Chapter 7 - Hardware Plus Software

Arduino Internals

by Dale Wheat

Available Peripherals

You've already used several of the Arduino's built-in peripherals in the example sketches. The serial port lets you communicate with the host PC as well as other serial devices. The digital ins and outs let you turn things like LEDs on and off, as well as tell if buttons have been clicked. The pulse-width-modulation (PWM) outputs are an excellent way to simulate an analog output using a digital interface, which is perfect for dimming lights and controlling motor speeds. Driving the PWM outputs are the lower-level timer/counter peripherals, which can be used, as their name implies, for both timing and counting purposes.

The Atmel AVR also has a couple of true analog peripherals, including the analog-to-digital converter (ADC) and an analog comparator (AC). These inputs take a varying voltage, generally ranging between ground (0V) and V_{CC} (5V), and convert it into ones and zeros.

Let's take a look at some of these available devices-within-the-device and see what happens when you flip them over to fully automatic.

Serial Port(s)

The Arduino's serial port gets a lot of use. It's the preferred mechanism for uploading new sketches from the development software to the chip itself. You can use it to debug programs and report readings. It works straight out of the box, using the supplied `Serial` library, and it uses only two device pins to do so.

The Arduino Uno has a single *hardware* serial port, USART, whereas the Arduino Mega 2560 has four of them. Both flavors of I/O Boards connect their first USART to a serial-USB interface. It's also possible, in a limited fashion, to use any pair of the other digital I/O pins as a serial port using the `SoftSerial` library. The advantages of using a real hardware USART certainly outweigh any software simulation, assuming you have enough USARTs to go around.

A Small Replacement for the Serial Library

In Chapter 6, you replaced the Arduino-supplied `Serial` library with a few simple functions. You saved a lot of program memory, but at what cost? No facility was provided to *receive* serial input. The transmission of data was quite wasteful of CPU bandwidth, as the processor waited, patiently, for every single character to shuffle slowly out the serial port, doing no useful work in the interim. No attempt was made to provide *handshaking* with the serial port on the other end of the line, and no contingency plan was in place to handle any errors that might occur.

With all of these shortcomings, the exercise was still worthwhile, because it did implement a very simple method for sending a limited amount of data via the serial port while saving quite a bit of precious program memory. Keep this technique in reserve for when you need to send a small amount of data but don't need all the bells and whistles provided by the `Serial` library.

Is the `Serial` library always a better alternative in nontrivial applications? Sometimes, but not always. There are still timing and reliability issues with this software, which should one day be sorted out. In the meantime, you can use the existing USART hardware, taking advantage of its talents and capabilities, which free up the CPU to do other tasks.

What the USART Does

The main job of each of the USARTs on board the AVR is to *serialize* and *deserialize* a stream of data bits. This allows a multibit piece of data, such as an eight-bit byte, to be transmitted or received using a single wire. Otherwise, eight data lines plus some sort of timing signal, or *strobe* line, would be needed to send or receive a single byte of information. That sort of arrangement is usually called a *parallel interface*.

The USART also handles, in hardware, the generation of the proper timing signals and the appropriate framing, error-checking, and synchronization bits that allow it to comply with asynchronous serial communication standards. Beyond these duties, the USART hardware in the AVR also provides a small amount of buffering, which permits a byte of data to be prepared for either transmission or reception while another one is actually being transmitted or received.

The very basic task of shifting bits in and out at the proper rate, with the right framing, *can* be accomplished with software, as illustrated by the abilities of the `SoftSerial` library. On the smaller AVR devices that lack any hardware USART, this is the only option, short of adding a dedicated USART peripheral to the circuit. Letting the hardware USART do its job frees the CPU to get on with other business.

Talking to the USART

Instead of dealing with the individual bit timings and all the other low-level details for serial communication, you did manage to use the built-in USART hardware to do all that work for you. Once the proper device configuration was performed (a total of four program statements), all you had to do in order to transmit a byte of information over the serial port was to write to a register, a la `UDR0 = c;`, and you were done. The USART considers it polite to wait for that byte to finish transmitting before transmitting another byte, and it even provides a status bit in one of its control registers to tell you when that moment occurs.

What you did in your first-generation USART code was sit there and watch the USART0 Data Register Empty (`UDRE0`) bit in the `UCSR0A` register until it changed. This is ~~a waste of time~~ an opportunity for excellence. In addition to the `UDRE0` status bit, the USART also provides an interrupt to let the CPU know when the `UDRE0` bit is set, (called the Data Register Empty interrupt), indicating that there is now room in the USART's output buffer mechanism for a new, transmittable byte.

Bear in mind that another, similar interrupt is available from the USART: the Transmit Complete interrupt. From the sound of its name, it would seem like this is the one you would want to use. There has been some confusion over this choice in the past. The Transmit Complete interrupt occurs when the serial transmission is *totally* finished: that is, after the last part of the last bit is sent out the serial port. The Data Register Empty interrupt occurs *before* the Transmit Complete interrupt, due to the internal buffer in the USART hardware. You should use the Data Register Empty interrupt to tell you when to reload the data register, if needed, because you can send out another byte before the previous byte has completely shifted out the serial port.

You *can* use the Transmit Complete interrupt in applications where some action needs to take place to commemorate the end of the serial transmission: for example, when special transmitter hardware needs to be powered down. Because you're just wanting to save some time at the moment, you use the interrupt that happens first and saves you the most time: the Data Register Empty interrupt.

The USART hardware can also generate an interrupt in response to the complete *reception* of a byte from the serial port. This interrupt is called the Receive Complete interrupt. See Listing 7-1 for an example sketch that toggles the LED every time a new character arrives via the serial port.

Listing 7-1: USART Receive Complete Interrupt Example Sketch

```
void usart_init(unsigned long rate) {
  UCSR0A = 0<<TXC0 | 0<<U2X0 | 0<<MPCM0;
  UCSR0B = 1<<RXCIE0 | 0<<TXCIE0 | 0<<UDRIE0 | 1<<RXEN0 | 0<<TXEN0 | 0<<UCSZ02 | 0<<TXB80;
  UCSR0C = 0<<UMSEL01 | 0<<UMSEL00 | 0<<UPM01 | 0<<UPM00 | 0<<USBS0 | 1<<UCSZ01 | 1<<UCSZ00 |
0<<UCPOL0;
  UBRR0 = (F_CPU / 16 / rate - 1);
}

void setup() {
  bitSet(DDRB, 5); // D13 (PB5) is connected to an LED
  usart_init(9600); // configure USART
}

void loop() {
  // nothing happens here
}

ISR(USART_RX_vect) {
  bitSet(PINB, 5); // toggle LED
  unsigned char c = UDR0; // read incoming byte to clear interrupt flag
}
```

There are a few important items to discuss in this sketch. First, the USART initialization code has been collected into its very own subroutine, called `usart_init()`. The desired communication rate (sometimes called the *baud rate*) is passed as its only parameter, a la the Arduino's `Serial.begin()` library function.

