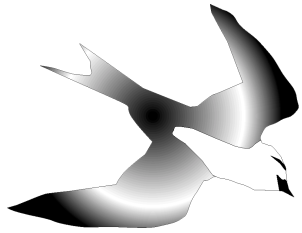

Getting Results Faster...

SwiftX AVR

Board-level Documentation for Atmel AVR-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

Second edition, July 1999
Printed 7/7/00

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 AVR 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 AVR; 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 AVR. Refer to Atmel's documentation for information about the AVR, 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
support@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.

1.1 INSTALLATION PROCEDURES

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

1. *Connect the target board* supported by this system to your PC, using the COM port you specified when you installed SwiftX. 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
Atmel STK200	Appendix A: ATMEL STK200 Board Instructions	37
Atmel MCU00100	Appendix B: ATMEL AVR Evaluation Board Instructions	43

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 pro-

gram 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 and the appropriate Appendix.

   General procedures for developing and testing software with SwiftX are provided in the *SwiftX Reference Manual*, Section 1.4.1. Further information regarding your board and special development procedures applicable to the AVR Development Board may be found in the appropriate Appendix.

2. THE AVR ASSEMBLER

The major uses for assembly routines are to implement the Forth Virtual Machine, to perform direct I/O operations when desired, and to optimize performance in time-critical functions. Virtually all Forth systems include an assembler; SwiftX cross-compilers provide an assembler for the target CPU, which in this case is a member of the AVR family of microcontrollers.

The AVR family has many variants designated by numeric identifiers such as the AT90S8515. 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 AVR only.

This section supplements, but does not replace, the CPU manufacturer's manuals. Departures from Atmel's usage are noted here; nonetheless, you should use the manufacturer's manuals for a detailed description of specific instructions. We use Atmel's mnemonics where possible, but postfix notation and Forth's data stack are used to simplify the assembler's operation. Therefore, words that specify registers, addressing modes, memory references, literal values, etc. *precede* the mnemonic.

Where **`boldface`** type is used here, it distinguishes Forth words (such as register names) from Atmel names. Usually these are the same; the name **`ADC`** can be used as a Forth word and as Atmel's name. Where boldface is *not* used, the name refers to Atmel'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 AVR processor. The mnemonics for the AVR opcodes have been defined as Forth words which, when executed, assemble the corresponding opcode at the next location in code space.

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

Some of the manufacturer's mnemonics have been replaced by different names, plus options to describe operations. The principal use of this strategy is in connection with conditional jumps. SwiftX constructs conditional jumps by using a condition code specifier followed by **IF**, **UNTIL**, or **WHILE**. Table 5 on page 11 summarizes the relationship between SwiftX condition codes used with one of these words and the corresponding Atmel 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 OVER ( x1 x2 -- x1 x2 x1)
   TPUSH                          \ Push x2 onto stack
   2 S TL LDD   3 S TH LDD        \ Fetch x1 to TOS
   RET   END-CODE
```

Register **usage** on the AVR is described in Section 2.3. In this example, the macro **TPUSH** pushes a copy of the top data stack item, which is kept in Register pair **XL:XH**, onto the stack, and the two **LDD** instructions put a copy of x_1 there.

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 *name* 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 Assembler macros, Section 2.5
Interrupt handling, Section 3.3
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 5

2.3 REGISTERS

The AVR's registers (shown in Figure 1) are defined as constants for use by the SwiftX assembler, using the published Atmel names. Certain registers are used in pairs, and those with special functions in the Forth virtual machine are given special Forth names. These are shown in Figure 1 and Table 2.

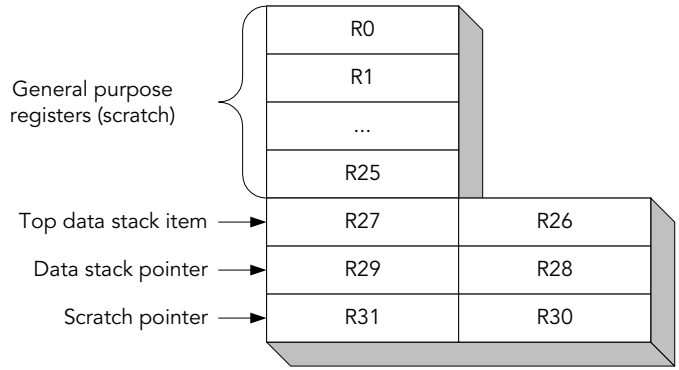

Figure 1. Register usage in AVR SwiftX

The names given to the registers designate either single registers (e.g., **TH**) or register pairs (e.g., **T** or **X**). Only a subset of AVR instructions can access register pairs. You may use either the Atmel names or the SwiftX names, although we recommend that you use the SwiftX names when the register is being used

in its SwiftX role (e.g., **TH** and **TL** when you're explicitly dealing with the top stack item).

Table 2: Special register assignments

Register(s)	Atmel name	SwiftX name	Description
R27:R28	X (XH:XL)	T (TH:TL)	Top data stack item
R29:R28	Y (YH:YL)	S (SH:SL)	Data stack pointer
R31:R30	Z (ZH:ZL)		Scratch pointer

Registers **TH** and **TL** contain the top data stack item. As an example of how these are used, consider the definition for **+**:

