Getting Results Faster...

SwiftX MSP430

Board-level Documentation for TI MSP430 Targets

FORTH, Inc.
5155 W. Rosecrans Ave., Suite 1018
Hawthorne, California USA 90250
310.491.3356 800.55.FORTH FAX: 310.978.9454
sales@forth.com www.forth.com

FORTH, Inc.
5155 W. Rosecrans Ave., Suite 1018
Hawthorne, California USA 90250
310.491.3356 800.55.FORTH FAX: 310.978.9454
sales@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 MSP430 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 MSP430; 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 MSP430. Refer to Texas Instruments' documentation for information about the MSP430, 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.
5155 W. Rosecrans Ave, #1018
Hawthorne, California USA 90250-6694
(+1) 310.491.3356 or 800.55.FORTH (U.S. and Canada)
support@forth.com

In addition, FORTH, Inc. maintains an email group that enables SwiftX users to share experiences and code and solve each others' application-related problems. Archives are located at **`http://www.forth.com/swiftx`**. You can also search the archives. Visit **`http://www.forth.com/search`** for details. To subscribe, send a blank email to **`listar@forth.com`** with the phrase **`subscribe swiftx`** as the subject. You may cancel your subscription at any time by sending email to **`listar@forth.com`** with the subject **`unsubscribe swiftx`**.

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
TI FET430P140	Appendix A: TI FET430P140 Board Instructions	41

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.
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 MSP430 Assembler

The MSP430 devices constitute a family of exceptionally low-power, 16-bit RISC microcontrollers with an advanced architecture and rich peripheral set.

The MSP430 consumes less than 400 µA in active mode operating at 1 MHz in a typical 3-V system and can wake up from a <2-µA standby mode to fully-synchronized operation in less than 6 µs. These exceptionally low current requirements, combined with the fast wake-up time, enable a user to build a system with minimum current consumption and maximum battery life.

Additionally, the MSP430x1xx family has an abundant mix of peripherals and memory sizes enabling true system-on-a-chip designs. The peripherals include a 12-bit A/D, slope A/D, multiple timers (some with capture/compare registers and PWM output capability), on-chip clock generation, H/W multiplier, USART(s), Watchdog Timer, GPIO, and others.

Throughout this book, we assume you understand the hardware and functional characteristics of the MSP430 processor as described in the *Texas Instruments MSP430x1xx Family User's Guide*. We also assume you are familiar with the basic principles of Forth.

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

Where **`boldface`** type is used here, it distinguishes Forth words (such as register names) from Texas Instruments' names. Usually these are the same; for example, the name **`MOV`** can be used as a Forth word and as a Texas Instruments mnemonic. Where boldface is *not* used, the name refers to the manufacturer's usage or to hardware issues that are not particular to SwiftX or Forth.

2.1 SwiftX Assembler Principles

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

The SwiftXMSP430 cross-compiler provides an assembler for the Texas Instruments MSP430. The mnemonics for the MSP430 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 4 on page 12 summarizes the relationship between SwiftX condition codes used with one of these words and the corresponding Texas Instruments 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 SWAP ( x1 x2 -- x2 x1 ) \ Swap top two stack items
   T R8 MOV              \ Top stack item to R8
   @S T MOV              \ 2nd stack item to top
   R8 0 (S) MOV          \ R8 to 2nd stack position
   RET
END-CODE
```

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

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

```
    WAIT 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 jump or call, for example. The code fragments used as interrupt handlers are constructed in this way, and the named locations are then passed to **`INTERRUPT`**, which connects the code address to a specified exception vector.

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

- If you reference a **`CODE`** definition inside a colon definition, the cross-compiler will assemble a call to it; if you invoke it interpretively while connected to a target, it will be executed.
- Reference to a **`LABEL`** under any circumstance will return the *address* of the labelled code.

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

Glossary

`CODE` `<name>` *(—)*

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

`LABEL` `<name>` *(—)*

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

WAIT *(— addr)*

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

END-CODE *(—)*

Terminate an assembler sequence started by **CODE** or **LABEL**.

References Interrupt handling, Section 2.8
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 5

2.3 Registers

The MSP430 incorporates a reduced and highly transparent instruction set and a highly orthogonal design. It consists of a 16-bit arithmetic logic unit (ALU), 16 registers, and instruction control logic.

All registers except the constant-generator registers R3/CG2 and part of R2/CG1 can be accessed using the complete instruction set. The constant generator supplies instruction constants, and is not used for data storage. The addressing mode used on CG1 separates the data from the constants. The SwiftX assembler will automatically generate the addressing modes that use the constant generator when presented with literal values -1, 0, 1, 2, 4, and 8.

In SwiftX, registers **R0**–**R7** are reserved for functions dedicated to the CPU and for the Forth Virtual Machine. Registers **R8**–**R15** are available for use by primitives, interrupts, and **CODE** definitions.

Table 2: Special register assignments

TI name	Forth alias	Usage
PC		Program counter
SP	**R**	Return stack pointer

Table 2: Special register assignments

TI name	Forth alias	Usage
SR		Status register
R3		Constant generator (not directly accessible)
R4	**S**	Data stack pointer
R5	**T**	Top data stack item
R6	**U**	User (task) pointer
R7		Reserved for future use

- **SP** (the processor stack pointer) is used as Forth's Return Stack pointer **R**.
- **R4** is used as Forth's Data Stack pointer **S**. It actually points to the *second* stack item.
- **R5** contains the actual top stack item **T**. Having it in a register speeds up many operations.
- **R6** or **U** contains the address of the user area for the task currently running. This provides easy access to its user variables.

It's conventional when you're using the Forth registers for their Forth purposes to use their aliases, but when they're used for other purposes you may use their TI names (except for R3, which isn't directly accessible).

You may use the registers dedicated to Forth functions for other purposes if you wish, providing you save and restore their contents. Unless you have dedicated **R8–R15** to application functions, you may use them freely without saving and restoring.

Additional I/O registers are defined as constants for use by the SwiftX assembler, using the published manufacturer names. These registers (such as **P1IN**) can be referred to by name inside both code and high-level Forth definitions. They are also available to the host command-line interpreter during interactive debugging. The actual register list varies somewhat with the various members of the MSP430 family (see Section 3.2).

2.4 Addressing Modes

The notation for specifying addressing modes in SwiftX differs from common assembler notation, in that the mode specifiers are operands that *precede* the mnemonics. In this MSP430 assembler, instruction mnemonics are words that actually assemble opcodes using parameters left on the stack by the mode operands. Note that the syntax is `<operand(s)> <opcode>`.

Table 3 lists the addressing modes of the MSP430 and offers examples that show the difference between Texas Instruments and SwiftX assembler notation. Note that SwiftX assembler modes and operands are separated by spaces.

Table 3: Addressing modes

Mode	TI example	SwiftX	Description
Register	R8	**`R8`**	Use **`R8`** contents
Indexed	5(R8)	**`5 (R8)`**	Use **`R8`** plus 5
Symbolic	TICKS1	**`TICKS1`**	Use the address returned by **`TICKS1`** (assembling a relative offset)
Absolute	&TACTL	**`TACTL &`**	Use the I/O register **`TACTL`**
Indirect register	@R8	**`@R8`**	Use the contents of the location pointed to by **`R8`**.
Indirect autoincrement	@R8+	**`@R8+`**	Use the contents of the location pointed to by **`R8`**, then increment **`R8`**.
Immediate	#500	**`500 #`**	Use the value 500.