You can save a few more bytes if you know ahead of time what baud rate you want to use, instead of passing it as a parameter to `usart_init()`. You can use the `BAUD_RATE_DIVISOR` macro definition from the previous sketch in Chapter 5 ("Writing to Configuration Registers" section), instead. This is because the compiler sees the constant value indicated and

does the math itself, reducing the equation to a single number. Otherwise, the Arduino has to calculate the baud rate on the fly, which involves two division operations and results in the final sketch containing the `__udivmodsi4` library function, which is artfully compact but still takes up some space.

Notice the second line of the `usart_init()` function:

```
UCSR0B = 1<<RXCIE0 | 0<<TXCIE0 | 0<<UDRIE0 | 1<<RXEN0 | 0<<TXEN0 | 0<<UCSZ02 | 0<<TXB80;
```

The `RXCIE0` bit (USART0 Receive Complete Interrupt Enable) and the `RXEN0` bit (USART0 Receiver Enable) are the only bits that are set to `1` in this assignment statement. This could just as easily have been accomplished with two much shorter lines of code:

```
bitSet(UCSR0B, RXCIE0); // enable receive interrupt
bitSet(UCSR0B, RXEN0); // enable receiver
```

This would have resulted in a slightly larger sketch, however. Because the USART control registers aren't in the addressable range of the `SBI` and `CBI` machine-language instructions, the compiler can't use them to directly manipulate the bits in the `UCSR0B` register. It's forced to perform a read/modify/write operation on the `UCSR0B` register *twice*. This results in a binary sketch size of 618 bytes, compared with the original, more compact sketch that contained 608 bytes.

The `TXEN0` bit isn't set to `1` because you're not using the USART transmitter at all in this sketch. The other bits remain the same from the previous example sketches found in Chapter 5. The `setup()` function is fairly predictable, configuring the D13 pin (PB5 on the Arduino Uno) as an output so you can see the LED later, and then calling the `usart_init()` function to get the USART rolling.

Of special interest is the `loop()` function, which is suspiciously empty. All the fun stuff happens in the interrupt service routine (ISR), which is defined just after the `loop()` function. `ISR()` looks like a function but is really a macro defined in the `avr/interrupt.h` header file. The parameter passed to the ISR macro is the predefined signal name of the interrupt handler, which in this case is `USART_RX_vect`. Signals are a holdover from long, long ago; the term *signal* described what you now know as interrupts.

Note You don't get to name your interrupt handlers. You must use the signal names supplied by the `avr-libc` library. See Table 7-1 and Table 7-2 in the section "Pin-Change Interrupts." And yes, they all end in `_vect`, which is short for *vector*.

Table 7-1: Pin-Change Interrupts for ATmega328

Open table as spreadsheet

General-Purpose I/O Port	Pin Name	Pin-Change Interrupt Bank	Pin-Change Interrupt Name
Port B	PB0	Bank 0	PCINT0
	PB1		PCINT1
	PB2		PCINT2
	PB3		PCINT3
	PB4		PCINT4
	PB5		PCINT5
	PB6		PCINT6
	PB7		PCINT7
Port C	PC0	Bank 1	PCINT8
	PC1		PCINT9
	PC2		PCINT10
	PC3		PCINT11
	PC4		PCINT12
	PC5		PCINT13
	PC6		PCINT14
Port D	PD0	Bank 2	PCINT16
Note: PC7 (PCINT15) doesn't exist on the ATmega328.			

General-Purpose I/O Port	Pin Name	Pin-Change Interrupt Bank	Pin-Change Interrupt Name
	PD1		PCINT17
	PD2		PCINT18
	PD3		PCINT19
	PD4		PCINT20
	PD5		PCINT21
	PD6		PCINT22
	PD7		PCINT23
Note: PC7 (PCINT15) doesn't exist on the ATmega328.			

Table 7-2: Pin-Change Interrupts for the ATmega2560

Open table as spreadsheet

General-Purpose I/O Port	Pin Name	Pin-Change Interrupt Bank	Pin-Change Interrupt Name
Port B	PB0	Bank 0	PCINT0
	PB1		PCINT1
	PB2		PCINT2
	PB3		PCINT3
	PB4		PCINT4
	PB5		PCINT5
	PB6		PCINT6
	PB7		PCINT7
Port E	PE0	Bank 1	PCINT8
Port J	PJ0		PCINT9
	PJ1		PCINT10
	PJ2		PCINT11
	PJ3		PCINT12
	PJ4		PCINT13
	PJ5		PCINT14
	PJ6		PCINT15
Port K	PK0	Bank 2	PCINT16
	PK1		PCINT17
	PK2		PCINT18
	PK3		PCINT19
	PK4		PCINT20
	PK5		PCINT21
	PK6		PCINT22
	PK7		PCINT23

The interrupt handler works *mostly* just like any other function, except that it never takes parameters and it never returns a value. You cover a few other ISR-specific details shortly.

In the `ISR(USART_RX_vect)` handler, you see the shortcut LED toggle method:

```
bitSet(PINB, 5); // toggle the LED
```

So every time this interrupt handler executes, you should be able to see the LED change state.

Also in the interrupt handler is a dummy read of the USART0 data register, `UDR0`. This read is *required* to reset the "receive complete" interrupt flag. Normally an application shows some sort of interest in the incoming data stream, but this example sketch just wants to demonstrate how an interrupt handler is written, and the received value is discarded.

Compile and upload this sketch to your Arduino Uno. Verify that the LED toggles for every character that is sent from the Serial Monitor window. Remember that you must click the Send button or press the Enter key to actually send anything. Sending two characters at a time toggles the LED and then quickly toggles it back again, so you should only see a very short blink.

You can do something more intelligent in the interrupt service routine if you like. Try the code in Listing 7-2, replacing the previous interrupt handler.