```
CODE + ( n1 n2 -- n3)
   S+ R0 LD   S+ R1 LD          \ Pop n1 into R0/R1
   R0 TL ADD   R1 TH ADC        \ Add n1 to n2
   RET   END-CODE
```

The AVR's processor stack pointer **SPH:SPL** is used for subroutine calls, functioning as Forth's return stack. As it is implemented as a pair of I/O registers, it is read by **IN** instructions, typically into register **Z**. For example:

```
CODE >R ( x -- ) ( R: -- x)
   TH PUSH   TL PUSH            \ Push TOS onto R
   TPOP   RET   END-CODE        \ Pop TOS
```

and:

```
CODE R@ ( -- x )   TPUSH        \ Save TOS on stack
   SPH ZH IN   SPL ZL IN        \ Read R-stack pointer to Z
   1 Z TL LDD  2 Z TH LDD       \ Get value
   RET   END-CODE
```

Only registers 16-31 can be used with literal data. Since the high-numbered registers have assigned functions, **R16** and up are often used as general-purpose registers when literal data may be involved. For example, here's the routine used to initialize the Timer0 interrupt:

```
CODE /TIMER ( -- )
   3 R16 LDI  R16 TCCR0 OUT     \ Prescale select = CK / 64
   TIMSK R16 IN   2 R16 ORI     \ Enable timer 0 overflow int
   R16 TIMSK OUT   RET   END-CODE
```

2.4 INSTRUCTION SYNTAX

This section describes instruction syntax in the SwiftX assembler, which differs from the manufacturer's primarily in the ordering of mnemonics and operands: operands *precede* mnemonics, in the order `<source> <destination>`.

2.4.1 Mnemonics

The mnemonics of the various AVR 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.

Most mnemonics require two addresses as operands: a source followed by a destination. These may be registers or other addressing modes. Thus:

```
TL R0 ADD
```

...adds the low order byte of the top stack item to the byte register **R1**. Similarly,

```
Z+ R0 LD
```

...moves the byte pointed to by the register pair **Z** to **R0**, incrementing **Z** after the move.

2.4.2 Operands

The notation for specifying addressing modes differs from Atmel's notation, in that the mode specifiers are operands that precede the mnemonics. Note that the syntax is consistently `<source> <destination> <opcode>`.

Table 3: Addressing modes

Mode	Example	Description
Direct	`R0 TL ADD`	Add the byte in **R0** to the byte in **TL**.
Immediate	`$98 R16 LDI`	Move 98_H to **R16**
Indirect	`S R0 LD`	Load the low-order byte of the second stack item into **R0**.
Indirect (post-increment)	`S+ R0 LD`	Pop the low-order byte of the second stack item into **R0**.
Indirect (pre-decrement)	`R0 -S ST`	Push **R0** as one byte of the second stack item.
Indexing	`1 S R1 LDD`	Load the middle byte of the second stack item into **R1**.
External	`R16 UDR OUT`	Output **R16** to UART data register **UDR**.

The pointer registers **X** (also known as **T**), **Y** (also known as **S**), and **Z** are referred to by their single-letter names when they are being used as register pairs for indirect or indexed addressing, but they must be addressed as their individual components (e.g., **XL** and **XH**) when loading or storing them.

2.4.3 Error Checking

The SwiftX assembler checks operands for the following error conditions:

1. A register is required but something other than a register is specified.
2. A bit number is not in the range 0 to 7.
3. The destination register of an immediate opcode is not within the range of registers allowed (generally, **R16**-**R31** for byte operations, and **R24**, **R26**, **R28**, **R30** for word operations).
4. An immediate value is out of range.
5. An absolute address is out of range.
6. An I/O register is out of range.
7. A condition code for a structured transfer is missing or not allowed.
8. A relative branch destination is out of range.

If an error is detected, SwiftX will abort with the message, `Illegal operand`.

If you type **L** following such a message, SwiftX will open your linked editor (if it is not already open) with the cursor positioned immediately after the opcode mnemonic that produced the error.

References Use of **L** following compile-time errors, *SwiftX Reference Manual* Section 2.4.1.

2.5 MACROS

The macros in Table 4 have been defined in order to simplify assembler coding for the 16-bit virtual machine.

Table 4: Simple macros

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

A 16-bit pointer register (**X**, **Y**, or **Z**) may be used as the source operand for **LDI** and **LDS** or as the destination for **STS**. The assembler splits the immediate or memory address operand and assembles two opcodes. For example:

$1234 Z LDI is equivalent to **$12 ZH LDI $34 ZL LDI**

2.6 RENAMED MNEMONICS

In most cases, SwiftX uses Atmel mnemonics and notation, though in postfix order. However, a few of Atmel'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 word **NOT** following a condition code inverts it.

Table 5: Conditional branch equivalencies

Atmel	SwiftX Assembler	Description
RJMP	**NEVER**	Unconditional branch.
BRCC	**CS**	Branch if carry clear.
BRNE	**0=**	Branch if non-zero.
BRPL	**0<**	Branch if not negative.
BRVC	**VS**	Branch if overflow clear.
BRGE	**S<**	Branch if greater than or equal to (signed compare).
BRHC	**HS**	Branch if half-carry clear.
BRTC	**TS**	Branch if T flag clear.
BRID	**IS**	Branch if interrupts disabled.

2.7 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 Table 5 and a glossary beginning on page 14. The combination of a condition code and a structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional branch instruction **BR**cc, where *cc* represents 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:

```
0 TL ADIW   0= IF
```

executes the true branch of the **IF** structure if register pair **TH:TL** contains 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 126 bytes in the object code. If it does, the assembler displays the `Range error` message and terminates. Control structures that span routines are not recommended—they make the source code harder to understand and harder to modify.

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

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

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 host's stack at compile time.

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 14.

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.

0< *(— cc)*

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

CS *(— cc)*

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

HS *(— cc)*

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

TS *(— cc)*

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

VS *(— cc)*

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

2.8 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**, **RCALL**, **JMP**, and **RJMP** are available in the generic AVR assembler, although the **CALL** and **JMP** opcodes arent implemented in all AVR microcontrollers. See your MCU manual for details.

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 126 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. 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 a character from R16
   BEGIN   UDRE USR SBIS  AGAIN \ Wait till port ready.
   R16 UDR OUT                   \ Output character
   RET   END-CODE
```

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

```
(OUT) RCALL
```

3. Implementation Issues

This section covers specific implementation issues involving this custom 24-bit Virtual Machine implementation on the AVR.

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. The AVR's subroutine stack is used by target primitives as Forth's return stack, to contain return addresses.

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

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

If you decompile it using the command **SEE DABS**, you get:

```
09B2   R27 -Y ST                    BA93
09B4   R26 -Y ST                    AA93
09B6   0< RCALL                     ECDC
09B8   R26 R27 OR                   BA2B
09BA   Y+ R26 LD                    A991
```

```
09BC   Y+ R27 LD                    B991
09BE   09C2 BRNE                    09F4
09C0   09C4 RJMP                    01C0
09C2   DNEGATE RCALL                96DF
09C4   RET                          0895 ok
```

The leftmost column shows the address, while the rightmost column shows the cells making up this definition. You can easily see the combination of direct code substitution for simple primitives, such as **DUP** and the **IF** code, combined with calls to **0<** and **DNEGATE**.

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 AVR is an 8-bit processor, but the Forth Virtual Machine is implemented with a 16-bit cell size, meaning that all addresses, stack items, and single-precision numbers are two bytes (16 bits) wide. The AVR is a "Harvard architecture" machine, with potentially 64K bytes each of directly addressable code and data space; however, the AT90S8515 allows only 8K bytes of code space in on-board flash. You will find a map showing memory usage in the file **..\Avr\Stk200\Config.f**.

Since all definitions must reside in on-chip flash memory (the only code space available for the 8515), all definitions must be in the kernel which is downloaded to flash as a single operation. It is not possible to add definitions interactively, as it is using SwiftX with targets providing development RAM.

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 internal SRAM. Stacks grow downward from high addresses. The return stack is implemented using 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 the following restrictions should always 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**;
- 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 AVR supports an adequate SwiftOS implementation. A task switch requires the following steps:

1. Save **T** on the data stack.
2. Save both stack pointers in the task's user variables **SSAVE** and **RSAVE**, respectively.

The three-byte **STATUS** area contains either a **WAKE** or **SLEEP** code 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 **STATUS** 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.

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

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

3.2 I/O REGISTERS

The AVR's I/O registers 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 Atmel designations, in a file for each supported MCU. (These files have names that take the form **Swiftx\Src\Avr\Reg_**<mcu>**.f**.) An example for the AVR90S8515 is shown in Table 6. Most are not used by the SwiftX kernel, and are available for program use.

The I/O registers 0-3F$_H$ are mapped to data space addresses 20-5F$_H$. The register names in Table 6 are defined in SwiftX such that when you assemble a reference to a register with an **IN** or **OUT** instruction, it will use the I/O address. All other references will use the data space address, so you may access these registers using **C@** and **C!**, even interactively for debugging (assuming you're connected to a running target).