Some instructions (e.g., **`MOV`**) can take both source and destination addresses. All seven of these addressing modes can be used for the source, but only the first four are valid as destination addresses.

2.5 Instruction Set

The MSP430 has a limited basic instruction set. Certain frequently-used combinations of basic instructions with register designations and other options are used to generate an extended set of "emulated instructions," described in TI documentation. SwiftX supports both the basic and emulated instruction set.

In addition, SwiftX provides two macros for combinations that occur frequently in Forth code:

- **`TPUSH`** pushes the top stack item in register **`T`** onto the data stack.
- **`TPOP`** pops the second stack item (pointed to by **`S`**) into register **`T`**.

A software multiply routine is provided for those parts that do not have a hardware multiplier. For these, the word **`UM*`** is the primitive multiply routine that is used as the basis for all multiply operations.

2.6 Direct Transfers

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

However, direct transfers are useful at times, particularly when compactness of the compiled code overrides all other criteria. **`JMP`**, **`CALL`**, and the emulated instruction **`BR`** are defined as described in Texas Instruments documentation.

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`**, **`CALL`**, **`BR`**, or an interrupt vector.

For example, in the multitasker we find:

```
LABEL GIVE      \ Give CPU to next task, addr on stack
   U POP   U DECD        \ Calculate task address in U
   SLEEP # STATUS (U) MOV\ Set task's STATUS to SLEEP
   SSAVE (U) S MOV   TPOP\ Restore data stack
   RSAVE (U) R MOV       \ Restore return stack,
   RET   END-CODE        \ ...return to task code

CODE STOP ( -- )         \ Suspend task execution
   TPUSH   GIVE # T MOV  \ Save TOS, set T to addr of GIVE
                         \ Save stack pointers:
   S SSAVE (U) MOV   R RSAVE (U) MOV
   FOLLOWER (U) BR       \ Begin executing next task.
   END-CODE
```

The code at **`GIVE`** starts up a new task. The address of **`GIVE`** is placed in **`T`**, and when **`STOP`** branches to a task with the **`WAKE`** instruction in its **`STATUS`**, it will do a **`T CALL`**.

References For a further discussion of the multitasker code, see Section 3.1.4.

2.7 Assembler Structures

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

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

The program structures supported in this assembler are:

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

In the sequences above, *cc* represents condition codes, which are listed in a glossary beginning on page 13. The combination of a condition code and a structure word (**UNTIL**, **WHILE**, **IF**) assembles a conditional jump instruction *Bcc*, where *cc* is the condition code. The other components of the structures—**BEGIN**, **REPEAT**, **ELSE**, and **THEN**—enable the assembler to provide an appropriate destination address for the jump.

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

```
R8 T CMP   S< IF
```

executes the true jump of the **IF** structure if the top stack item (in **T**) is less than the contents of **R8**.

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 jump forward more than 1024 bytes or back more than 1022 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 4 shows the instructions generated by a SwiftX conditional phrase. These examples use **IF**. **WHILE** and **UNTIL** generate the same instructions, but satisfy the jump 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 jump in the true case*. Therefore, the combination of the condition code and jump instruction assembled by **IF**, etc., jump on the *opposite* condition (i.e., ≥ 0 in this case).

Table 4: Instructions generated by SwiftX conditional structure words

Phrase	Instruction assembled	Description
`0= IF`	**`JNZ`**	Jump if non-zero.
`0= NOT IF`	**`JZ`**	Jump if zero.
`0< IF`	**`JN JMP`**	Jump if non-negative
`0< NOT IF`	**`JN`**	Jump if negative
`S< IF`	**`JGE`**	Signed less-than; jump if the N and V bits are both set or both clear.
`S< NOT IF`	**`JL`**	Signed greater-or-equal; jump if the Z bit is set, or if the N and V bits differ from each other.
`CS IF`	**`JNC`**	Jump if the carry bit is not set.
`CS NOT IF`	**`JC`**	Jump if the carry bit is set.
`NEVER IF`	**`JMP`**	Unconditional jump

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

In the glossary entries below, the stack notation *cc* refers to a condition code. Available condition codes are listed beginning on page 13.

Glossary

Jump Macros

`BEGIN` *(— addr)*

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

`AGAIN` *(addr —)*

Assemble an unconditional jump to *addr*.

`UNTIL` *(addr cc —)*

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

REPEAT *($addr_2$ $addr_1$ —)*

Set the destination address of the jump 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 jump to the location *$addr_2$* (presumably left by a preceding **BEGIN**).

IF *(cc — addr)*

Assemble a conditional jump whose destination address is not given, and leave the address of the jump 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 jump (with unspecified destination) whose address *$addr_2$* is left on the stack.

THEN *(addr —)*

Set the destination address of a jump 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 jump on positive (N bit not set).

0= *(— cc)*

Return the condition code that—used with **IF**, **WHILE**, or **UNTIL**—will generate a jump on non-zero (Z bit not set).

S< *(— cc)*

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

CS *(— cc)*

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

NEVER *(— cc)*

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

NOT *(cc_1 — cc_2)*

Invert the condition code cc_1 to give cc_2.

2.8 Interrupt Handling

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

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

To attach the code to the handler, use the word **INTERRUPT**, which takes an address for the trap handler and an interrupt number, and links them such that when the interrupt occurs, it will be vectored directly to the code. No overhead is imposed by SwiftX, and no task needs to be directly involved in interrupt handling. If a task is performing additional high-level processing (for example, calibrating data acquired by interrupt code), the convention in SwiftX is that the handler code performs only the most time-critical processing, and notifies a task of the event by modifying a variable or by setting the task to wake up. Further information on task control may be found in Section 5 of the *SwiftX Reference Manual*.

One example of the use of interrupts is given in Section 4.2. Another example of a simple interrupt handler is the one provided for the real-time interrupt in **\Swiftx\Src\Msp430\TimerA.f**. It looks like this:

```
LABEL <MSEC>           \ TimerA 0 Interrupt response
   T(MS) # CCR0 & ADD  \ Advance CCR0 for next interrupt
    K1(MS) # MSECS CELL+ & ADD \ Add fractional part
```

```
      K0(MS) # MSECS & ADDC     \ Accumulate integer part
      CCIFG # CCTL0 & BIC       \ Clear interrupt
      RETI   END-CODE

   <MSEC> TIMERA0_VECTOR INTERRUPT
```

`<MSEC>` is the real-time interrupt handler, and **`TIMERA0_VECTOR`** is the interrupt vector. This increments the fractional and (when appropriate) integer components of the **`2VARIABLE MSECS`**, and returns.

Table 5 lists the interrupt vectors that are named in SwiftX for the MSP430F149 (see **`Swiftx\Src\Msp430\Reg_14x.f`**). Similar files are provided for other members of this family.