Listing 7-2: Alternate USART Receive Complete interrupt handler

```
ISR(USART_RX_vect) {
  unsigned char c = UDR0; // read incoming byte to clear interrupt flag
  if(c =='1') {
    bitSet(PORTB, 5); // turn on LED
  } else if(c == '0') {
    bitClear(PORTB, 5); // turn off LED
  }
}
```

That's all there is to writing an interrupt handler. The hard part is to pick the right interrupt to use, enable it, and then find the correct signal name for it from the available documentation. See Table 7-7 and Table 7-8 in the section "Interrupt Reference." Note that the current `avr-libc` documentation as supplied by the Arduino software is chock-full of errors and omissions, especially in the signal-names section. This is mostly because the documentation was automatically generated from the source code comments, which suffer from repeated "improvements," usually of the copy-and-paste variety.

This sketch is specific to the Arduino Uno, not only because the LED bit is hardwired to PB5 (recall that the LED is attached to PB7 on the Arduino Mega 2560) but also because the ATmega2560 has *four* USARTs (USART0-3), and the handler names are different:

```
ISR(USART0_RX_vect)
ISR(USART1_RX_vect)
ISR(USART2_RX_vect)
ISR(USART3_RX_vect)
```

Caution The compiler doesn't check for proper handler names. At this point, it can only look to see whether the handler name ends in `_vect`. Using USART_RX_vect (for the ATmega328) instead of USART0_RX_vect (for the ATmega2560) will produce no compile-time errors, only runtime errors.

ISR() Options

The `avr-libc` library provides several options to pass along to the interrupt service vectors, depending on how you want them to behave.

The first parameter to the ISR function is the signal name that is to be handled. It bears repeating that you do *not* get to name your own signals. You must use one of the predefined signal names from the `avr-libc` library.

As discussed in Chapter 3, interrupts must be *enabled* in two places to ever fire. The first is specific to the hardware that is generating the interrupt, such as the USART peripheral. The second is the *global interrupt enable bit* (`I`) contained in the processor's status register (`SREG`). If the `I` bit isn't set, then no interrupts are recognized, acknowledged, or acted upon by the processor. The Arduino software provides two simple macros for enabling and disabling the global interrupt enable bit:

```
interrupts();
noInterrupts();
```

These two macro definitions resolve into the single machine-language instructions Set I Bit in SREG (`SEI`) and Clear I Bit in SREG (`CLI`), respectively.

When enabled, and the right conditions prevail, an interrupt can be generated. This launches a series of actions on the part of the processor. First, the global interrupt enable bit `I` is cleared. This prevents any further interrupts from being handled until either

- The interrupt handler routine returns or

- The interrupt handler routines explicitly reenables the `I` bit

The address of the current program instruction is saved on the stack. The processor then *vectors* (jumps) to the specific location in the interrupt vector table that is reserved for that interrupt. The interrupt vector table usually contains jump (`JMP`) instructions that point the processor to the appropriate interrupt handler routine.

The interrupt vector executes, and when it's complete, issues the special Return from Interrupt (`RETI`) instruction. This instruction recalls the program counter's previous value by popping it from the stack and reenables the global interrupt enable bit. Program execution resumes from the point where it was interrupted.

Although the processor automatically saves the program counter's contents, this isn't true of either the processor's `SREG` or the contents of any of the general-purpose registers, R1-R31. All but the most trivial of interrupt service routines need to save the processor's state before performing any calculations; otherwise the interrupted program running in the foreground is in for a nasty surprise.

The compiler knows how to properly save the processor's state, and the code to do so is automatically included both before and after the code written by the application programmer in the interrupt service handler.

Nested Interrupts
To allow the use of nested interrupts—that is, an interrupt handler that allows other events to interrupt it—you can add the `ISR_NOBLOCK` modifier to the ISR handler:

```
ISR(USART_RX_vect, ISR_NOBLOCK) ...
```

Using this modifier reenables global interrupts earlier in the execution of the handler than specifically executing the `interrupts();` macro call would.

Empty Interrupts
Some interrupt flags are cleared when the interrupt service routine executes, unlike the USART Receive Complete interrupt, which required the data register to be read. In these and similar cases, no other action may be required to be performed by the interrupt handler, and the interrupted program can resume as quickly as possible. This is often the case when the processor is put to sleep and one of the peripherals needs to wake it up. To do this and generate the minimum amount of code, you can use the `EMPTY_INTERRUPT()` macro instead of the `ISR()` macro:

```
EMPTY_INTERRUPT(TIMER0_OVF_vect);
```

This macro then creates a handler that does nothing except return to the interrupted program using the `RETI` instruction. No function body is required when using this alternative form.

Bare-Naked Interrupts
When you need the most control over the code going into your interrupt handler routines, you can use the `ISR_NAKED` modifier, which generates a function body lacking both the standard prolog and epilog for saving and restoring the processor state. You even have to explicitly include the `RETI` instruction within such a function, because the compiler isn't going to put *anything* in there for you.

If your handler only needs to update a flag or perform some other miniscule task that doesn't affect either the `SREG` or the general-purpose registers, you can use this option:

```
ISR(TIMER0_OVF_vect, ISR_NAKED) {
  bitSet(PINB, 5);
  reti(); // return from interrupt
}
```

The `reti()` function is another macro that inserts the correct machine-language code for the Return from Interrupt instruction. It's defined in the `avr/interrupt.h` header file. You *must* include this macro as the last statement in your interrupt handler if you use the `ISR_NAKED` modifier.

Undefined Interrupt Behavior
If an interrupt occurs for which no handler has been written, the default action taken by the compiler is to tell the processor to jump back to the beginning of its program, effectively restarting the chip. It does this by filling the interrupt vector table with bad interrupt vectors, unless the application program writer specifies a proper handler for a particular interrupt.

You can change this default behavior by defining your own default function, using the `BADISR_vect` signal name:

```
ISR(BADISR_vect) ...
```

Undefined interrupts generally indicate software bugs. Restarting the sketch is a bit drastic, however. You can more elegantly handle it using the `BADISR_vect` option.

General-Purpose Digital Inputs and Outputs

You've been playing with the digital input and output (I/O) pins in almost every example sketch. You've learned how to configure them as either inputs or outputs, either using the `pinMode()` function or specifically setting the direction bits in the data direction registers (`DDRx`). You've mostly been working with individual I/O lines using either the Arduino-supplied `digitalWrite()` function or the `bitSet()` and `bitClear()` macros. It's also possible to read the values present on the pins by either using the `digitalRead()` function or directly interrogating the `PINx` registers.

Another fun trick is to enable and disable the built-in pullup resistors on each of the input pins, using the otherwise-unused data register, which is normally used when the pins are set up to be outputs. Along the same line, the output state of any pin can be toggled by writing to the Input Pins address; this is very useful when you don't want to disturb any of the neighboring I/O pins.