Table 6: I/O registers in the AT90S8515

Addr.	Name	Description
3F	**SREG**	Status Register
3E	**SPH**	Stack Pointer High
3D	**SPL**	Stack Pointer Low
3B	**GIMSK**	General Interrupt MaSK register
3A	**GIFR**	General Interrupt Flag Register
39	**TIMSK**	Timer/Counter Interrupt MaSK register
38	**TIFR**	Timer/Counter Interrupt Flag register
35	**MCUCR**	MCU general Control Register
33	**TCCR0**	Timer/Counter0 Control Register
32	**TCNT0**	Timer/Counter0 (8-bit)
2F	**TCCR1A**	Timer/Counter1 Control Register A
2E	**TCCR1B**	Timer/Counter1 Control Register B
2D	**TCNT1H**	Timer/Counter1 High Byte

Table 6: I/O registers in the AT90S8515 *(continued)*

Addr.	Name	Description
2C	TCNT1L	Timer/Counter1 Low Byte
2B	OCR1AH	Timer/Counter1 Output Compare Register A High Byte
2A	OCR1AL	Timer/Counter1 Output Compare Register A Low Byte
29	OCR1BH	Timer/Counter1 Output Compare Register B High Byte
28	OCR1BL	Timer/Counter1 Output Compare Register B Low Byte
25	ICR1H	T/C 1 Input Capture Register High Byte
24	ICR1L	T/C 1 Input Capture Register Low Byte
21	WDTCR	Watchdog Timer Control Register
1F	EEARH	EEPROM Address Register High Byte
1E	EEARL	EEPROM Address Register Low Byte
1D	EEDR	EEPROM Data Register
1C	EECR	EEPROM Control Register
1B	PORTA	Data Register, Port A
1A	DDRA	Data Direction Register, Port A
19	PINA	Input Pins, Port A
18	PORTB	Data Register, Port B
17	DDRB	Data Direction Register, Port B
16	PINB	Input Pins, Port B
15	PORTC	Data Register, Port C
14	DDRC	Data Direction Register, Port C
13	PINC	Input Pins, Port C
12	PORTD	Data Register, Port D
11	DDRD	Data Direction Register, Port D
10	PIND	Input Pins, Port D
0F	SPDR	SPI I/O Data Register
0E	SPSR	SPI Status Register
0D	SPCR	SPI Control Register
0C	UDR	UART I/O Data Register
0B	USR	UART Status Register

Table 6: I/O registers in the AT90S8515 *(continued)*

Addr.	Name	Description
0A	UCR	UART Control Register
09	UBRR	UART Baud Rate Register
08	ACSR	Analog Comparator Control and Status Register

In addition to defining the registers, SwiftX has also defined individual bit masks for defined bits in many of these registers. The complete list of the defined bit masks is given in Table 7.

Table 7: Named bit numbers for AT90S8515 I/O registers

Bit	Name	Description
GENERAL INTERRUPT MASK REGISTER - GIMSK		
7	INT1	External Interrupt Request 1 Enable
6	INT0	External Interrupt Request 0 Enable
TIMER/COUNTER INTERRUPT MASK REGISTER - TIMSK		
7	TOIE1	Timer/Counter1 Overflow Interrupt Enable
6	OCIE1A	Timer/Counter1 Output CompareA Match Interrupt Enable
5	OCIE1B	Timer/Counter1 Output CompareA Match Interrupt Enable
3	TICIE1	Timer/Counter1 Input Capture Interrupt Enable
1	TOIE0	Timer/Counter0 Overflow Interrupt Enable
TIMER/COUNTER INTERRUPT FLAG REGISTER - TIFR		
7	TOV1	Timer/Counter1 Overflow Flag
6	OCF1A	Output Compare Flag 1A
5	OCF1B	Output Compare Flag 1B
3	ICF1	Input Capture Flag 1
1	TOV0	Timer/Counter0 Overflow Flag
MCU CONTROL REGISTER - MCUCR		
7	SRE	External SRAM Enable
6	SRW	External SRAM Wait State

Table 7: Named bit numbers for AT90S8515 I/O registers *(continued)*

Bit	Name	Description
5	SE	Sleep Enable
4	SM	Sleep Mode
3	ISC11	Interrupt Sense Control 1 bit 1
2	ISC10	Interrupt Sense Control 1 bit 0
1	ISC01	Interrupt Sense Control 0 bit 1
0	ISC00	Interrupt Sense Control 0 bit 0
TIMER/COUNTER1 CONTROL REGISTER A - TCCR1A		
7	COM1A1	Compare Output Mode1A, bit 1
6	COM1A0	Compare Output Mode1A, bit 0
5	COM1B1	Compare Output Mode1B, bit 1
4	COM1B0	Compare Output Mode1B, bit 0
1	PWM11	Pulse Width Modulator Select Bits
0	PWM10	Pulse Width Modulator Select Bits
TIMER/COUNTER1 CONTROL REGISTER B - TCCR1B		
7	ICNC1	Input Capture1 Noise Canceler (4 CKs)
6	ICES1	Input Capture1 Edge Select
3	CTC1	Clear Timer/Counter1 on Compare match
2	CS12	Clock Select1, bit 2
1	CS11	Clock Select1, bit 1
0	CS10	Clock Select1, bit 0
WATCHDOG TIMER CONTROL REGISTER - WDTCR		
4	WDTOE	Watch Dog Turn-Off Enable
3	WDE	Watch Dog Enable
2	WDP2	Watch Dog Timer Prescaler 2
1	WDP1	Watch Dog Timer Prescaler 1
0	WDP0	Watch Dog Timer Prescaler 0
EEPROM CONTROL REGISTER - EECR		
2	EEMWE	EEPROM Master Write Enable
1	EEWE	EEPROM Write Enable