Table 5: Named Interrupt Vectors

Address	Name	Device
FFE2	**`PORT2_VECTOR`**	I/O port P2
FFE4	**`UART1TX_VECTOR`**	USART 1 transmit
FFE6	**`UART1RX_VECTOR`**	USART 1 receive
FFE8	**`PORT1_VECTOR`**	I/O port P1
FFEA	**`TIMERA1_VECTOR`**	Timer A
FFEC	**`TIMERA0_VECTOR`**	TimerA
FFEE	**`ADC_VECTOR`**	A/D converter
FFF0	**`UART0TX_VECTOR`**	USART 0 transmit
FFF2	**`UART0RX_VECTOR`**	USART 0 receive
FFF4	**`WDT_VECTOR`**	Watchdog timer
FFF6	**`COMPARATORA_VECTOR`**	Comparator A
FFF8	**`TIMERB1_VECTOR`**	Timer B
FFFA	**`TIMERB0_VECTOR`**	Timer B
FFFC	**`NMI_VECTOR`**	Non-maskable interrupt
FFFE	**`RESET_VECTOR`**	System reset

Power-up initialization for any of these vectors that are to be used in the target should be done by the word **`START`**, which can be found in **`\Swiftx\Src\Msp430\`**`<platform>`**`\Start.f`**.

For maximum flexibility, the actual interrupt vector table goes through a jump table in RAM on the MSP430F135 and larger parts. The vector table in flash is initialized with branches to the corresponding entries in the RAM table. This layer of indirection enables you to add new interrupt handlers while testing and modify interrupt vectors at run time. The smaller parts do not have enough RAM for either program RAM (useful for interactive testing) or this "shadow" interrupt vector table, so their vectors reside in flash only.

Glossary

INTERRUPT *(addr n —)*

Store address *addr* into interrupt vector *n*. Potentially three versions are supplied: the **INTERPRETER** version is used to set the code image vector on all targets, while the **TARGET** version sets the RAM vector at run time in the target (for those parts that support it). An **ASSEMBLER** macro is also provided, which is equivalent to the **TARGET** version, for use in code.

References See the System Clock example, Section 4.2.

3. IMPLEMENTATION ISSUES

This section covers specific implementation issues involving various versions of the MSP430 processor. For board-specific details, see the relevant appendix.

3.1 IMPLEMENTATION STRATEGY

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

3.1.1 Execution Model

The execution model is a subroutine-threaded scheme, with in-line code substitution for simple primitives. The MSP430's subroutine stack is used by target primitives as Forth's return stack, and may contain return addresses.

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

You may see examples of SwiftX MSP430 optimization strategies by decompiling some simple definitions. For example, the source definition for **/STRING** is:

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

but if you decompile it, you get:

```
SEE /STRING
1570   R5 PUSH                            0512
1572   @R4+ R5 MOV                        3544
1574   SWAP # CALL                        B012AC11
1578   2 # R4 SUB                         2483
157A   R5 0 (R4) MOV                      84450000
157E   @SP R5 MOV                         2541
1580   @R4+ R5 ADD                        3554
1582   SWAP # CALL                        B012AC11
1586   2 # R4 SUB                         2483
1588   R5 0 (R4) MOV                      84450000
158C   @SP+ R5 MOV                        3541
158E   - JMP                              9E3E ok
```

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

When the last thing in a definition is a subroutine reference, the compiler automatically substitutes a **JMP** for the subroutine return (**@SP+ PC MOV**), as in the example above.

More extensive optimization is provided by a powerful rule-based optimizer provided with SwiftX Pro, which can optimize a number of common high-level phrases. This optimizer is normally running, but can be turned off for debugging or comparison purposes. For example, consider the definition of **DIGIT**, which converts a small binary number to a digit:

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

With the optimizer turned off, you would get:

```
SEE DIGIT
262C   DUP # CALL                         B0128C11
2630   (LITERAL) # CALL                   B0120011
2634      LITERAL 9
2636   > # CALL                           B0122413
263A   0 # R5 CMP                         0593
263C   @R4+ R5 MOV                        3544
263E   2648 JZ                            0424
2640   (LITERAL) # CALL                   B0120011
```

```
2644       LITERAL 7
2646   @R4+ R5 ADD                      3554
2648   (LITERAL) # CALL                 B0120011
264C       LITERAL 30
264E   @R4+ R5 ADD                      3554
2650   @SP+ PC MOV                      3041 ok
```

But with it turned on, you would get:

```
SEE DIGIT
2652   9 # R8 MOV                       38400900
2656   R5 R8 CMP                        0895
2658   265E JGE                         0234
265A   7 # R5 ADD                       35500700
265E   30 # R5 ADD                      35503000
2662   @SP+ PC MOV                      3041 ok
```

Another example shows the compiler's ability to "fold" literals and operations on literals into shorter sequences. The definition…

```
 : TEST   $FFFF AND DUP 6 CELLS + CELL+ @ ;
```

…generates the following code un-optimized:

```
SEE TEST
23DC   (LITERAL) # CALL                 B0120011
23E0      LITERAL FFFF
23E2   @R4+ R5 AND                      35F4
23E4   DUP # CALL                       B0128C11
23E8   (LITERAL) # CALL                 B0120011
23EC      LITERAL 6
23EE   R5 R5 ADD                        0555
23F0   @R4+ R5 ADD                      3554
23F2   2 # R5 ADD                       2553
23F4   @R5 R5 MOV                       2545
23F6   @SP+ PC MOV                      3041 ok
```

…and the following code with the optimizer running:

```
SEE TEST
23D2   DUP # CALL                       B0128C11
```

```
23D6   E (R5) R5 MOV                          15450E00
23DA   @SP+ PC MOV                            3041 ok
```

The optimized version is significantly shorter (10 bytes vs. 28 for the un-optimized version, not counting the code at **(LITERAL)** which is called twice in the unoptimized version, and is much faster.

To experiment with this further, follow this procedure:

1. Turn off the optimizer, by typing:

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

   ```
   +OPTIMIZER
   ```
5. Re-enter your definition.

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

6. Decompile it and compare.

3.1.2 Data Format and Memory Access

Because the MSP430 is a 16-bit processor, its directly addressable memory space is limited to 64K. The appendix concerning the board that accompanies your system includes a memory map showing the available memory.

The low byte of a 16-bit cell on the MSP430 occupies the lower address in memory—i.e., this is a *little-endian* machine.

This processor requires instructions and the objects of 16-bit fetches and stores to be on even addresses. SwiftX ensures that the beginnings of definitions, all assembled instructions, variables, and uses of **BUFFER:** with an even length to be aligned on cell boundaries. In general, addresses will be even and you don't need to worry about alignment unless you add an odd increment to an address. Also, you should avoid mixing the use of **C,** and **,** when compiling values in the data space following a **CREATE**.

If you feel you need to force alignment (e.g., you're incrementing an address by a value that may be odd, or to force alignment after an odd number of **C,**'s before a **,**) use **ALIGN** or **ALIGNED** (see *Forth Programmer's Handbook*, Section 4.3.2).

3.1.3 Stack Implementation and Rules of Use

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

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

3.1.4 SwiftOS Multitasker Implementation

The MSP430 supports a very efficient SwiftOS implementation, with six instructions required to deactivate a task and eight to activate one. The subroutine-threaded implementation means there is no **I** register (see address interpreter, Section 5.3 of the *SwiftX Reference Manual*) to be saved and restored, only the return and data stack pointers.

The four-byte **STATUS** contains a **BR** to the next task in the round robin.