You can also read and write up to eight bits at a time by talking to the I/O ports' data register and Input Pins address. This can be much faster and more compact than updating or checking each pin individually.

Pin-Change Interrupts
Besides all this general functionality, all of the I/O pins on the Arduino Uno, and up to 24 of the pins on the Arduino Mega 2560, can be configured as pin change interrupts. This permits any of these pins to be set up to notify the CPU if anything changes on their inputs, without forcing the CPU to constantly check in to see what's happening.

Pin-change interrupts are grouped into three banks of up to eight individual lines apiece. Each bank has an interrupt associated with it. These banks generally correspond with the general-purpose I/O ports. See Table 7-1 and Table 7-2.

To respond to a pin-change interrupt, you must do three things:

1. Enable the appropriate pin-change interrupt bank using the PCICR register.
2. Enable the specific pins to be used using the PCMSK0–2 registers.
3. Write an interrupt handler routine.

Single Pin Interrupt Example
Let's try using a single pin-change interrupt to see how it works. You take advantage of the fact that the pin-change interrupts don't care if the pin that's involved is an input or an output. This way, you can invoke an interrupt through software by changing the state of an output pin, without having to attach any new hardware to the Arduino.

Let's go back to your faithful friend, Blink. Select File ➤ Examples ➤ 1.Basics ➤ Blink from the menu. Add the following lines to the end of the `setup()` function:

```
Serial.begin(9600);
bitSet(PCICR, PCIE0); // enable pin change interrupt bank 0
bitSet(PCMSK0, PCINT5); // enable pin change interrupt on PCINT5/D13
```

You leave the `loop()` function alone. It continues to turn on the LED, wait a second, turn off the LED, wait another second, and then repeat. That's what you like about Blink.

Add the following interrupt handler for pin-change interrupt bank 0:

```
ISR(PCINT0_vect) {
  if(digitalRead(13) == HIGH) {
    Serial.println("LED is on");
  } else {
    Serial.println("LED is off");
  }
}
```

Compile and upload this sketch. Open the Serial Monitor window, and observe what happens.

You should get a message about once a second, alternating between "LED is on" and "LED is off." The interrupt handler gets invoked every time there is a *change* in the state of the PCINT5 pin, which you also know as D13 or PB5. You don't get to specify what kind of change. So when the `loop()` function calls the `digitalWrite()` function to either turn on or turn off the LED, as long as the new value is different from the present value, a pin-change interrupt is detected, and the interrupt service routine is called.

In the interrupt service routine, the `digitalRead()` function is called to determine the present state of the pin. If it's `HIGH`, the LED is on, and a suitable status message is sent. Otherwise, the LED is most likely off, and that status is likewise reported.

This example is quite simplified because only one pin holds your interest. You knew that only one pin had been authorized to generate an interrupt, so it wasn't hard to figure out what happened.

You can use any of the I/O pins on the ATmega328 as pin-change triggers. Each pin-change interrupt bank has its own interrupt handler. It's up to the individual interrupt handler to determine which pin changed, triggering the interrupt request. This can be discovered by reading the status of the port pins, using the Input Pins address (`PINx`) for the applicable port.

The ATmega2560 also has three banks of pin-change interrupts. The same registers are used to enable the individual pins and banks, although banks 1 and 2 are mapped to different ports.

Timers and Counters

The ATmega328 has three unique timer/counter peripherals, cleverly named Timer/Counter 0, Timer/Counter 1, and Timer/Counter 2. Each timer/counter peripheral is slightly different from the others. Let's look at Timer/Counter 0 first and learn what makes it so special. Once you understand what makes this timer tick, then you can go on to the other two timer/counters and concentrate on how they differ from Timer/Counter 0.

Timer/Counter 0
At the core of Timer/Counter 0 is an eight-bit counter. This means it can count from 0 to 255, at which point it starts over again at 0. It can also count from 0 to 255, then back down from 255 to 0, over and over again. This is the basic thing that it does: its reason for existence.

Several variations on this basic theme can be made with Timer/Counter 0. You can specify a different maximum number instead of 255. It can be either a counter or a timer. In its role as a timer, it can be clocked by the CPU oscillator or through a prescaler with predefined divisors. These divisors can be 1, 8, 64, 256, or 1,024. As a counter, it can be driven from an external input pin.

Timer/Counter 0 has two pulse-width-modulation (PWM) channels associated with it. (PWM is explained in more detail in the section "Pulse-Width-Modulation (PWM) Outputs.")

Timer/Counter 1
The heart of Timer/Counter 1 is a 16-bit counter. This allows it to count from 0 to as much as 65,535. Like Timer/Counter 0, it can either always be counting up or it can count up, then down, then up, then down.

Timer/Counter 1 also has an input capture unit, which lets it put a timestamp on an incoming signal. Two PWM channels are available for use with Timer/Counter 1 on the Arduino Uno, and three PWM channels on the Arduino Mega 2560.

Timer/Counter 2
Timer/Counter 2 is remarkably similar to Timer/Counter 0. It's an eight-bit counter with two associated PWM channels, having both timer and counter facilities as well as the ability to generate several interrupts.

What makes Timer/Counter 2 special is its ability to be driven asynchronously from the CPU clock, via a low-frequency 32KHz watch crystal attached to the TOSC1 and TOSC2 pins. On the ATmega328, these pins serve double duty with the primary XTAL1 and XTAL2 pins, normally connected to the 16MHz quart crystal or ceramic oscillator. This prevents most Arduino Uno-compatible circuits from implementing this feature.

On the ATmega2560, however, the TOSC1 and TOSC2 pins are only shared with the PG4 and PG3 general-purpose I/O lines. The ATmega2560 has dedicated XTAL1 and XTAL2 pins, so no conflict is present. Timer/Counter 2 can be clocked using a low-power (as well as relatively low-frequency) watch crystal. This allows the implementation of a real-time clock that can continue to run when the processor is put to sleep.

ATmega2560
The ATmega2560 has all the same timer/counter peripherals as the ATmega328, and then some.

Timer/Counter 1 on the ATmega2560 is also a 16-bit counter, just like on the ATmega328. It adds an additional PWM channel,

for a total of three PWM channels on Timer/Counter 1 alone.

If that weren't enough, the ATmega2560 sports three more 16-bit timer/counter peripherals, which are all clones of Timer/Counter 1. They're aptly named Timer/Counter 3, Timer/Counter 4, and Timer/Counter 5.