Table 7: Named bit numbers for AT90S8515 I/O registers *(continued)*

Bit	Name	Description
0	**EERE**	EEPROM Read Enable
PORT A DATA REGISTER - PORTA		
7	**PA7**	
6	**PA6**	
5	**PA5**	
4	**PA4**	
3	**PA3**	
2	**PA2**	
1	**PA1**	
0	**PA0**	
PORT A DATA DIRECTION REGISTER - DDRA		
7	**DDA7**	
6	**DDA6**	
5	**DDA5**	
4	**DDA4**	
3	**DDA3**	
2	**DDA2**	
1	**DDA1**	
0	**DDA0**	
PORT A INPUT PINS ADDRESS - PINA		
7	**PINA7**	
6	**PINA6**	
5	**PINA5**	
4	**PINA4**	
3	**PINA3**	
2	**PINA2**	
1	**PINA1**	
0	**PINA0**	

Table 7: Named bit numbers for AT90S8515 I/O registers *(continued)*

Bit	Name	Description
PORT B DATA REGISTER - PORTB		
7	PB7	
6	PB6	
5	PB5	
4	PB4	
3	PB3	
2	PB2	
1	PB1	
0	PB0	
PORT B DATA DIRECTION REGISTER - DDRB		
7	DDB7	
6	DDB6	
5	DDB5	
4	DDB4	
3	DDB3	
2	DDB2	
1	DDB1	
0	DDB0	
PORT B INPUT PINS ADDRESS - PINB		
7	PINB7	
6	PINB6	
5	PINB5	
4	PINB4	
3	PINB3	
2	PINB2	
1	PINB1	
0	PINB0	
PORT C DATA REGISTER - PORTC		
7	PC7	

Table 7: Named bit numbers for AT90S8515 I/O registers *(continued)*

Bit	Name	Description
6	PC6	
5	PC5	
4	PC4	
3	PC3	
2	PC2	
1	PC1	
0	PC0	
PORT C DATA DIRECTION REGISTER - DDRC		
7	DDC7	
6	DDC6	
5	DDC5	
4	DDC4	
3	DDC3	
2	DDC2	
1	DDC1	
0	DDC0	
PORT C INPUT PINS ADDRESS - PINC		
7	PINC7	
6	PINC6	
5	PINC5	
4	PINC4	
3	PINC3	
2	PINC2	
1	PINC1	
0	PINC0	
PORT D DATA REGISTER - PORTD		
7	PD7	
6	PD6	
5	PD5	

Table 7: Named bit numbers for AT90S8515 I/O registers *(continued)*

Bit	Name	Description
4	PD4	
3	PD3	
2	PD2	
1	PD1	
0	PD0	
PORT D DATA DIRECTION REGISTER - DDRD		
7	DDD7	
6	DDD6	
5	DDD5	
4	DDD4	
3	DDD3	
2	DDD2	
1	DDD1	
0	DDD0	
PORT D INPUT PINS ADDRESS - PIND		
7	PIND7	
6	PIND6	
5	PIND5	
4	PIND4	
3	PIND3	
2	PIND2	
1	PIND1	
0	PIND0	
UART STATUS REGISTER - USR		
7	RXC	UART Receive Complete
6	TXC	UART Transmit Complete
5	UDRE	UART Data Register Empty
4	FE	Framing Error
3	OR	OverRun

Table 7: Named bit numbers for AT90S8515 I/O registers *(continued)*

Bit	Name	Description
SPI CONTROL REGISTER - SPCR		
7	**SPIE**	SPI Interrupt Enable
6	**SPE**	SPI Enable
5	**DORD**	Data ORDer
4	**MSTR**	Master/Slave Select
3	**CPOL**	Clock POLarity
2	**CPHA**	Clock PHAse
1	**SPR1**	SPI Clock Rate Select 1
0	**SPR0**	SPI Clock Rate Select 0
SPI STATUS REGISTER - SPSR		
7	**SPIF**	SPI Interrupt Flag
6	**WCOL**	Write COLlision flag
UART CONTROL REGISTER - UCR		
7	**RXCIE**	RX Complete Interrupt Enable
6	**TXCIE**	TX Complete Interrupt Enable
5	**UDRIE**	UART Data Register Empty Interrupt Enable
4	**RXEN**	Receiver Enable
3	**TXEN**	Transmitter Enable
2	**CHR9**	9 Bit Characters
1	**RXB8**	Receive Data Bit 8
0	**TXB8**	Transmit Data Bit 8
ANALOG COMPARATOR CONTROL AND STATUS REGISTER - ACSR		
7	**ACD**	Analog Comparator Disable
5	**ACO**	Analog Comparator Output
4	**ACI**	Analog Comparator Interrupt Flag
3	**ACIE**	Analog Comparator Interrupt Enable
2	**ACIC**	Analog Comparator Input Capture enable
1	**ACIS1**	Analog Comparator Interrupt Mode Select
0	**ACIS0**	Analog Comparator Interrupt Mode Select

For example:

```
LABEL (OUT)    BEGIN    UDRE USR SBIS    AGAIN
   R16 UDR OUT    RET    END-CODE
```

Note in this example (from **..\Avr\Serial.f**) the use of the bit number **UDRE** with register **USR**.