Because the **BR** in the active task's **STATUS** is always followed by the address of the next task, **WAKE** decodes to **T CALL**. The appropriate destination address for the task wake-up code is placed in **T** when any task gives up control of the CPU.

Table 6: SwiftOS user status instructions

Name	Value (hex)	Instruction	Description
WAKE	1285	**T CALL**	Call to wake-up code pointed to by **T**.
SLEEP	4030	**a # BR**	Jump to next task at address **a** (in next cell).

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

A task can block itself from high-level Forth by using **PAUSE** (which relinquishes the CPU for exactly one circuit through the multitasker round-robin) or **STOP** (which suspends it indefinitely, until it is awakened by an interrupt routine or another task). From assembler code, the appropriate instruction is:

```
SLEEP # STATUS (U) MOV
```

…since Register **U** (**R6**) contains a pointer to the current running task. A code definition containing this instruction should end with **WAIT JMP** instead of the usual **RET**.

If a task is asleep pending completion of an I/O operation, it will normally be awakened by the interrupt code that detects the completion event. That code sets the task to awaken by storing **WAKE** in its status. However, when an interrupt occurs, **U** will *not* point to the task you wish to awaken, because it is asleep. You must either build into your code the address of the task's status (obtained by the phrase `<taskname>` **STATUS HIS**) or "remember" which task is using the resource in a variable. For an example of the latter strategy, see the sample program **..\FET430P140\Distress.f**, described in Section 4.3.

If you wish to review the simple code for this SwiftOS multitasker, you will find it in the file **Swiftx\Src\Msp430\Tasker.f**.

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

3.2 I/O Registers

The MSP430's I/O registers vary somewhat with each microcontroller version. Refer to TI documentation for your particular MCU for details regarding the use of these registers.

SwiftX defines names for the registers, corresponding to their TI designations, in a file for each supported MCU (these files have names that take the form **`Swiftx\Src\Msp430\Reg_`**`<mcu>`**`.f`**). An example for the MSP430x14x is shown in **`Reg_14x.f`**. These registers may be referenced by these names in code or in high-level definitions; they may also be interrogated interactively if your target is connected. Most are not used by the SwiftX kernel, and are available for program use. SwiftX as shipped provides register files for the MSP430x12x, MSP430x13x, and MSP430x14x families. These are automatically included in the kernels for each target supported.

3.3 Timers

The system millisecond timer is implemented using the MSP430 TimerA Interrupt which is programmed here to use a 32.768 kHz crystal.

See Section 11.3 of TI's *MSP430x1xx Family User's Guide* SLAU049.pdf, Chapter 10, for details about TimerA. The number of milliseconds is accumulated by the **`<SEC>`** interrupt handler in the variable **`MSECS`**.

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

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

The interrupts associated with the capture/compare registers on TimerA are used to accumulate the current date and time.

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

3.4 JTAG Interface Support

The MSP430 family includes a JTAG (IEEE 1149.1, *Test Access Port and Boundary Scan Architecture*[*]) interface, which is used by SwiftX for writing code into flash memory and also for XTL communication. The host is connected to the JTAG port on the target using TI's "MSP430 Flash Emulator Tool" connected to the host's parallel (printer) port.

Use of the JTAG interface significantly changes the Cross-Target Link (XTL) protocol from the serial protocol described in the *SwiftX Reference Manual*. Whereas a serial protocol implies a table of commands encoded as bytes that are interpreted on the target, use of the JTAG interface by the host to communicate with the target requires far less cooperation in the target, and hence less response code. Basically, the host asserts control over the CPU using the JTAG interface, performs a desired function such as examining memory or registers, and releases control. These functions, which are frequently useful in hardware bring-up, are described in Section 3.4.4.

When the Debug facility in SwiftX is launched, the word **`RELOAD`** compares the compiled object code checksum with the current contents of flash memory in the target. If the checksums differ, a new target program is downloaded using the JTAG port[†]. The target is then activated via its "power-up" sequence, whereupon it transmits its start-up greeting to the host for display. Thereafter, the target is awaiting commands from the host, which it will execute. During the course of its execution, the target may request services (e.g., keyboard input or display functions) from the host, as described in Section 3.4.2.

---

*. *IEEE Std 1149.1- 1990* (Includes *IEEE Std 1149.1a- 1993*), IEEE Standard Test Access Port and Boundary- Scan Architecture, ISBN 1-55937- 350- 4, IEEE order number SH16626 (the official document that specifies the international standard for a test access port and boundary- scan architecture. Informally known as the JTAG standard, it was officially ratified by the IEEE in February 1990. Since then, it has been supplemented twice. The first supplement, ratified in June 1993, is included in the referenced document. The second supplement is currently a separate document, as referenced below).
IEEE Std 1149.1b- 1994, Supplement to *IEEE Std 1149.1- 1990*, ISBN 1-55937- 497- 7, IEEE order number SH94256 (the official document which specifies the international standard for a boundary- scan description language. This supplement to IEEE Std 1149.1- 1990 was ratified in September 1994).

†. In the rare case in which the program has been changed in a way that does not affect the checksum, you can force a download using the command **`RELOAD!`**.

Details of the JTAG protocol may be obtained from Texas Instruments.

3.4.1 Host-to-target Communication

The host communicates to the target using the JTAG interface to perform the following functions:

- Read and write memory
- Read and write registers
- Force the target to begin executing at a specified address

The data stack is passed to the target with each host command, and returned with its response. The stack is passed bottom item first. When sent from the host to the target, the top stack items are the execution address for the function and the host's **`BASE`**. The target executes the function using **`CATCH`**. Upon return from the target to the host, the top stack item is the function's **`THROW`** code (0 indicates successful completion).

3.4.2 Target-to-host Communication

The host can supply terminal I/O services to the target via the JTAG cross-target link while the target is performing a host command. The target enters a loop at the address contained in **`'XTL`** to indicate that it has either finished performing a host command or that it is requesting a host service. The function code for the requested service is read from the target CPU's register **`R5`** (**`T`**). Parameters are read from the target task's data stack (**`R4`** is the data stack pointer).

Table 7: Target-to-host responses

Fn	Description	Parameters from target
255	Announce reset	
254	Ack (command completed successfully)	
253	Nak (command aborted)	

Table 7: Target-to-host responses *(continued)*

Fn	Description	Parameters from target
252	**EMIT** (display character)	*char*
251	**TYPE** (display string)	*addr len*
250	**CR** (new-line function)	
249	**PAGE** (clear-screen function)	
248	**AT-XY** (cursor position)	*col row*
247	**KEY** (input key and send it back)	0 (place to write key)
246	**KEY?** (test for keypress, send status back)	0 (place to write key status)
245	Display contents of address	*addr*
244	**ACCEPT** (input string, return actual length)	*addr len* (*len* updated with actual)

The target-host XTL protocol uses register **T** to pass a function code from the target to the host when target enters the **REPLY** loop. Table 7 lists the possible function codes from the target, with other parameters, if any, returned in registers as indicated. Any other value returned in **T** indicates abnormal entry to background mode.

Break

If the program under test never returns control to the host, pressing the red Break toolbar button (or File > Break) will display the message BREAK and abort out of the wait loop. This is useful when testing a routine whose behavior is an infinite loop, or when a program behaves unexpectedly.