Timer Interrupts

You can perform some simple experiments with the timer/counter interrupts to get an idea of how they operate. The Arduino software has already configured all the timer/counter peripherals for use with the `analogWrite()` function, as well as time-keeping duties for Timer/Counter 0. You use Timer/Counter 1 for this experiment, fully aware that you then lose the use of analog (PWM) outputs 9 and 10 on the Arduino Uno. See Listing 7-3.

Listing 7-3: A Simple Timer Interrupt

```
void setup() {
  bitSet(DDRB, 5); // PB5/D13 has an LED attached
  TCCR1A = 0;
  TCCR1B = 1<<CS12;
  bitSet(TIMSK1, TOIE1); // enable overflow interrupt
}

void loop() {
  // nothing happens here
}

ISR(TIMER1_OVF_vect) {
  bitSet(PINB, 5); // toggle LED
}
```

Compile and upload this sketch. Behold! You have yet another blinking LED. Let's see exactly what's going on here.

The `setup()` function starts off with the familiar `pinMode()` function replacement, `bitSet(DDRB, 5)`, just to save a little program space. Then you see some manipulation of the timer control registers, which is where all the magic happens.

The second line of the `setup()` function clears all the bits in the `TCCR1A` register. This register controls the assignment of the PWM output pins and also contains two of the four bits used to select the timer mode. Because the Arduino software had already programmed this register to provide for PWM outputs that you aren't using, you clear out all the bits at once by writing a zero to the register.

The next timer configuration statement writes a single `1` to the `CS12` bit location of the `TCCR1B` register, which has the side effect of also writing zeros to all the other bit locations. The other two mode-select bits were in this register, but you wanted to set them to zeros, anyway. The `CS12` bit is one of three bits (`CS10`, `CS11`, and `CS12`) that select the prescaler value for the system clock to drive the counter in a timer mode. There are eight possible combinations (see Table 7-3), and you want the one that provides a divide-by-256 prescaler of the system oscillator. Setting only the `CS12` bit makes this selection for you.

Table 7-3: Timer Clock Select Bits for Timer/Counter 1

Open table as spreadsheet

CS12	CS11	CS10	Description
0	0	0	No clock
0	0	1	Divide by 1
0	1	0	Divide by 8
0	1	1	Divide by 64
1	0	0	Divide by 256
1	0	1	Divide by 1,024
1	1	0	Use the falling edge of external input T1
1	1	1	Use the rising edge of external input T1

Timer/Counter 0 has a similar set of predefined clock prescalers that can be selected. The only differences are that the bit names are `CS02`, `CS01`, and `CS00`, and that the last two options select input pin T0 instead of T1. Timer/Counter 2 is different. It doesn't support an external input in the same way that Timer/Counter 0–1 can. See Table 7-5, later in the chapter.

Because the system clock on a standard Arduino is 16MHz, the clock signal provided to the timer circuit is now 16,000,000 ÷

256 = 62,500Hz, or 62.5KHz. This value was selected with some care, and it nearly approximates the maximum value that the 16-bit counter in Timer/Counter 1 can hold, which is 65,535. In other words, when driven at this clock speed, the counter overflows in just over 1 second. This generates an overflow interrupt, the interrupt service routine executes, and the LED is toggled.

The final timer configuration statement in the `setup()` function specifically enables the overflow interrupt for this timer, Timer Overflow Interrupt Enable 1 (`TOIE1`). Because the Arduino software has already enabled the global interrupt bit, things should start happening in just about one second, assuming your interrupt service routine is in place.

The interrupt handler couldn't be simpler:

```
ISR(TIMER1_OVF_vect) {
  bitSet(PINB, 5); // toggle LED
}
```

This example illustrates the normal mode of the timer, which is only one of many possible modes that are available using the timer/counter peripherals. For example, if you wanted an interrupt to occur *exactly* every second, you could use one of the two Clear Timer on Compare Match (CTC) modes. In this mode, you configure a prescaler and select the counter/timer mode in a manner similar to the previous example, but then you program the desired match value used to trigger a reset of the counter, instead of letting it roll over on its own. See Listing 7-4.

Listing 7-4: Generating a Timer Interrupt Every Second

```
void setup() {
  bitSet(DDRB, 5); // LED on PB5/D13
  TCCR1A = 0;
  TCCR1B = 1<<WGM13 | 1<<WGM12 | 1<<CS12 | 1<<CS10;
  ICR1 = 15625;
  bitSet(TIMSK1, ICIE1); // enable capture event interrupt
}

void loop() {
  // nothing happens here
}

ISR(TIMER1_CAPT_vect) {
  bitSet(PINB, 5); // toggle LED
}
```

This mode uses the Input Capture Register (`ICR1`) to hold the compare value. The value is calculated by dividing the system clock by the requested prescaler, which determines the number of clock cycles to count before starting over, which fires the input capture event interrupt. In this case, the largest prescaler, 1,024, is used to divide the system clock, resulting in a clock signal of 15.625KHz driving the timer.

Pulse-Width-Modulation (PWM) Outputs

Each of the timer/counter peripherals has at least two PWM channels associated with it. The Arduino software automatically configures all of the available timers for hardware PWM duty at the beginning of every sketch.

Using the Arduino-supplied `analogWrite()` function, the hardware PWM channels can be easily programmed with any value between 0 (completely off) and 255 (completely on). Anything in between produces a variable duty-cycle pulse stream at approximately 490Hz.

The Arduino software treats all the PWM channels the same, limiting them to eight-bit resolution and hardwiring them to a relatively slow frequency. It's not hard to reprogram any of the available PWM channels and reconfigure them to your liking. The only trick is choosing among the many operational modes and setting the parameters accordingly.

Listing 7-5 is a short experiment you can try, which illustrates the simplicity of the Arduino's `analogWrite()` function and the use of PWM.

Listing 7-5: Fading the LED Using PWM

```
void setup() {
```

```
}

void loop() {

  static byte pwm = 0;

  analogWrite(11, pwm); // set the PWM duty cycle
  delay(10); // a very short delay
  pwm++; // increase the PWM duty cycle
}
```

Note that you don't have to explicitly set the direction bit for the output pin when using the Arduino's `analogWrite()` function. It takes care of this for you. If you don't use the `analogWrite()` function, and you set up the PWM outputs yourself, you need to remember to configure the proper pin as an output, either using the `pinMode()` function or setting the bits in `DDRx` appropriately.