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 processor interrupt.

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

INTERRUPT takes an address for the handler and a vector address, and compiles an **RJMP** in the vector. When an interrupt occurs, control is passed to the handler without any further overhead.

The word **INTERRUPT** is only available at compile time, since it generates code in flash memory.

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

Table 8: Reset and interrupt vectors in the AT90S8515

Vector	Addr.	Source	SwX Name	Description
1	$000	RESET		Hardware Pin and Watchdog Reset
2	$001	INT0	**INT0addr**	External Interrupt Request 0
3	$002	INT1	**INT1addr**	External Interrupt Request 1

Table 8: Reset and interrupt vectors in the AT90S8515 *(continued)*

Vector	Addr.	Source	SwX Name	Description
4	$003	TIMER1 CAPT	**ICP1addr**	Timer/Counter1 Capture Event
5	$004	TIMER1 COMPA	**OC1Aaddr**	Timer/Counter1 Compare Match A
6	$005	TIMER1 COMPB	**OC1Baddr**	Timer/Counter1 Compare Match B
7	$006	TIMER1 OVF	**OVF1addr**	Timer/Counter1 Overflow
8	$007	TIMER0, OVF	**OVF0addr**	Timer/Counter0 Overflow
9	$008	SPI, STC	**SPIaddr**	Serial Transfer Complete
10	$009	UART, RX	**URXCaddr**	UART, Rx Complete
11	$00A	UART, UDRE	**UDREaddr**	UART Data Register Empty
12	$00B	UART, TX	**UTXCaddr**	UART, Tx Complete
13	$00C	ANA_ COMP	**ACIaddr**	Analog Comparator

Except for saving and restoring 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. Information on task control may be found in the *SwiftX Reference Manual*'s Section 5.

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\Avr\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 **\Avr**<platform>**\Start.f**. **POWER-UP** is assigned as the destination of the reset vector (by the phrase **POWER-UP 0 INTERRUPT** in **Start.f**).

Glossary

INTERRUPT *($addr_1$ $addr_2$ —)*

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

RETI *(—)*

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 is maintained using the Timer 0 overflow (**TF0**) interrupt. There is no date/time-of-day clock implemented at this time.

See Atmel's *Microcontroller Data Book* for details about Timer0. The number of milliseconds is accumulated by the **<TIMER0>** 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 6.1
Timer0 interrupt routine analyzed, Section 4.2

3.5 SERIAL CHANNEL

The AVR internal UART is used as the SwiftX Cross-Target Link (XTL), whose control is described in Section 4.9 of the *SwiftX Reference Manual*.

The driver for this port may be found in **Src\Avr\Serial.f**, and may be used as an example for 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 a specific device, efficiently reusing the common code.

   The on-board registers and bit numbers for your microcontroller are named in files whose names are **`\Swiftx\Src\Avr\Reg_`**`<mcu>`, where *mcu* is the particular variant of AVR controller (e.g., **`Reg_8515.f`**). These are described in Table 6 and Table 7. 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!) Unfortunately, you can only read and write on-board I/O registers from code previously assembled and downloaded in the kernel.

   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 INTERVAL TIMER

SwiftX provides basic timer services using both the Timer0 and Timer1 interrupts. The Timer1 version is simpler and more precise, but Timer1 is a more valuable application resource, so Timer0 may be a better choice. The Timer0 interrupt provides a good example of a simple interrupt routine. The source may be found in **`Swiftx\Src\Avr\Timer0.f`**.

The timer rate is set by a constant named **`CPUCLOCK`** in the file **`Swiftx\Src\Avr\Stk200\Config.f`** (along with other configuration information). This value is used in **`Timer0.f`** to calculate incremental values **`TC0`**, **`TC2`**, and **`TC3`** to add to components of the counter **`MSECS`** in interrupt code. **`MSECS`** itself is a double-cell integer whose first cell contains the free-running millisecond counter, and whose second cell is used to accumulate fractions of a millisecond. When an interrupt occurs, the fractional incremental values are added to the fractional component, and the carry (if any) is added to **`MSECS`**, along with its incremental component. When **`MSECS`** overflows, it will wrap. Its absolute value is meaningless; however, successive readings of it can be subtracted to time intervals of up to about 32 seconds.

