SwiftX AVR

for the Arduino Prototyping Platform

FORTH, Inc.

Software products and services since 1973
www.forth.com

Current revision: February, 2011

FORTH, Inc.
Los Angeles, California USA
www.forth.com

SECTION 1: OVERVIEW

This paper discusses the SwiftX Interactive Development Environment (IDE) specific to the Atmel AVR family of microcontrollers. It demonstrates the use of SwiftX in the development of a very simple embedded application, a Morse code beacon that flashes out the universal distress signal "S.O.S." on an LED.

The application is ported to two of the popular Arduino hardware prototyping platform boards, the Uno and the Diecimila. SwiftX is used here as an alternative to the Arduino "sketches" development environment.

1.1 About SwiftX

SwiftX is FORTH, Inc.'s interactive cross compiler, a fast and powerful tool for the development of software for embedded microprocessors and microcontrollers. SwiftX is based on the Forth programming language and is itself written in Forth.

SwiftX is available for the following microprocessor and microcontroller families:

- Atmel AVR
- Freescale 68HC11
- Freescale 68HC12 (S12, S12X, etc.)
- Freescale 68HCS08
- Freescale 68K
- Freescale 6801 / Renesas 6303
- Freescale 6809
- Aeroflex UTMC 69R000
- Intel (NXP, SiLabs, others) 8051
- Atmel, Cirrus, NXP, ST Microelectronics (and other) ARM core
- Freescale ColdFire
- Renesas H8H (H8/300H, H8S)
- Intel (AMD, other) i386
- Texas Instruments MSP430
- Patriot PSC1000
- Harris RTX2010

The suite of SwiftX cross compilers are themselves applications that run in the SwiftForth for Windows programming environment. As such, they inherit all the features of SwiftForth and extend its interactive development environment to manage multiple program and data spaces as well as generate the code and data that fill them. The SwiftForth and SwiftX host environments as well as all target implementations are written ***entirely in Forth***.

SwiftX provides the user with the most intimate, interactive relationship possible

with the target system, speeding the software development process and helping to ensure thoroughly tested, bug-free code. It also provides a fast, multitasking kernel and libraries to give you a big head start in developing your target application.

1.2 Evaluation Boards

SwiftX is available ready to run on a wide variety of off-the-shelf evaluation boards. Two of the popular Arduino boards will be used in this sample demo application.

The Arduino Uno board features the Atmel AVRmega328P microcontroller, with 32 kB of flash memory, 2 kB SRAM, and 1 kB EEPROM.

The older Arduino Diecimila board has the Atmel AVRmega168 microncontroller, with 16 kB flash, 1 kB SRAM, and 512 bytes EEPROM.

Details about the Arduino boards are provided in the next section.

Section 2: Arduino Platform

This section describes the Arduino boards used in the demo application. The SwiftX IDE replaces the Arduino development environment, so this section only discusses the Arduino hardware platform.

2.1 About Arduino

The remander of this subsection is adapted from the Arduino project's online documentation at http://www.arduino.cc/en/Guide/Introduction.

2.1.1 What is Arduino?

Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer. It's an open-source physical computing platform based on a simple microcontroller board.

Arduino can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs. Arduino projects can be standalone, or they can communicate with software running on your computer. The boards can be assembled by hand or purchased preassembled.

2.1.2 Why Arduino?

There are many other microcontrollers and microcontroller platforms available for physical computing. Parallax Basic Stamp, Netmedia's BX-24, Phidgets, MIT's Handyboard, and many others offer similar functionality. All of these tools take the messy details of microcontroller programming and wrap it up in an easy-to-use package. Arduino also simplifies the process of working with microcontrollers, but it offers some advantage for teachers, students, and interested amateurs over other systems:

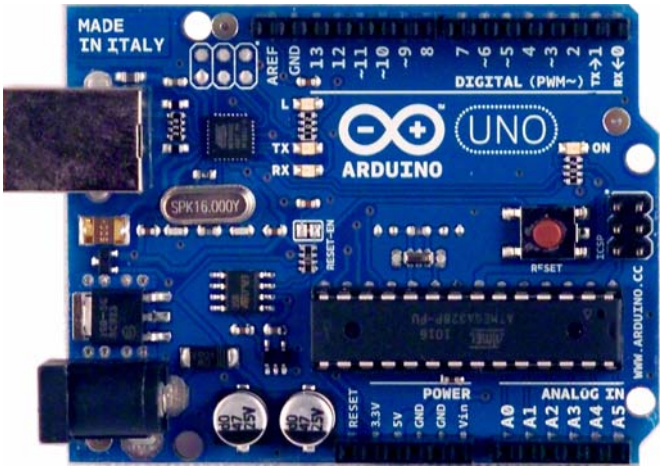
- Inexpensive - Arduino boards are relatively inexpensive compared to other microcontroller platforms. The least expensive version of the Arduino module can be assembled by hand, and even the preassembled Arduino modules cost less than $50.
- Open source and extensible hardware - The Arduino is based on Atmel's AVR microcontrollers. The plans for the modules are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works.

2.2 Arduino Uno

2.2.1 Uno Board Overview

The Arduino Uno board (see Figure 1) is a microcontroller board based on the Atmel ATmega328P. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button.

Figure 1. Arduino Uno Board



2.2.2 Uno Board Features

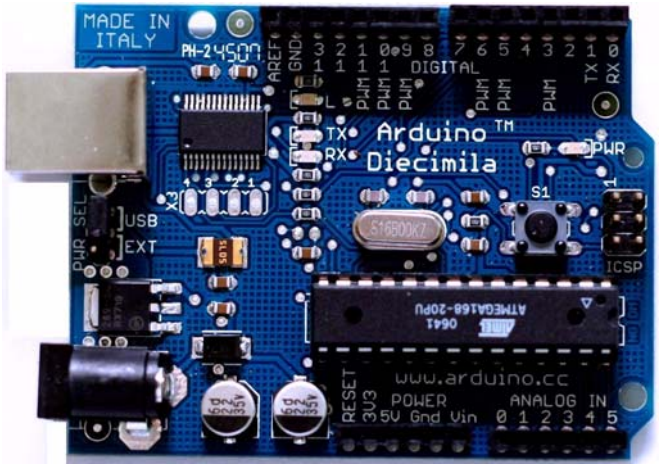
Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	14 (up to 6 PWM outputs)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Memory (flash, SRAM, EEPROM)	32 kB, 2 kB, 1 kB
Clock Speed	16 MHz

2.3 Arduino Diecimila

2.3.1 Diecimila Board Overview

The Arduino Diecimila (see Figure 2) is a microcontroller board based on the Atmel ATmega168. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button.

Figure 2. Arduino Diecimila Board



2.3.2 Diecimila Board Features

Microcontroller	ATmega168
Operating Voltage	5V
Input Voltage (recommended)	7-12 V
Input Voltage (limits)	6-20 V
Digital I/O Pins	14 (up to 6 PWM outputs)
Analog Input Pins	6
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Memory (flash, SRAM, EEPROM)	16 kB, 1 kB, 512 bytes
Clock Speed	16 MHz

2.4 Arduino Board Documentation

Schematics for the Arduino Uno and Diecimila boards are attached to the end of this document. Further details about these boards can be found on the Arduino project web site:

http://arduino.cc/en/Main/Boards

Section 3: Implementation Issues

This section covers specific issues involving this implementation of SwiftX on the AVR family of processors.

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, look at the source file **SwiftX/src/avr/code/core.f.**

3.1.2 Code Optimization

More extensive optimization is provided by a rule-based optimizer, included with

SwiftX Pro, that can optimize a number of common high-level phrases. This optimizer is normally running, but can be turned off for debugging or comparison purposes. For example, consider this test definition:

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

With the optimizer turned off, you would get:

```
SEE TRY
176A   (LITERAL) CALL            0E943000
176E   LITERAL 4
1770   R26 R26 ADD               AA0F
1772   R27 R27 ADC               BB1F
1774   + CALL                    0E948B01
1778   (LITERAL) CALL            0E943000
177C   LITERAL 7
177E   AND JMP                   0C946A01 ok
```

But with it turned on, you would get:

```
SEE TRY
1762   8 R26 ADIW                1896
1764   0 R27 ANDI                B070
1766   7 R26 ANDI                A770
1768   RET                       0895 ok
```

This code is significantly smaller and faster because it has pre-applied **CELLS** to the literal **4** and in-lined the **+** and **AND**.