Compile and upload this sketch to either an Arduino Uno or an Arduino Mega 2560. Nothing happens. To make something happen, all you have to is connect D11 to D13 using a short length of wire or a small-value resistor (see "Digital Signal Probe" in the Chapter 6).

On the Arduino Mega 2560, you can use D13 instead of D11 and the sketch works without the help of the extra wire. This is because the Arduino Mega 2560 has 14 PWM outputs, which happen to include D13—the one that is already connected to the LED.

Using the Arduino Uno, however, you have only six PWM outputs, and D13 is *not* one of them. That's OK—you can still connect one of the proper PWM outputs (D11 is the closest, so that's what you use) to D13 and light up the LED, as long as D13 itself isn't programmed to be an output.

Without the wire, using `analogWrite()` on a non-PWM pin produces `1` for values 128 and above and `0` for values less than 128.

PWM Tricks

You can eliminate the jumper wire in the previous exercise and replace it with a bit of software trickery, based on what you previously learned about using pin-change interrupts. See Listing 7-6.

Listing 7-6: Transferring the PWM Output to a Different Pin Using Pin-Change Interrupts

```
void setup() {
  bitSet(DDRB, 5); // LED pin
  bitSet(PCICR, PCIE0); // enable pin change interrupts on bank 0
  bitSet(PCMSK0, PCINT3); // enable PCINT3 (PB3) pin change interrupt
}

byte pwm = 0;

void loop() {
  analogWrite(11, pwm); // set the PWM duty cycle
  delay(10); // a very short delay
  pwm++; // increase the PWM duty cycle
}

ISR(PCINT0_vect) {
  if(bitRead(PINB, 3)) {
    bitSet(PORTB, 5); // LED on
  } else {
    bitClear(PORTB, 5); // LED off
  }
}
```

Nothing changes in the `loop()` function. The value of the `pwm` variable is used to write an analog value to the PWM output pin, using the `analogWrite()` function. Then the `pwm` variable is incremented. When it gets to its maximum value (255 for a `byte`, which is the same thing as an `unsigned char`), it rolls over to its minimum value, which is 0.

The `setup()` function now contains the initialization code to enable PCINT3, which corresponds with the digital pin 11 used

by the `analogWrite()` function. Because PCINT3 is contained in the first bank of pin-change interrupts, you write an interrupt handler for the `PCINT0_vect` signal.

The interrupt handler checks the state of the port pin, using the `bitRead()` macro. Notice that no explicit comparison is done in this conditional statement. The `bitRead()` macro returns either `1` or `0`, depending on the state of the bit being examined. In the C programming language, conditional statements are evaluated as either true (non-zero) or false (zero). When the output pin is a logical `1`, the `bitRead()` macro returns a value of true, so the LED is turned on. When it returns a false value—that is, zero, indicating the pin is in a logical low state—the LED is turned off. The conditional statement could be more explicitly written as

```
if(bitRead(PINB, 3) == 1) ...
```

This is yet another example of the possible compactness possible in the C language. Those guys were ~~lazy~~ efficient. Compactness and clarity aren't always the same thing, however. It's certainly possible to write such dense code that you, yourself, can't comprehend it. This is even more prevalent in other programming languages, such as Forth, which is considered by many to be a write-only language. Good commenting discipline helps!

Remove the jumper wire from your Arduino Uno, compile and upload the sketch, and observe what happens. If all goes well, the LED should be ramping up in brightness over the period of approximately 2.56 seconds and then starting over again.

More PWM Trickery

The PWM hardware can generate its own interrupts. In the previous example sketch, the PWM signal was present on two pins at once. This is a neat trick and can come in handy when you need to duplicate a relatively slow signal on multiple pins, for example, using the hardware USART to transmit on several different lines at once.

Let's use the PWM interrupt for overflow to increment the PWM duty cycle for you. You continue to use one of the Arduino Uno's PWM-capable pins, D11. You saw from the previous sketch that this pin is also referred to as PB3 (Port B, bit 3). To get even closer to the bare metal, you need to know the timer/counter with which this PWM output is associated. Table 7-4 lists the timer/counters, their PWM outputs, and both the AVR names and Arduino names for the Arduino Uno.

Table 7-4: Timer/Counter PWM Pins on the Arduino Uno

Open table as spreadsheet

Timer/Counter	PWM Output	AVR Name	Arduino Pin
Timer/Counter 0	OC0A	PD6	6
	OC0B	PD5	5
Timer/Counter 1	OC1A	PB1	9
	OC1B	PB2	10
Timer/Counter 2	OC2A	PB3	11
	OC2B	PD3	3

Working backward through Table 7-4, you see that Arduino PWM pin 11 is connected to Timer/Counter 2's `OC2A` output. Each of the timer/counters on the ATmega328 has two PWM channels associated with it, referred to as A and B. When configured for PWM usage, the internal counter is continuously compared with the values stored in the two Output Compare Registers, `OCRxA` and `OCRxB`. Several options exist for what to do when the values line up properly.

For your second PWM experiment, you configure Timer/Counter 2 for normal counter mode and enable both the Compare Match A interrupt and Overflow interrupt. When the counter overflows, the count starts over at zero. At this point, you want to turn on the LED. When the compare match event takes place, you want to turn off the LED. To do so, you define two interrupt service routines. See Listing 7-7.

Listing 7-7: Reassigning PWM Outputs to Any Output Pins, Using Interrupts

```
void setup() {
  bitSet(DDRB, 5); // LED pin
  TCCR2A = 0; // normal mode
  TCCR2B = 5; // super slow CK/128
  TIMSK2 = 1<<OCIE2A | 1<<TOIE2; // enable match and overflow interrupts
}

byte pwm = 0;
```

```
void loop() {
  analogWrite(11, pwm); // set PWM duty cycle
  delay(10); // a short delay
  pwm++; // increase PWM duty cycle
}

ISR(TIMER2_OVF_vect) {
  if(pwm) bitSet(PORTB, 5); // LED on, maybe
}

ISR(TIMER2_COMPA_vect) {
  if(pwm < 255) bitClear(PORTB, 5); // LED off, maybe
}
```

The `setup()` function looks pretty familiar, configuring the LED output pin and the timer/counter peripheral. Timer/Counter 2 has a slightly different set of available clock prescalers; see Table 7-5. You also enable two distinct interrupts associated with Timer/Counter 2, the Match Compare A and Overflow interrupts, by setting the appropriate bits (`OCIE2A` and `TOIE2`) in the Timer/Counter 2 Interrupt Mask Register, `TIMSK2`.