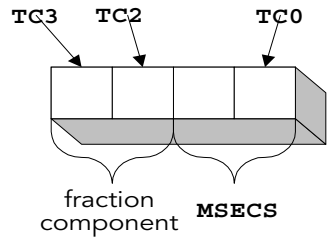

Figure 2. Timer management using incremental addends

The interrupt routine looks like this:

```
LABEL <TIMER0>   R16 PUSH
  SREG R16 IN    R16 PUSH   R17 PUSH \ Save registers
  TC2 R16 LDI    MSECS 2 + R17 LDS    \ Low byte of fractional part
  R16 R17 ADD    R17 MSECS 2 + STS    \ Add TC2
  TC3 R16 LDI    MSECS 3 + R17 LDS    \ High byte of fractional part
  R16 R17 ADC    R17 MSECS 3 + STS    \ Add TC3 + carry from TC2
  TC0 R16 LDI    MSECS R17 LDS        \ Low byte of integer ms count
  R16 R17 ADC    R17 MSECS STS        \ Add TC0 + carry from TC0
  CS IF   MSECS 1+ R17 LDS   R17 INC\ High byte of integer ms count
  R17 MSECS 1+ STS   THEN             \ Increment if carry from TC0
  R17 POP   R16 POP   R16 SREG OUT  \ Restore registers
```

```
    R16 POP    RETI    END-CODE
<TIMER0> OVF0addr INTERRUPT
```

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

4.3 Switches and LEDs

The Atmel Starter Kit Evaluation Board is equipped with eight LEDs and eight switches. The file **..\Stk200\Demo.f** contains a simple driver for these.

To run this demo, install jumpers as follows: Port B, 0-7; Port D, 2-7. Then look at **..\Stk200\App.f**. This file is intended to load your application. It should have **INCLUDE DEMO** following the initial **TARGET**. If it is there, this little application will be automatically loaded by the **BUILD** command (see Section B.3). When you have downloaded a kernel containing this code (following the instructions in Section B.3.1), the application will automatically start up the demo (done by the word **/SWITCHER** called in **GO**). Press various buttons, and observe the behavior of the LEDs. You can modify this behavior by typing **FAST**, **MEDIUM**, or **SLOW**.

This demo illustrates how easy it is to control hardware from SwiftX. You can read the switches by just typing

```
PIND C@
```

or write LEDs by typing

```
<n> PORTB C!
```

...where *n* is a value between 0 and 255. To improve readability, the demo has renamed these ports **SWITCHES** and **LEDS**, respectively. This will work fine for high-level Forth, but if you want to write assembler code you should use the port names, as the assembler handles these specially (see Section 3.2).

@SWITCHES returns the current switch value, and **!LEDS** writes a pattern. The word **PATTERN** generates an LED pattern based on the previous value written and the current switch reading. Finally, **/SWITCHER** starts a background task (*SwiftX Reference Manual*, Section 4) updating the LEDs in real time.

APPENDIX A: ATMEL STK200 BOARD INSTRUCTIONS

This section provides information pertaining to the ATMEL STK200 Starter Kit, which is supported by SwiftX for the AVR 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 AVR STK200 Starter Kit is designed to help new MCU users get acquainted quickly with the AT90S AVR microcontroller family. The STK200 can also be used to "breadboard" and try new designs before a final PCB.

Features include:

- AT90S8515 MCU
  - 8K bytes of In-System Reprogrammable Flash
  - SPI Serial Interface for Program Downloading
  - Flash Endurance: 1,000 Write/Erase Cycles
  - 512 bytes EEPROM
  - EEPROM Endurance: 100,000 Write/Erase Cycles
  - 512 bytes Internal SRAM
- Supports 8-pin, 20-pin, 28-pin & 40-pin AT90S AVR devices
- Includes In-System Programming cable for all classic AVR microcontrollers
- A/D Converter support with adjustable analog reference
- LCD interface with contrast adjustment
- RS232 serial port for PC interface

- External SRAM interface and latch socket
- 8 push-button switches and 8 LEDs
- All device ports easily accessible through 0.1" header connectors
- Selectable 3.3V or 5V operation
- Regulated power supply

A.2 Installation Instructions

If your board was delivered with SwiftX from FORTH, Inc., it will be properly configured to run SwiftX with no further changes.

Only the diskette labeled "AVR ISP Software" need be installed for the AVR ISP utility. The CD-ROM supplied by Atmel with the Starter Kit is not needed for SwiftX development.

Connect your board as described in the STK200 manual. Note there are two cables: a parallel cable connected to your printer port is used by the AVR ISP utility for downloads, and a serial cable is used for the SwiftX XTL.

A.3 Development Procedures

On the AT90S8515, the SwiftX kernel resides in flash memory. 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 SwiftX systems.

A.3.1 Installing a New Kernel in Flash Memory

Running an AVR system requires a SwiftX kernel installed in its on-chip flash memory. If you make any changes to your kernel, you must reprogram the flash memory. 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.

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 in the **`SwiftX\Src\Avr\Stk200`** directory.
2. Use the Atmel AVR ISP utility (which you may run using the Tools > Run menu item or Toolbar button) as described in Atmel documentation to download the object file into the flash memory on the board.