You can manually control the optimizer using the commands **+OPTIMIZER** to turn it on (the default condition) and **-OPTIMIZER** to turn it off.

3.1.3 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 64 kB each of directly addressable code and data space.

Because all definitions must reside in on-chip flash memory (the only code space available), all definitions must be in the kernel, which is downloaded to flash as a single operation via the ISP module. It is not practical to add definitions interactively, as with targets that support execution of code in RAM.

Some members of the ATmega group of parts provide extra flash memory. For example, the ATmega128 provides 128 kB of flash. However there are restrictions on how this flash can be used, as the Forth 16-bit implementation doesn’t support directly addressing more than 64 kB.

3.1.4 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.

3.1.5 Multitasker Implementation

A task switch in the SwiftX multitasker 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.

3.2 I/O Registers

Refer to the Atmel data sheet for your particular MCU for details regarding the use of the I/O registers.

SwiftX defines names for the registers, corresponding to their Atmel designations, in a file for each MCU. The files' names take the form **SwiftX\src\avr\reg_<mcu>.f**.

The I/O registers $0–3F_H$ are mapped to data space addresses $20–5F_H$. The register names 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).

In addition to defining the registers, SwiftX has also defined individual bit masks

for defined bits in many of these registers.

For example:

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

Note in this example (from **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 a jump instruction 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, because it generates code in flash memory.

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

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.

Glossary

INTERRUPT (addr1 addr2 –)

Install *addr1* in the interrupt vector at *addr2*.

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 interrupt. 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**, 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
```

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 **SwiftX\src\avr\code\serial.f** and may be used as an example for an application using the serial port.

Section 4: SwiftX AVR Assembler

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 or ATmega103. In general, each executes a subset of the instructions defined for the whole family, and varies by the amount of internal memory and which I/O devices it supports. Consult the user's manual for your particular variant of the AVR family for the specific instructions that processor will accept. For convenience in this book, we will refer to the AVR only.

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 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.

4.1 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 AVR 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 4 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

4.2 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 4.3. In this example, the macro **TPUSH** pushes a copy of the top data stack item, which is kept in register pair **XH:XL**, onto the stack, and the two **LDD** instructions put a copy of *x1* there.

As an alternative to the normal **RET**, 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 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 labeled 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 4.5
Interrupt handling, Section 3.3
SwiftOS multitasking executive, *SwiftX Reference Manual*, Section 5

4.3 Registers

The AVR's registers 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. They are shown in Figure 3 and Table 1.

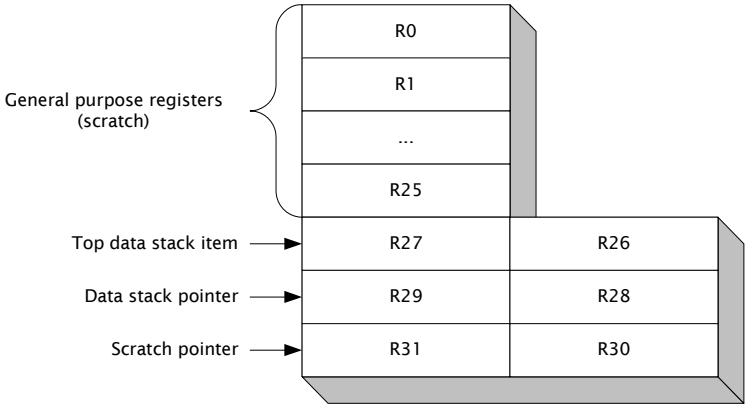


Figure 3. Register usage in AVR SwiftX

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 are explicitly dealing with the top stack item).

Table 1: Special register assignments

Register(s)	Atmel name	SwiftX name	Description
R27:R26	**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)**	(None)	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. Because 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 immediate 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
```

4.4 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>`.

4.4.1 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.

4.4.2 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 2: 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 byte operands (e.g., **XL** and **XH**) when loading or storing them. Use the low-order register of a pair for **ADIW** and **SBIW** instructions:

```
R24   R26    R28    R30
       XL     YL     ZL
```

4.4.3 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

4.5 Macros

The macros in Table 3 have been defined in order to simplify assembler coding for the 16-bit virtual machine.

Table 3: 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**

4.6 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 4 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 4: 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.

