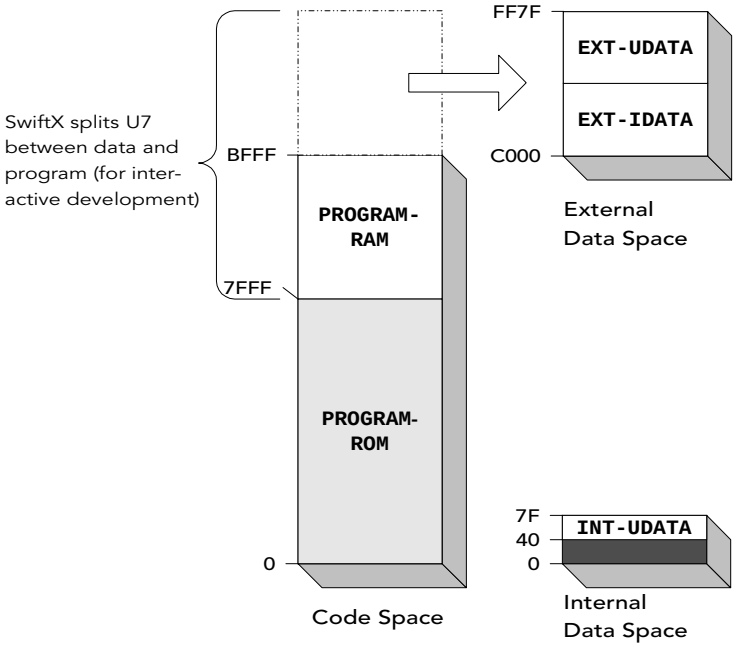

Figure 3. Memory organization in the Axiom CME-562

APPENDIX B: BCC52 BOARD INSTRUCTIONS

This section provides information pertaining to the BCC52, which is supported by SwiftX for the 8051 family. 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 BCC52 is a simple, low-cost, single-board controller development system. The board uses the Intel 8052AH-BASIC chip, although the on-board BASIC language support is not used by SwiftX.

Features include:

- Intel 80C52
- 256 bytes on-chip RAM
- Three 16-bit timer/counters
- Digital I/O: 24 bits PIA
- 11.0592 MHz system clock
- Six interrupts
- 8K bytes each data and code memory (expandable to 64K each)
- Keypad/serial accessory interface
- Console I/O RS232 serial port with auto baud rate to 19,200 baud
- On-board EPROM programmer

Specifications:

- Board size 6" × 4.5"
- Power (fully loaded): +5 V at 150 mA, ±12V at 25mA

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 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. The jumper settings on the BCC52 are all at factory settings except for:

   JP4 2–3 (2000_H base address)
   JP9 Installed (this disables the on-board BASIC)

   These changes are made by FORTH, Inc. prior to shipment. We have also moved the RAM chip from its factory installed position in U1 to U3, and we have installed the 8051 SwiftX PROM in U1.

2. Connect the UPS11 power supply to J6 on the BCC52. The blue wire (ground) is nearest the card edge connector. This power supply provides -5V instead of -12V for the negative RS232 line, but this is adequate according to Micromint for proper RS232 operation.
3. Connect a communications cable between the serial port on your PC and the serial port on the BCC52. The standard connection is your PC COM port to the BCC52 DB-25 connector. Only a three-wire connection is needed, in null-modem configuration.

   To connect to a standard nine-pin COM port, the cable should look like this:

PC DB9S	BCC52 DB25P
2	2
3	3
5	7

4. Apply power to the BCC52.

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

B.3 Development Procedures

On the BCC52, 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 those for other 8051 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 or its corresponding toolbar button. This completely compiles the kernel and compares it to the target's kernel 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 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 respond to the prompt by pressing *y* to try again—in which case you will choose Project > Debug next—or *n* to abort.

If the connection was successfully established and the host version of the kernel matches the PROM's kernel, the target will display the system ID. 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

The BCC52 is shipped with a SwiftX kernel installed in its on-board PROM. 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 PROM.

To install a new kernel:


Build


Run

1. Select the Project > Build menu item or its toolbar button—this generates a new **Target.hex** object file.
2. Turn off power to the board, and remove the PROM.
3. Use a PROM programmer utility of your choice to burn a new PROM for U1 using this file. You may use the Tools > Run menu item or toolbar button for this. Many people find it convenient to handle this with a batch file in their project directory.
4. Install the new PROM and apply power to the board.
5. Continue using Project > Debug as described in Section B.3.1.

B.3.3 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 must use the host keyboard and screen via the XTL (the default configuration).

To run the demo program, 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*.

B.4 Board-level Implementation Issues

This sections describes features of the SwiftX implementation that are specific to the BCC52. The memory layout of the BCC52 board is shown in Figure 4.

The SwiftX system runs in the PROM code space at 0000_H. As shipped, it occupies about 5K of this space. The 8K RAM in socket U3 is divided into code space (for interactive development) and iData/uData.

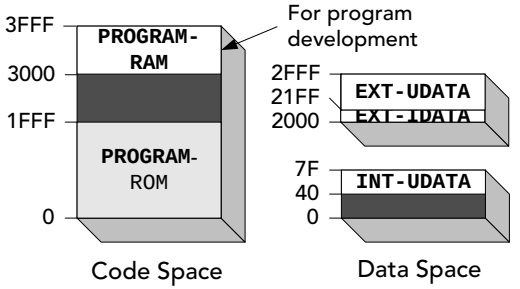

Figure 4. Memory organization in the BCC52

The kernel configures a small amount of uData in internal RAM (in the **`..\BCC52\Config.f`** file). Access to this internal RAM may only be from within **CODE** words. SwiftX uses only 13 bytes of internal RAM; the rest is available for application use.

GENERAL INDEX