Table 7-5: Timer Clock Select bits for Timer/Counter 2

Open table as spreadsheet

CS22	CS21	CS20	Description
0	0	0	No clock
0	0	1	Divide by 1
0	1	0	Divide by 8
0	1	1	Divide by 32
1	0	0	Divide by 64
1	0	1	Divide by 128
1	1	0	Divide by 256
1	1	1	Divide by 1,024

The `loop()` function slowly adjusts the PWM duty cycle upward by continuously writing to the PWM hardware using the `analogWrite()` function. You could save 538 bytes of program memory by not using the `analogWrite()` function and writing directly to the compare register, `OCR2A`.

There is a bit of finesse going on in the interrupt service routines. You could have simply turned on the LED in the overflow interrupt (where the counter has just restarted from zero) and turned it off again at the compare match interrupt. This mostly works. It fails at the endpoints, 0 and 255. This is because the simplistic approach to PWM (turn on at zero, turn off at compare match) fails to consider that you may not want to turn on the output (for example, when the output is zero) or that you may not ever want to turn off the output (for example, when the output is full-scale).

In the overflow interrupt handler, the zero output scenario is tested by the shortened conditional `if(pwm)`, which returns `true` as long as the value of `pwm` isn't zero; otherwise it returns `false` and the rest of the statement isn't executed. The compare match interrupt, on the other hand, only clears the output if the PWM value is less than full scale (255).

Let's try one more variation of the PWM reassignment experiment. This time, you move all the functionality that is presently in the `loop()` function—that is, the ramping up of the PWM output value—into the overflow interrupt handler. Why? Because the overflow interrupt handler is being invoked on a regular basis, every time the counter overflows from 255 back to 0. This happens on a fixed, periodic basis.

Either delete or comment-out all the code in the `loop()` function. Now update the overflow handler as follows:

```
ISR(TIMER2_OVF_vect) {
  if(pwm) bitSet(PORTB, 5); // LED on, maybe
  OCR2A = pwm;
  pwm++;
}
```

Compile and upload the sketch. You should see the same old thing, just a little faster. Why faster? Let's calculate the update

frequency.

The system clock remains at 16MHz. The prescaler you selected for Timer/Counter 2 was divided by 128 by writing a 5 to the control register, `TCCR2B`. This means the counter is being clocked at 16,000,000Hz ÷ 128 = 125KHz. After every 256 clocks, the counter overflows. This occurs at roughly 488Hz. This is when the overflow interrupt occurs. In the interrupt handler, the PWM value (`pwm`) is incremented by one, which correspondingly adjusts the PWM duty cycle when it's written to the Output Compare Register A for Timer/Counter 2, `OCR2A`. Because this value, too, overflows when it gets to 255 (or after every 256 interrupts, if you want to look at it like that), this results in an apparent blinking frequency of ~488Hz ÷ 256 = ~1.9Hz, or about twice a second. This can be adjusted in large steps by changing the prescaler value of the timer.

What's interesting about this approach is that the `loop()` function is now doing absolutely nothing. All the action is happening behind the scenes in the interrupt handlers.

Analog Inputs

Both the Arduino Uno and the Arduino Mega 2560 have a single ADC peripheral on board. Each ADC unit has an input multiplexer with several inputs. The ATmega328 has 8 analog inputs, and the ATmega2560 has 16 analog inputs. Only six of the eight available analog inputs are available on the plastic DIP version of the device. The surface-mount version has all eight inputs available, but only six of them are connected to anything on the Arduino Uno. Several Arduino clones provide extra headers for these two extra ADC inputs.

The inputs to the ADC multiplexer share pins with the general-purpose digital inputs and outputs. This means if you don't want or need to use the analog inputs in your application, you're free to use those pins as regular digital I/O.

If, on the other hand, you're going to use any of the pins in analog mode, you should consider disabling any unused digital inputs by setting the individual bits (`ADC0D–ADC5D`) in the Digital Input Disable Register 0 (`DIDR0`). This reduces the amount of power that would have been used by these unused digital inputs.

The ATmega2560 has an additional Digital Input Disable Register to accommodate its larger array of analog inputs, but curiously it's called `DIDR2`. The missing `DIDR1` is found on the earlier ATmega16, which used a completely different bit-mapping in the register, so the name was most likely changed to prevent confusion.

The ADC peripheral measures analog voltages by a method known as *successive approximation*. You may already be familiar with this process, but as the children's game Guess the Number. After each guess at the secret number, the guesser is informed if they're too high or too low until the right number is eventually guessed. The AVR's ADC peripheral methodically guesses at each of the ten bits in the resulting conversion by creating its own internal, adjustable voltage and comparing it to the sampled voltage from the analog input pin. This process is repeated ten times and eventually produces the analog conversion result, which is a number in the range of 0–1,023.

The ADC peripheral runs on its own clock, which in most cases is much slower than the processor clock. The optimum clock frequency ranges between 50KHz and 200KHz. A prescaler is provided to divide the processor clock to an appropriate frequency.

When the conversion process has begun, it takes 13 of these scaled-down ADC clocks to complete the conversion. The exception is the very first conversion performed after the ADC peripheral is enabled, which takes 25 ADC clock cycles to complete.

When the conversion is complete, the result is stored in a register and the peripheral's status flags are updated. The ADC can generate an interrupt once a conversion finishes.

If the ADC function isn't required in your application, consider disabling the peripheral by resetting the ADC Enable (`ADEN`) bit in the `ADCSRA` control register. Although this bit is off by default after power-on or a chip reset, the Arduino software turns it on at the beginning of every sketch.

Arduino Uno as Thermometer

The ATmega328 on the Arduino Uno also has an additional, internal ADC channel that is connected to a temperature sensor in the chip. This feature isn't implemented on the ATmega2560 chip.

You need to tweak this sketch to get an accurate reading on your Arduino Uno. Each temperature sensor is relatively linear in response, but the overall accuracy is rated at ±10°C. See Listing 7-8.

Listing 7-8: Arduino Uno as Thermometer

```
void setup() {
  Serial.begin(9600);
  Serial.println("Arduino Uno as Thermometer");
```

```
  ADMUX = 1<<REFS1 | 1<<REFS0 | 1<<MUX3; // 1.1V reference, ADC channel "8"
  ADCSRA = 1<<ADEN | 1<<ADSC | 0x07; // enable ADC, start conversion, 125 KHz clock
}

#define OFFSET 343

  void loop() {

  Serial.print(ADC - OFFSET);
  Serial.println("C");
  bitSet(ADCSRA, ADSC); // start next conversion
  delay(250); // wait
}
```

Compile and upload the sketch. Open the Serial Monitor, and see what it says. The first few readings may well be way off, but soon it will settle down and start reporting the temperature of the internal sensor.