Run

3. Continue using Project > Debug as described in Section A.3.2.

The AVR ISP utility is somewhat complex to set up the first time, but since it provides for saving information in a "project" file, it's extremely easy to run thereafter. Specific procedural details vary somewhat with different releases, but in general the process is as follows:

1. Launch the AVR ISP. Start a session using the "New Project" button on the toolbar. Of the various MCU options presented, select AT90S8515 and enter a project title (e.g., `SwiftX`).
2. Bring the "Program Memory" window to the foreground by clicking on it. Using the File > Load menu option or Load File toolbar button, navigate to **`SwiftX\Src\Avr\Stk200`** and select the file **`Target.hex`**. This will load it into the Program Memory window.
3. Select Program > Auto-Program Options, which presents a dialog box. The default option is to select all; for SwiftX, *de*select the following: Program EEPROM, Verify EEPROM, Program Security Bits.
4. Save your project (using the Project > Save menu option or toolbar button), giving it a convenient name such as `SwiftX`.
5. Download the SwiftX kernel into the board using Auto-Program.

Once you have set up your project in this way, all you have to do is launch the ISP program and open your project (you may prefer to leave it open during your programming session), and just repeat the Auto-Program step for each download.

A.3.2 Starting a Debugging Session


Debug

Launch SwiftX as described in Section 1.3.1 of the *SwiftX Reference Manual* (if you haven't already done so). 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 flash memory.

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 flash memory as described in Section A.3.1.

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 press Y to try again, or press N and do a Project>Debug at a later time.

If the connection was successfully established and the host version of the kernel matches the kernel in flash memory, 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.3.3 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 that is loaded by the file **..\Stk200\App.f** when you build a kernel for downloading as described in Section A.3.1.

There are two possible demos: the "conical pile calculator" described in the *SwiftX Reference Manual* is in the file **\SwiftX\Src\Conical.f**. Another, which is specific to the Atmel evaluation board, may be found in **\SwiftX\Src\Avr\Stk200\Demo.f**; it is described in Section 4.3.

The "conical pile calculator" demo uses the host keyboard and screen via the XTL (the default configuration). To run this 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*.

APPENDIX B: ATMEL AVR EVALUATION BOARD INSTRUCTIONS

This section provides information pertaining to the ATMEL AVR Evaluation, which is supported by SwiftX for the AVR 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 AVR Development Board is designed to help new MCU users get acquainted quickly with the Atmel baseline microcontrollers The AVR Development Board can also be used to "breadboard" and try new designs before a final PCB.

Features include:

- AT90S8515 MCU
  - 8K bytes of In-System Reprogrammable Flash
  - SPI Serial Interface for Program Downloading
  - Flash Endurance: 1,000 Write/Erase Cycles
  - 512 bytes EEPROM
  - EEPROM Endurance: 100,000 Write/Erase Cycles
  - 512 bytes Internal SRAM
- Two serial ports
- Software utility to download into the onboard flash memory
- Regulated power supply for both DC and AC voltage sources
- Eight push-buttons for general use

- Eight LEDs for general use
- All AVR ports are easily accessible through header connectors

B.2 Installation Instructions

If your board was delivered with SwiftX from FORTH, Inc., it will be properly configured to run SwiftX with no further changes. If you received your board separately, you must connect a two-wire jumper (supplied with the Starter Kit) from Port D pins 1 and 2 to J132 pins 1 and 2 to connect the internal UART to J131.

Only the diskette labeled "AVR Development Board Software" need be installed for the AvrProg utility. The other diskettes and CD-ROM supplied by Atmel with the Starter Kit are not needed for SwiftX development.

B.3 Development Procedures

On the AT90S8515, the SwiftX kernel resides in flash memory. 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 SwiftX systems.

Note that since both the SwiftX XTL and Atmel's flash memory download utility use serial ports, it's convenient to have two serial ports on your PC, using one for each purpose. Otherwise, you will have to change serial cables or use an A/B switch to alternate between flash memory downloads and interactive debugging sessions.

B.3.1 Installing a New Kernel in Flash Memory

Running an AVR system requires a SwiftX kernel installed in its on-board flash memory. If you make any changes to your kernel, you must replace this kernel. 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.

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. Use the Atmel AvrProg utility (which you may run using the Tools > Run menu item or Toolbar button) as described in Atmel documentation to download the object file into the flash memory on the board.
3. Continue using Project > Debug as described in Section B.3.2.

B.3.2 Starting a Debugging Session


Debug

Launch SwiftX as described in Section 1.3.1 of the *SwiftX Reference Manual* (if you haven't already done so). 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 flash memory.

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 flash memory as described in Section B.3.1.

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 press Y to try again, or press N and do a Project>Debug at a later time.

If the connection was successfully established and the host version of the kernel matches the kernel in flash memory, 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.3.3 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 that is loaded by the file **`..\Evb\App.f`** when you build a kernel for downloading as described in Section B.3.1.

There are two possible demos: the "conical pile calculator" described in the *SwiftX Reference Manual* is in the file **`\SwiftX\Src\Conical.f`**. Another, which is specific to the Atmel evaluation board, may be found in **`\SwiftX\Src\Avr\Evb\Demo.f`**; it is described in Section 4.3.

The "conical pile calculator" demo uses the host keyboard and screen via the XTL (the default configuration). To run this 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*.

GENERAL INDEX