3.4.3 Register Display and Address Illumination

The presence of the JTAG interface gives this system the ability to examine registers. The command **R.** (see glossary below) displays the registers; the following is an example of the display it generates:

```
R.
PC  044A  SP  0444  SR  0009
R4  03B2  R5  1DAE  R6  0446  R7  75CB
R8  2032  R9  234E  R10 3FFF  R11 CB76
R12 D59F  R13 F976  R14 0E33  R15 7FA3
```

Register display can be useful if a program crashes during testing. It may be possible through successive **R.** displays to determine what the target is doing. The data and return stack pointers (registers **R4** and **SP**) and the **PC** offer primary clues. In a multitasking application, the user pointer (register **R6**, or **U**) is also of interest.

Of course, for this information to be of real use, you would want to know in what definition an address such as the contents of the return stack falls. The word **.'** provides *address illumination*: given an address, it will show what definition the address is in. For example, you could type:

```
HEX 1AB6 .'
```

and get:

```
DIGIT +08  ok
```

showing that the address is 8 bytes into the definition **DIGIT**.

Glossary

R. *(—)*

Display target CPU registers.

.' *(addr —)*

Display the name of the target definition before *addr*, and *addr*'s offset within that definition. Pronounced "dot-tick."

3.4.4 Using the JTAG Interface for Hardware Diagnostics

The JTAG interface can also be helpful in diagnosing possible hardware problems, particularly in a newly manufactured board. The commands described in this section are primarily intended as low-level support for the host side of the XTL, and hence are found in the **INTERPRETER** scope. They don't require the cooperation of any running software on the target. By using these low-level commands, you can probe your hardware even before you have a SwiftX kernel running on it.

There are two basic functions useful in this situation: reading and writing memory, and reading registers. It is also useful to be able to download a string of code or data to the target, and to start the target executing.

Memory reads and writes are accomplished by a set of "fetch" and "store" operators analogous to their Forth counterparts, except that the suffix **(J)** indicates they are JTAG-only operators that act directly on target addresses. The available operators include: **W@(J) C@(J) W!(J) C!(J)**.

Finally, **DOWNLOAD(J)** will send a string of cells from a given host address to a given target address. Host addresses may be obtained in a number of ways: data objects defined in **HOST** scope, standard host scratch areas such as **PAD** (invoked in **HOST** or **INTERPRETER** scope), or by converting a target address to a host address using either **>IDATA** or **>CDATA** (depending on which kind of section the address is in).

Glossary

RESET *(—)*

Asserts the reset line to the board connected through the JTAG port for 100 ms. Equivalent to pressing a "reset" button. This causes the target to start up from its hardware power-up vector, and it will not activate the XTL.

RESET-JTAG *(—)*

Establish JTAG control over the target CPU. Must be used before any of the other commands in this section. Displays "No target" if no device responds on standard parallel port addresses.

C@(J) *(addr — b)*

Fetch a byte from *addr* in the target using the JTAG port only.

W@(J) *(addr — x)*

Fetch a 16-bit cell (a "word" from the perspective of the 32-bit host) from *addr* in the target using the JTAG port only.

C!(J) *(b addr —)*

Store a byte at *addr* in the target using the JTAG port only.

W!(J) *(x addr —)*

Store a 16-bit cell at *addr* in the target using the JTAG port only.

>IDATA, **>CDATA** *($addr_1$ — $addr_2$)*

Convert target address $addr_1$ in iData or cData, respectively, to host address $addr_2$, the actual address in the host's target image.

DOWNLOAD(J) *($addr_1$ $addr_2$ u —)*

Using the JTAG port, download *u* bytes from $addr_1$ in the host to $addr_2$ in the target. The destination address must be even, and *u* will be rounded to the next even number, as the transfer is an integral number of cells.

3.4.5 Breakpoint Support

SwiftX does not use the JTAG interface for breakpoints in debugging. Instead, it provides a word **BREAKPOINT** for use between Forth words. This provides an analogous facility, with the added convenience of passing the target's stack and **BASE** to the host when a breakpoint occurs.

To use **BREAKPOINT**, simply place it at some point in a target definition. **BREAKPOINT** can be placed at any level of nesting in a target definition. When it is executed, it returns control to the host just as if the original word being executed had completed normally. At this point, you may interact with the target to learn whatever is of interest at the breakpoint—taking great caution not to break anything that would prevent completion of the code following the **BREAKPOINT**. For example, you may examine the stack non-destructively by typing **.S**, you may examine the contents of **VARIABLE**s or other data structures, and you may even change the stack or stored data values.

RESUME will return control after the **BREAKPOINT** and allow completion of the target definition. Multiple instances of **BREAKPOINT** may be executed in a series but must not be nested, because the **BREAKPOINT** and **RESUME** sequence is not reentrant.

Glossary

BREAKPOINT *(—)*

Cease executing, saving the current return stack pointer, and enter a new instance of the JTAG command loop to return control to the host.

RESUME *(—)*

Return out of the current instance of the JTAG command loop and back to the point where **BREAKPOINT** was called. Does nothing if no breakpoint was set.

References **.S** and other debugging words, *SwiftX Reference Manual*, Section 2.4

3.5 Low power Mode

The MSP430 was designed for ultra-low power applications. SwiftX supports low power mode by modifying the SwiftOS multitasker to enter your selected low power mode whenever no task has been active for one cycle through the round-robin.

References SwiftOS multitasker principles, *SwiftX Reference Manual*, Section 5.
Low power mode in the MSP430, *MSP430x1xx User's Guide*, Section 3.5.
Low power mode in "Distress Signal" demo program, Section 4.3.

3.5.1 Principles of Operation

The SwiftOS multitasker configures tasks in a "round-robin" in which each task's **STATUS** cell contains a jump to the next task. To set a task to become active, you must replace this jump with a call to code to activate it (the specific commands are discussed in Section 3.1.4). Whenever a task performs I/O or awaits an event, it relinquishes the CPU until the I/O is complete or the event occurs. In most embedded applications, I/O operations occur frequently, so this algorithm means that tasks will pause frequently, and service to all tasks is maximized.

When a SwiftX target is launched, it has only one task, conventionally called **OPERATOR**. More tasks may be instantiated in the start-up code, or later. If the application is single-threaded, that is, no additional tasks are used, **OPERATOR** is still defined and running.

SwiftX's support of low power mode in the MSP430 modifies this algorithm slightly, by inserting a tiny bit of code called **<LPM>** (found in **..\MSP430\Lpm.f**) in the round-robin just before **OPERATOR**. This code tests Register **U** (**R6**), which normally contains a pointer to the *last* task to run.

If this register contains zero, it indicates that no task has run through one complete cycle of the round-robin, and the system will enter whatever low power mode you have specified (see Section 3.5.2). If the register is non-zero, it will be set to zero; if any task runs during the next cycle, its address will replace the zero, but if not, the zero will signal a power-down.

It is up to the user to include code in an interrupt routine that can be activated during the specified low power mode to clear the low power mode bit in the processor status register (the top cell on the hardware stack on entry to an interrupt handler). An example may be found in Section 4.3. It is also the user's responsibility to write any custom I/O drivers following the SwiftOS convention that tasks are suspended while awaiting I/O, and disable the current task by calling **`STOP`** or (in code) jumping to **`WAIT`** after initiating the I/O operation, as described in Section 3.1.4.