4.7 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 4 and in a glossary beginning on page 23. 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 that 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 4 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 23.

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 (addr1 cc – addr2 addr1)

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 (addr2 addr1 –)

Set the destination address of the branch that is at *addr1* (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 *addr2* (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 (addr1 – addr2)

Set the destination address *addr1* of the preceding **IF** to the next word, and assemble an unconditional branch (with unspecified destination) whose address *addr2* 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 (cc1 – cc2)

Invert the condition code *cc1* to give *cc2*.

4.8 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 struc-

tured 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 aren't implemented in all AVR microcontrollers. See your MCU manual for details.

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

```
LABEL <name>
```

…described in Section 4.2. 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
```

SECTION 5: DEMO APPLICATION

Keeping portability in mind, the simple Morse code "S.O.S." application is structured in two layers. The lower layer supplies the hardware application programming interface. The upper layer is the application itself.

5.1 About Morse Code

The material in this section is excerpted from the Morse code topic on Wikipedia. The complete topic with all its technical details, illustrations, and references can be found here:

Reference http://en.wikipedia.org/wiki/Morse_code

Morse code is a method for transmitting telegraphic information, using standardized sequences of short and long elements to represent the letters, numerals, punctuation and special characters of a message. The short and long elements can be formed by sounds, marks, or pulses and are commonly known as "dots" and "dashes" or "dits" and "dahs".

International Morse code is composed of six elements:

1. short mark, dot or "dit" (·)
2. longer mark, dash or "dah" (-)
3. intra-character gap (between the dots and dashes within a character)
4. short gap (between letters)
5. medium gap (between words)
6. long gap (between sentences — about seven units of time)

These six elements serve as the basis for International Morse code and therefore can be applied to the use of Morse code world-wide.

Morse code can be transmitted in a number of ways: originally as electrical pulses along a telegraph wire, but also as an audio tone, a radio signal with short and long tones, or as a mechanical or visual signal (e.g. a flashing light) using devices like an Aldis lamp or a heliograph. Morse code is transmitted using just two states (on and off) so it was an early form of a digital code. However, it is technically not binary, as the pause lengths are required to decode the information.

The length of the dit determines the speed at which the message is sent, and is used as the timing reference.

The speed of Morse code is typically specified in words per minute (WPM). A dah is conventionally three times as long as a dit. The spacing between dits and dahs within a character is the length of one dit; between letters in a word it is the length of a dah (three dits); and between words it is seven dits. The "Paris" Morse code standard defines the speed of transmission as the dot and dash timing needed to send the word "Paris" a given number of times per minute. The word "Paris" is used

because it is precisely 50 “dits” based on the textbook timing.

Under this standard, the time for one “dit” can be computed by the formula:

T = 1200 / W

Where: W is the desired speed in words-per-minute, and T is the duration of one dit in milliseconds.

5.2 Driver Layer Development

Morse code can be transmitted in bursts of audio tone, continuous wave radio frequency (CW RF), direct current over a wire, or light. In this application, we will use a LED on the Arduino board to provide an “S.O.S.” beacon.

The LED driver layer needs to supply three basic functions:

- Turn the LED on
- Turn the LED off
- Hardware initialization

We will define these Forth words to supply those functions:

Glossary

+LED (--)

Turn the LED on.

-LED (--)

Turn the LED off.

/LED (--)

Perform any necessary hardware initialization for LED output.

As shown in the attached schematics, both Arduino boards have an on-board LED connected to GPIO port B, bit[5]. We also see from the schematic that the LED is active high. The default function and mode for each pin is GPIO, input. Therefore, our initialization routine needs to set PB5 as an output with an initial state of logic 0:

```
CODE /LED ( -- )   5 DDRB SBI   5 PORTB CBI   RET   END-CODE
```

Turning the LED on and off is accomplished by setting PTF0 low and high, respectively:

```
CODE +LED ( -- )   5 PORTB SBI   RET   END-CODE
CODE -LED ( -- )   5 PORTB CBI   RET   END-CODE
```

Note the use of **CODE** definitions to take advantage of the nice I/O bit manipulation operators in the AVR instruction set.

5.3 Application Layer Development

The Morse code “S.O.S.” application can be divided into sections:

- Timing functions
- Primary code elements (“dit” and “dah”)
- Character codes
- Message

5.3.1 Timing Functions

Recall from the discussion of Morse code in Section 5.1 that the speed of Morse code transmission is typically specified in words per minute (WPM) and that the timing for a dah is conventionally 3 times as long as a dit. The spacing between dits and dahs within a character is the length of one dit; between letters in a word it is the length of a dah (3 dits); and between words it is 7 dits.

Under the “Paris” standard, the time for one "dit" can be computed by the formula:

```
Tu = 1200 / W
```

Where: W is the desired speed in words-per-minute, and Tu is one dit-time in milliseconds.

So our basic timing and WPM setting functions might look like this:

```
CREATE Tu 120 ,

: WPM ( n -- )   1200 SWAP /  Tu ! ;
: DELAY ( n -- )   Tu @ * MS ;
```

Tu holds the current value of one "dit" time. The default value of 120 sets the initial rate at 10 WPM as defined above.

WPM sets **Tu** based on the formula above.

DELAY pauses for *n* dit times using the standard SwiftX **MS** (millisecond delay) function.

5.3.2 Primary Code Elements

Using our LED API calls (**+LED** and **-LED**) along with **DELAY** timing defined above, we can build the **DIT** and **DAH** code elements:

```
: DIT ( -- )   +LED  1 DELAY  -LED  1 DELAY ;
: DAH ( -- )   +LED  3 DELAY  -LED  1 DELAY ;
```

Note that the one-unit intra-character delay time trails each code element.

5.3.3 Character Codes

Only the codes for letters “S” (dit-dit-dit) and “O” (dah-dah-dah) are required here.

```
: S ( -- )   DIT DIT DIT  2 DELAY ;
: O ( -- )   DAH DAH DAH  2 DELAY ;
```

Note the additional **2 DELAY** at the end of each character to supply the total three-unit inter-character delay time.

5.3.4 The Distress Signal

```
: SOS ( -- )  S O S  4 DELAY ;
```

Again, note the **4 DELAY** at the end, which results in gap that is seven units in duration.

5.4 Background Task Assignment

We finish by assigning the infinite loop **SOS** output to a background task using the SwiftOS multitasker, available in all SwiftX implementations.

First, we define the task:

```
|U| |S| |R| BACKGROUND BEACON
```

This defines a background task named **BEACON** with initial user area size **|U|**, data stack size **|S|**, and return stack size **|R|**. These contants (“size of U”, “size of S”, and “size of R”) are defined for each SwiftX implementation. They define “full-size” user and stack spaces. In applications tight for RAM, smaller values may be used as needed.

Next, we define an initialization procedure that performs the hardware setup and assigns the task’s behavior:

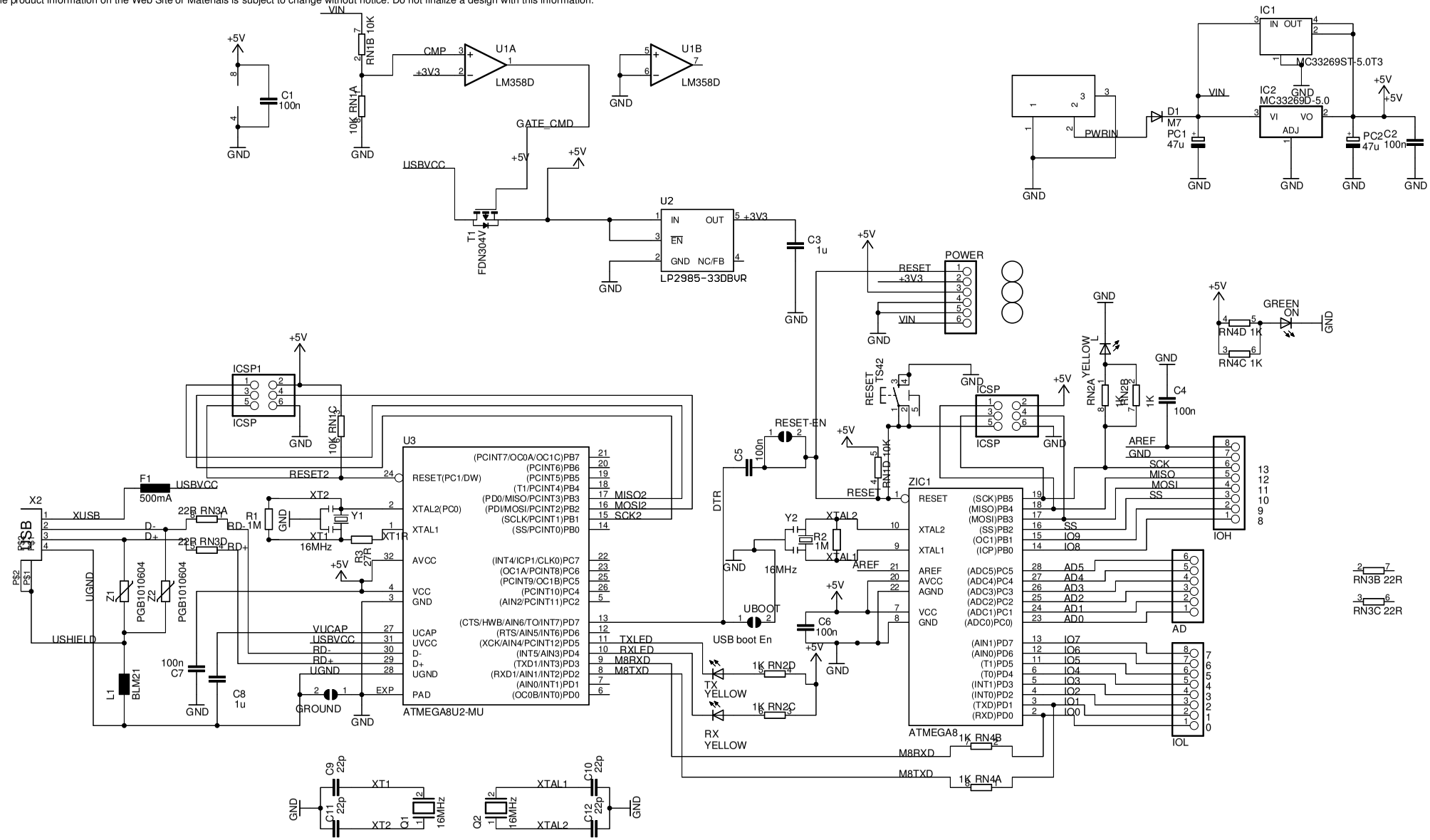
```
: /BEACON ( -- )   /LED
   BEACON ACTIVATE  BEGIN  SOS  AGAIN ;
```

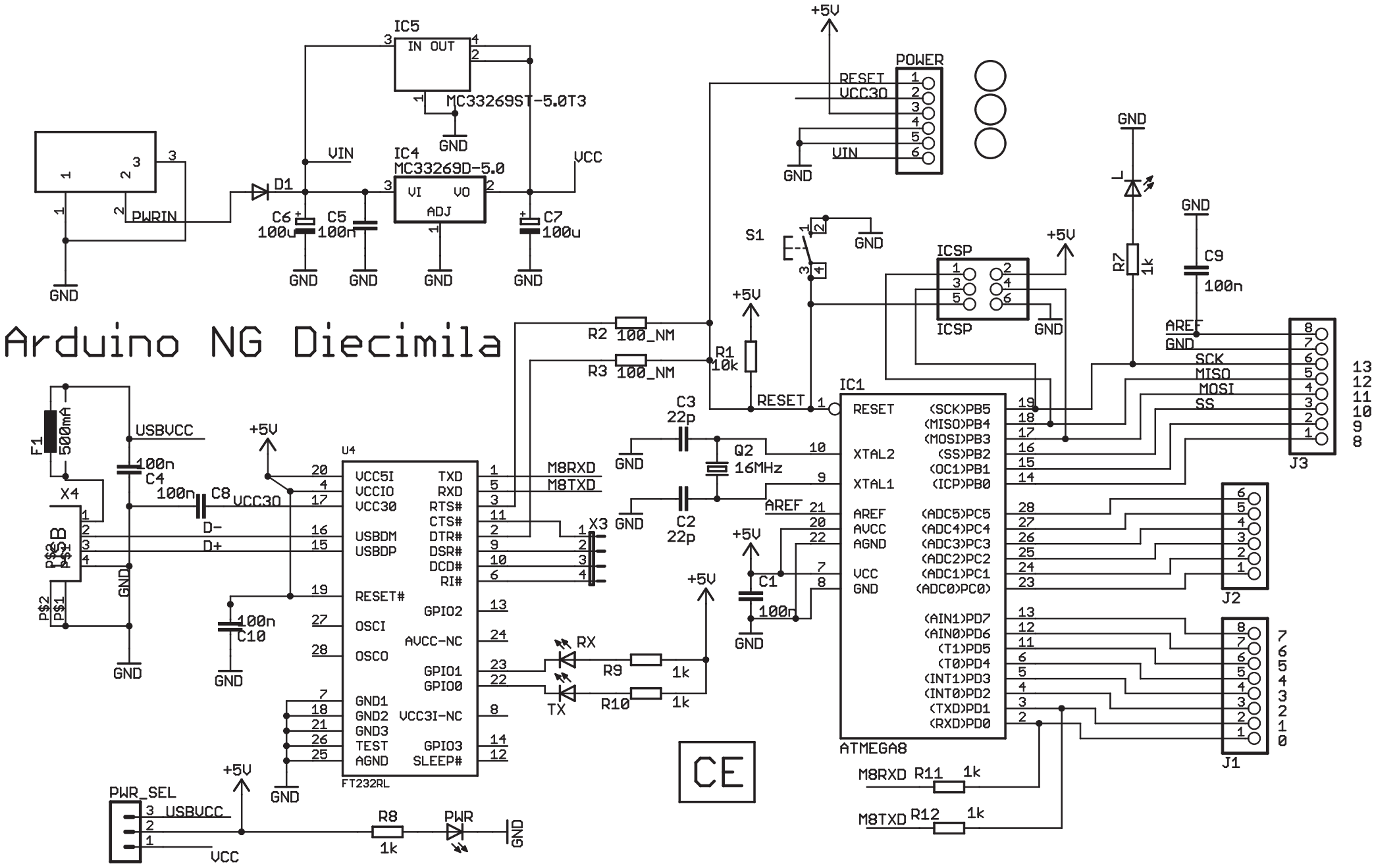
Finally, the instantiation of the task and assignment of its behavior need to be added to the main system start-up code:

```
BEACON BUILD
/BEACON
```

We now have a portable application that can sit on top of the LED API for just about any target processor supported by SwiftX (not just these two Arduino boards).

Arduino™ UNO Reference Design




Arduino NG Diecimila
IC5
IN OUT
MC33269ST-5.0T3
IC4
MC33269D-5.0
VI VO
ADJ
VIN
VCC
GND
PWRIN
D1
C6 100u
C5 100n
C7 100u
POWER
RESET
VCC30
VIN
+5V
S1
ICSP
R1 10k
R2 100_NM
R3 100_NM
RESET
IC1
ATMEGA8
RESET
(SCK)PB5
(MISO)PB4
(MOSI)PB3
(SS)PB2
(OC1)PB1
(ICP)PB0
XTAL2
XTAL1
AREF
AVCC
AGND
VCC
GND
(ADC5)PC5
(ADC4)PC4
(ADC3)PC3
(ADC2)PC2
(ADC1)PC1
(ADC0)PC0
(AIN1)PD7
(AIN0)PD6
(T1)PD5
(T0)PD4
(INT1)PD3
(INT0)PD2
(TXD)PD1
(RXD)PD0
C3 22p
C2 22p
Q2 16MHz
C1 100n
R7 1k
C9 100n
L
J3
SCK
MISO
MOSI
SS
AREF
J2
J1
U4
FT232RL
VCC5I
VCCIO
VCC30
USBDM
USBDP
RESET#
OSCI
OSCO
GND1
GND2
GND3
TEST
AGND
TXD
RXD
RTS#
CTS#
DTR#
DSR#
DCD#
RI#
GPIO2
AVCC-NC
GPIO1
GPIO0
VCC3I-NC
GPIO3
SLEEP#
M8RXD
M8TXD
X3
X4
USB
F1 500mA
USBVCC
C4 100n
C8 100n
C10 100n
D-
D+
VCC30
RX
TX
R9 1k
R10 1k
R11 1k
R12 1k
R8 1k
PWR
PWR_SEL
USBVCC
VCC
CE