References Register usage in SwiftX, Section 2.3.

3.5.2 Configuring a Low Power Mode

The file **`..\MSP430\FET430P140\Config.f`** includes the line:

```
LPM3 EQU LPMODE                    \ Establish low power mode
```

This specifies the default low power mode to be Mode 3. Constants specifying all available modes, **`AM`** and **`LPM0`** through **`LPM4`**, are defined in the MCU register file, **`..\MSP430\Reg_14x.f`** (or whatever is appropriate for your chip). You may change this line to specify the mode that is most appropriate for your application.

When you are running a JTAG debugging session, low power mode is not allowed. However, you do not need to change this specification, as SwiftX will automatically override your setting with **`AM`** (Active Mode) when it launches a JTAG debugging session.

3.6 Flash Memory Support

SwiftX supplies simple erase and programming routines for the flash memory on the MSP430, found in the file **`..\Msp430\Flash.f`**. This software resides in the target, which can actually burn its own flash memory.

There are two regions of flash in the MSP430:

- Information memory (1000_H–$10FF_H$) contains two 128-byte segments, A and B, intended to keep persistent application data. These can be individually erased or programmed.
- Program memory (1100_H–$FFFF_H$) is main program storage. It can be erased or programmed in 512-byte segments.

Constants are provided to identify the starting addresses of each of these regions. These and the words for managing flash memory are described in the glossary.

Glossary

`SegmentA` *(— addr)*
Returns the address of the first Information Memory segment (1000_H).

`SegmentB` *(— addr)*
Returns the address of the second Information Memory segment (1080_H).

`SegmentN` *(— addr)*
Returns the address of the start of main program memory (1100_H).

`FLASH-ERASE` *(addr — flag)*
Erase the sector of flash memory that contains *addr*. Returns *true* if the operation was successful. The address must be in flash memory.

`FLASH-PROGRAM` *($addr_1$ $addr_2$ u — flag)*
Write *u* bytes from source address $addr_1$ to destination address $addr_2$. Returns *true* if the operation was successful; returns *false* if the operation times out. The size and destination address must be word-aligned, and the destination address must be in flash memory.

4. Writing I/O Drivers

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

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

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

4.1 General Guidelines

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

1. **Name your device registers**, usually by defining them as **`CONSTANT`**s or **`EQU`**s. This will help make your code more readable. It will also help "parameterize" your driver: for example, if you have several devices that are similar except for their hardware addresses, you can write the common control code and pass it a port or register address to indicate a specific device, efficiently reusing the common code.

   The on-board registers for your microcontroller are named in files whose names are **`\Swiftx\Src\Msp430\Reg_`**`<mcu>`, where *mcu* indicates your variant of the processor (e.g., **`Reg_14x.f`**). Special registers associated with other devices may be named at the beginning of the file containing the driver.

2. **Test the device** before writing a lot of code for it. It may not work; it may not be connected properly; it may not work exactly like the documentation says it should. It's best to discover these things before you've written a lot of code based on incorrect information, or have gotten frustrated because your code isn't behaving as you believe it should!

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

   ```
   P1IN C@ .
   ```

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

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

4.2 Example: System Clock

SwiftX uses the TimerA interrupt driven by a 32.768 kHz watch crystal to provide basic clock services. This provides a good example of a simple interrupt routine. The complete source is in **`Swiftx\Src\Msp430\TimerA.f`**. The high-level date and time functions in SwiftX may be found in **`..\Src\Clock.f`**.

Our first design decision is the units to store: milliseconds, seconds, or just a count of clock ticks. Storing clock ticks provides the simplest interrupt code, but requires the routine that delivers the current time of day to convert clock ticks to time units. In this case, the interrupt rate is 1024 Hz, which is only slightly faster than one millisecond per interrupt, so it will be almost as easy for us to accumulate a fractional component in one cell and an integral number of milliseconds in another. Since the MSP430x14x supports three capture/compare registers with TimerA, we can use **`CCR0`** to accumulate millisecond interrupts and **`CCR1`** to accumulate seconds.

The millisecond interrupt routine looks like this:

```
2VARIABLE MSECS                \ Millisecond counter

1024 EQU T/S                   \ Interrupts per second
32768 1024 / EQU T(MS)         \ Clock ticks per MS

\ K1(MS) is the time constant for the fractional
\ component of the MS accumulator per interrupt tick,
\ K0(MS) is the integer component.
0 1 1000 T/S M*/  $10000 UM/MOD
   EQU K1(MS)  EQU K0(MS)

LABEL <MSEC>                   \ Millisecond interrupt
   T(MS) # CCR0 & ADD          \ Advance counter
   K1(MS) # MSECS CELL+ & ADD  \ Add fraction component
   K0(MS) # MSECS & ADDC       \ Add integer + carry
   CCIFG # CCTL0 & BIC         \ Reset interrupt flag
   RETI   END-CODE             \ End

<MSEC> TIMERA0_VECTOR INTERRUPT    \ Set vector

\ COUNTER returns the current (integer) MS counter
: COUNTER ( -- n )   MSECS @ ;
```

We would like to be able to set a time of day, and have it updated automatically. For this purpose, we'll keep a separate counter measuring whole seconds. This is a 32-bit counter, which will simply cycle indefinitely. Whenever you fetch the current time using **@NOW**, it checks for midnight overflow and increments the system date (in the variable **TODAY**) appropriately. Therefore, if you have set the system date and time using **!NOW**, **@NOW** will always return correct values.

Note that the variables **TODAY** and **SECS** are not intended for direct user access; read and write them using **@NOW** and **!NOW** only.

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).

Glossary

@NOW *(— ud u)*

Return the current day as an MJD (*u*) and number of seconds since midnight (*ud*). Before returning these values, it checks for midnight overflow and adjusts **TODAY** and **SECS** accordingly.

Usage example:

```
@NOW .DATE .TIME
```

!NOW *(ud u —)*

Store the current day as an MJD (*u*) and number of seconds since midnight (*ud*). Before returning these values, it checks for midnight overflow and adjusts **TODAY** and **SECS** accordingly.

Typical usage:

```
13:30:00 HOURS  @TIME 6/01/01 M/D/Y !NOW
```

References Timing functions in SwiftX, *SwiftX Reference Manual*, Section 6.2.

4.3 Example: "Distress Signal" Using Multitasker, Port1, and Low power Mode

SwiftX includes a sample application that uses the LED on the board to send a Morse Code "SOS" distress signal. The source for the application is in **`ForthInc\Projects\Distress\Distress.f`**. It's a good example of a simple application using Port 1 output, the SwiftOS multitasker, and low power mode.

When you launch **`Debug.f`**, the startup initialization (**`GO`** in **`App.f`**) starts a background task sending the distress signal "SOS" in Morse code on the board's LED, which is connected to the low-order bit of Port1. This task will run while you do whatever you wish: examine memory or registers, execute commands, etc.

You can stop the task by typing **`BEACON HALT`**. **`BEACON`** is the name of the task (defined toward the end of **`Distress.f`**), and **`HALT`** causes it to execute an idle behavior instead of running the SOS signal. While the task is halted, you can manually control the LED by typing **`+LED`** to turn it on, **`-LED`** to turn it off, or executing any other words in **`Distress.f`**. You may restart the background task by typing **`/BEACON`**.

This application was developed in several stages:

1. First, we explored the port directly, by typing instructions like the contents of **`/LED`**, **`+LED`**, etc. from the command line. This confirmed that the port was "alive" and the LED worked as we expected.
2. Then we defined **`/LED`**, **`+LED`**, etc., and started putting them together into higher-level words like **`DIT`** and **`DAH`**, and then the highe-level words. We used **`MS`** for timing. We edited these simple definitions into a file, and repeatedly loaded and tested it until **`SOS`** worked as we desired.
3. We then set up the background task described in Section 4.3.3 and "taught" it to run SOS.
4. Finally, we added the feature of letting the application run in low power mode, as described in Section 4.3.2.

References SwiftOS multitasker principles, *SwiftX Reference Manual*, Section 5.

4.3.1 LED Control Using Port1

On TI’s FET430P140 board, there’s a single LED attached to the low-order bit of Port1. Since the MSP430’s I/O is all memory-mapped, it is easy to read and write registers using **C@** and **C!**. This is how the words **/LED**, **+LED**, and **-LED** (to initialize the port and turn the LED on and off, respectively) are defined.

By convention in SwiftOS one should deactivate a task when it performs I/O. Using **C!** as in these examples doesn’t do that. In this application, all uses of these words will be followed by a **DELAY** (Section 4.3.2), so it isn’t necessary. If you were using, for example, **+LED** *without* a **DELAY**, you could follow the **C!** by a **PAUSE**. **PAUSE** will deactivate the task for exactly one lap of the round-robin. It’s appropriate for an operation that occurs instantaneously (i.e., you don’t have to wait for completion). If you have to wait, you should block the task (see Section 3.1.4) and let the interrupt that detects completion wake it up. Here, we are assuming that a **+LED** or **-LED** followed by a **DELAY** is a single conceptual operation, and do our waiting in the interrupt routine initialized by **DELAY**.

4.3.2 Using Low Power Mode

The Distress Signal demo application includes a modified TimerA interrupt handler to illustrate management of low power mode in the MSP430.

The word **DELAY** in **Distress.f** is similar to **MS** (which waits a specified number of milliseconds), except that it saves the desired delay time in milliseconds in the variable **#DLY** and (most important) the address of the task that called it in **'DLY**. **DELAY** also sets the TimerA interrupt vector to special code at **<DELAY>** instead of the default interrupt handler **<MSEC>**. **<DELAY>** compares the current value of **MSECS** maintained by the standard clock interrupt with the desired value; if the time has expired, it wakes up the task, resets the interrupt vector to the normal **<MSEC>**, and clears the low power mode bit in the processor status register (which is on top of the stack on entry to the interrupt routine). Finally, whether time has expired or not, it branches to **<MSEC>** to complete the standard processing of the clock tick.

Note that the last word in **DELAY** is **STOP**. This suspends the task executing **DELAY** until the interrupt routine detects that time has expired and awakens it.

If no other tasks are running, the processor will go into low power mode very quickly.

By having **DELAY** save the address of the task running this special timer, it can be used with any task (in this case, either **OPERATOR** or **BEACON**). If a device is only controlled by a single task, you can build the task to be awakened into the interrupt routine.

To see this little application run in low power mode, execute **RESET**. This **HOST** function (described in Section 3.4.4) is equivalent to doing a hardware power-up, without activating the XTL.

4.3.3 Multitasking Usage

This application uses a separate task called **BEACON** to run the distress signal. **BEACON** is defined at the end of **Distress.f**, as a background task. The definition **/BEACON** initializes the **LED** and starts **BEACON** executing **SOS** in an infinite loop.

But before we can execute **/BEACON** the task has to be instantiated in RAM. Whereas the task is defined when the application is compiled, the task cannot be instantiated until run time. Although we might want to start the task and stop it (using **BEACON HALT**, as described above), it can only be instantiated once. This is done by the phrase **BEACON BUILD** in the start-up word **GO** in **App.f**.

Since the code is written such that it can be executed by any task, you could conceivably run **SOS** from the command line (by the **OPERATOR** task) while **BEACON** is executing it repeatedly. Since no provision is made to “lock” the LED while it’s in use, the result will probably be a flawed sequence. In theory, you could establish an interlock using the words **GET** and **RELEASE** described in the *SwiftX Reference Manual*, Section 5.2.4. However, in a real application it would be more appropriate to have a single task responsible for performing this function, and avoid the unnecessary complexity of interlocks. Interlocks should be used only to control resources that multiple tasks *must* share (to avoid even greater complexities elsewhere).

References Defining and initializing background tasks, *SwiftX Reference Manual* Section 5.3.

Appendix A: TI FET430P140 Board Instructions

This section provides information pertaining to the TI FET430P140 Starter Kit, which is supported by SwiftX for the MSP430. 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 and CPU Description

The TI FET430P140 Starter Kit is designed to help new users get acquainted quickly with the MSP430 microcontroller family.

A.1.1 Board Features

The basic starter kit includes:

- MSP430F13x/14x FET development board with ZIF socket, LED indicator, FET to PC adapter, and header pinouts for prototyping
- Two MSP430F149 flash devices

The FET-to-PC adapter is a “smart” cable that connects the JTAG port on the chip to the PC’s parallel (printer) port. It is used by SwiftX for the Cross-Target Link (XTL). SwiftX does not use the IAR software tools supplied with the kit.

Processor features include:

- Low supply-voltage range, 1.8 V . . . 3.6 V
- Ultralow-power consumption:

- Standby mode: 1.6 µA
- RAM retention off mode: 0.1 µA

- Low operating current:
  - 2.5 µA at 4 kHz, 2.2 V
  - 280 µA at 1 MHz, 2.2 V
- Five power-saving modes
- Wake-up from standby mode in 6 µs
- 16-bit RISC architecture, 125-ns instruction cycle time
- Two 8-bit I/O ports, each with an interrupt
- Four 8-bit I/O ports, without interrupts
- 12-bit A/D converter with internal reference, sample-and-hold and autoscan features
- 16-bit timer with seven capture/compare-with-shadow registers, Timer_B
- 16-bit timer with three capture/compare registers, Timer_A
- On-chip comparator
- Serial onboard programming, no external programming voltage needed
- Programmable code protection by security fuse
- MSP430F149 includes:
  - 60KB+256B flash memory,
  - 2KB RAM

A.1.2 FORTH, Inc. Board Modifications

FORTH, Inc. modifies the boards shipped by TI by adding a 32.768 kHz watch crystal to provide the clock services described in Section 4.2. Neither the crystal nor the clock software is required by the kernel; they are provided primarly for user convenience and as an example.

If you do not wish to use this feature, just omit **`Clock.f`** and **`Timedate.f`** from your **`Kernel.f`** file.

A.1.3 Support for Other Microcontrollers

The FET430P140 Flash Emulation Tool can support all of the MSP430F13x/14x processors. All you have to do is replace the MSP430F149 with the device of your choice. As shipped, SwiftX provides configurations for a number of members of this family. Although we recommend doing as much development as possible with the MSP430F149 to take advantage of its extra program RAM (which is extremely convenient during development), when you are ready to work with your actual target suitable register and configuration files are available.

To launch SwiftX for a different member of the MSP430 family, just select it from the Start menu, as shown in Figure 1.

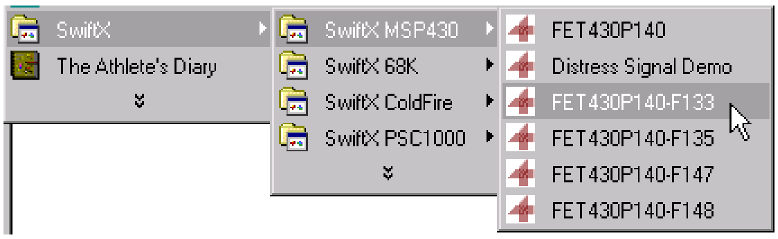

Figure 1. Launching members of the MSP430 family

Memory configuration and utilization for these processors is discussed in Section A.4.1.

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, and the SwiftX kernel will be installed in the flash memory. The CD-ROM supplied by TI with the Starter Kit is not needed for SwiftX development.

Connect your board as described in the FET430P140 manual. The parallel cable should be connected to your printer port. It is used for flash programming and the SwiftX XTL.

A.3 Development Procedures

On the MSP430, the SwiftX kernel resides in flash memory. Although there is ample flash memory available, RAM is quite limited. As a result, procedures for developing and testing applications may differ from those described in the generic SwiftX manual and from those for other SwiftX systems.

A.3.1 Starting a Debugging Session

Running an MSP430 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.

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, downloading a new program if necessary.

Debug

Whenever you launch Project > Debug (from the menu item or toolbar button) the command **`RELOAD`** will compare the most recently compiled program with the checksum in the target's flash. If they do not agree, the PC's program image will automatically be written to the target's flash.

If you get the message, `Kernel Mismatch`, it indicates that the flash programming step has failed for some reason. In most cases, a retry will be successful. If repeated retries fail, it may indicate that your board's flash memory is defective.

If the host failed to establish XTL communication with the target, you will get the error message, "No target." If this happens, check your connections carefully and try again.

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*, to examine your host’s image of target CData or IData.

A.3.2 Running the Demo Applications

Debug

As shipped, SwiftX includes a sample applicationthat uses a background task to display a Morse Code “SOS” distress signal on the board’s LED. The application is described in detail in Section 4.3. To run it, launch the SwiftX “project” icon in the **ForthInc\Projects\Distress** directory. The interactive testing program **Debug.f** (loaded by the Project > Debug menu item or Toolbar button) is configured to run a multitasking demo application that is loaded by the file **..\Distress\App.f** when you build a kernel for downloading as described in Section A.3.1.

If you wish, you can also run the “Conical Pile Calculator” demo program described in the SwiftX Reference Manual. This demo program uses the host keyboard and screen via the XTL for its user interface.

To run the “Conical Pile Calculator,” edit **App.f** to **INCLUDE %SWIFTX\SRC\Conical** instead of **Distress**, and remove **BEACON BUILD** and **/BEACON** from **GO**. Select Project > Debug to bring up the target program. Type **CALCULATE** to start the application.

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 TI FET430P140.

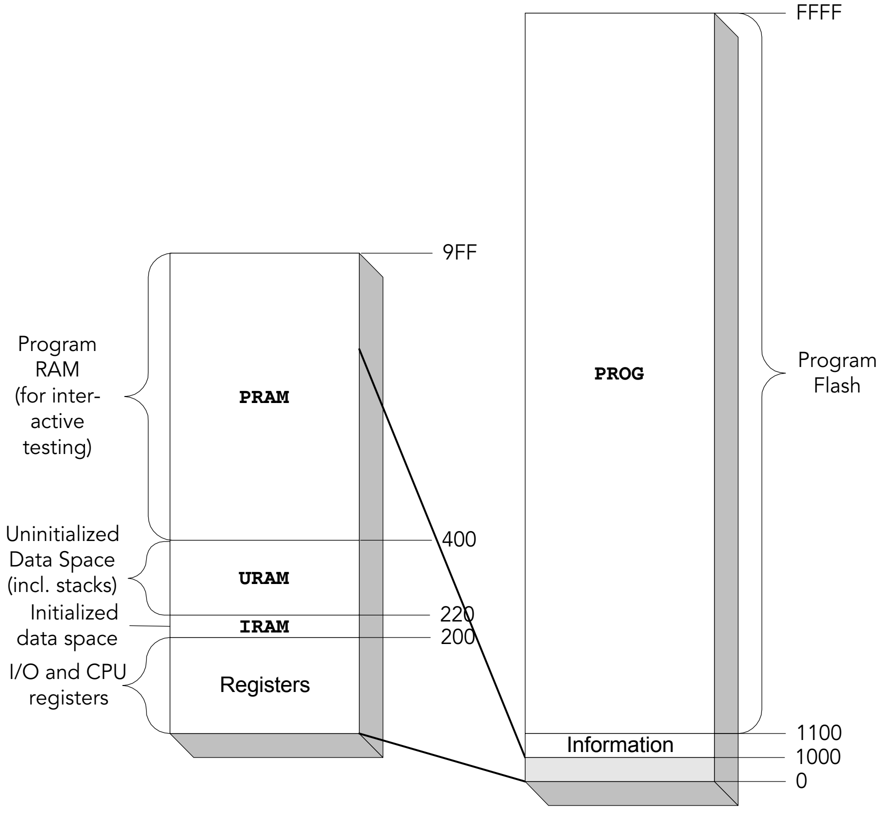

Figure 2. Memory organization in the MSP430F149

A.4.1 Memory Configuration

The TI FET430P140 with an MSP430F149 processor provides memory as shown in Figure 2.

Memory configuration for the each processor supported is set by a set of **SECTION** definitions in the **..\MSP430**<project>**\Config.f** file. You may adjust them if necessary, to suit your needs. The default configurations are shown in Table 8.

Table 8: Default memory configuration for MSP430 MCUs

MCU	PROG Flash cData	IRAM RAM iData	URAM RAM uData	PRAM RAM cData
MSP430F133	8192	16	240	0
MSP430F135	16384	16	496	0
MSP430F147	32768	32	480	512
MSP430F148	49152	32	480	1536
MSP430F149	61184	32	480	1536

As shipped, the SwiftX kernel occupies about 6K bytes of code space in **PROG** plus four bytes of **IRAM**. The MSP430F133 implementation has only a flash set of interrupt vectors (see Section 2.8) since its RAM space is so limited, whereas the others have interrupt vectors in RAM, referenced from flash. The SwiftX kernel uses about 200 bytes of **URAM** on the MSP430F133, and 300-400 bytes on the others. Note that SwiftX as shipped does not use the Information Memory pages at all, leaving them free for applications.

SwiftX configures a small amount of RAM for cData on the MSP430x14x parts. This allows you to type in definitions and exercise them immediately, or even load small progam files. However, since this space is so limited and burning flash is quick and easy, the normal development mode is to perform preliminary testing on small amounts of code in RAM, then add it to your growing application which is loaded by the file ****<project>**\App.f**, which itself is loaded at the end of the kernel, in ****<project>**\Kernel.f**. Any initialization that must be performed at power-up should be included in ****<project>**\Start.f**, in the word **START** prior to the call to **GO** (which launches the XTL).

General Index