The `setup()` function first initializes the serial port and then configures the ADC to measure the special ADC channel assigned to the internal temperature sensor. It specifies that the 1.1V internal voltage reference is to be used instead of the A_{VCC} voltage, which is normally 5.0V. This gives a better match to the output of the temperature sensor, resulting in a more accurate conversion.

The `loop()` function prints the conversion result, adjusted by the `OFFSET` value, followed by the unit of measure (`C`). The code continues on to start a new conversion by setting the Start Conversion bit (`ADSC`) in the `ADCSRA` control register. A short delay is performed, and the `loop()` function repeats.

You need to adjust the `OFFSET` value to calibrate your Arduino Uno. Bear in mind that the ATmega328 chip will be as much as 3°C warmer than the ambient air.

Arduino as Voltmeter

In Chapter 6 you built a digital signal probe, using just an Arduino and a short jumper wire. This tool is very useful for determining if a digital signal is high or low. It's not especially helpful if the signal is somewhere in between. For measuring voltage levels, you need a *voltmeter*.

You can build a limited-range voltmeter using the same Arduino and jumper wire and a different sketch. It would be nice if you could attach a variable resistor called a *potentiometer* to the circuit or use something like the Maker Shield (see Figure 1-6 in Chapter 1), which already has a potentiometer installed. If not, you can still explore some of the varying voltages on the Arduino itself.

> **Caution** If you're powering your Arduino via the USB cable, there are no dangerous voltage levels to worry about. If, on the other hand, you're powering your Arduino via the external power connector, take care *not* to connect the Vin connector directly to the pins of the ATmega chip on your Arduino.

Listing 7-9 shows a simple example sketch that repeatedly reads analog pin A0 and then prints out the average value over the serial port.

Listing 7-9: Simple Analog Voltmeter

```
void setup() {
  Serial.begin(9600);
  Serial.println("Arduino Voltmeter");
}

#define SAMPLES 2500

void loop() {

  unsigned long voltage = 0;
  unsigned int i;

  for(i = 0; i < SAMPLES; i++) {
    voltage += analogRead(0); // accumulate samples
  }
  Serial.print((((voltage * 5.0) / 1024) / SAMPLES), 4);
  Serial.println("V");
```

```
}
```

Compile and upload this sketch. Then open the Serial Monitor window to see what's going on. You should see the program banner and then a series of voltage measurements.

Connect A0 to the 5V pin on the power expansion header. The readings may vary due to the acceptable tolerances on your Arduino's power supply, so don't be alarmed if you see 4.9785V on the 5V line.

Now try the 3.3V connector and see whether you're getting something in the neighborhood. Finally, try connecting your voltage probe to one of the GND connections and make sure the sketch is reporting something quite close to 0.0000V.

The `setup()` function initializes the serial port and prints a short announcement as to the application's primary intent. The `loop()` function takes a large number of samples, specified by the constant defined as `SAMPLES`, adding all the samples together in a big pile. After all the samples have been gathered together, the actual voltage is calculated based on the number of samples and adjusted by the analog reference voltage (AREF) and the resolution of the ADC. The resulting voltage is reported to four decimal places on the serial port, and the `loop()` function repeats.

The ADC produces a reading that has a resolution of ten bits. The lowest value, being a signal at or near ground (0V), should be zero. The largest reading, a signal just at or above the AREF voltage, should return a value of 1,023. These numbers are always integers, having no fractional part or digits to the right of the decimal point. Each bit in the result represents AREF/1,024. On the Arduino Uno, AREF is 5.0V, so each bit represents approximately 4.88mV, or 0.00488V. If your Arduino is running on a different voltage—for example, the Arduino Pro or Arduino Mini at 3.3V—you need to adjust the sketch accordingly.

To improve the accuracy of the readings, you can use the technique of *oversampling*. You take more than one reading and calculate the average (technically, the arithmetic mean) of all the readings by dividing the sum of all the readings by the number of readings taken.

The number of samples specified in the example sketch was chosen to produce a summary voltage report about 4 times every second (specifically, it turns out to be closer to 3.496 reports per second). This is fast enough to be useful and slow enough to be fairly accurate.

This reporting frequency wasn't chosen at random, however. The ADC is clocked by a signal derived from the system clock. The optimum ADC frequency is stated in the AVR datasheet as 50KHz to 200KHz. The Arduino software selects a clock prescaler of 128, which produces an ADC clock of 125KHz, well within the optimal range.

Each ADC conversion, after the first calibration conversion has been completed, takes exactly 13 ADC clocks. This results in a maximum sampling rate of just over 9,615 conversions per second, assuming a system clock of 16MHz. This doesn't take into account the time required to read the conversion from the ADC hardware, add it to the running total, or keep track of the loop count.

Theoretically, by using the ADC's conversion-complete interrupt, you could accumulate 2,500 samples and report the resulting voltage as many as 3.846 times per second. That's not a giant increase in reporting frequency—only about 10%. However, it may be a good idea in certain applications to have the ADC conversions happening in the background while a higher-level application executes in the foreground. Let's see what it takes to get that going; see Listing 7-10.

Listing 7-10: Arduino as Automatic Voltmeter

```
#define SAMPLES 2500

void setup() {
  Serial.begin(9600);
  Serial.println("Arduino Automatic Voltmeter");
  ADMUX = 1<<REFS0; // select ADC0 (A0), AREF=AVCC (5.0V)
  ADCSRA = 1<<ADEN | 1<<ADSC | 1<<ADATE | 1<<ADIE | 1<<ADPS2 | 1<<ADPS1 | 1<<ADPS0;
  ADCSRB = 0; // free running mode
  bitSet(DIDR0, ADC0D); // disable digital input on ADC0
}

void loop() {
  // nothing happens here
}

ISR(ADC_vect) {
```

```
  static unsigned int i = 0; // sample counter
  static unsigned long voltage = 0; // voltage reading accumulator

  voltage += ADC; // accumulate voltage readings
  i++; // also count samples taken
  ADMUX = 1<<REFS0; // re-select ADC0 (A0), AREF=AVCC (5.0V)

  if(i >= SAMPLES) {
    Serial.print((((voltage * 5.0) / 1024) / SAMPLES), 4); // report
    Serial.println("V"); // label units of measure
    voltage = 0; // reset voltage readings
    i = 0; // reset sample count
  }
}
```

Compile and upload this sketch, and then open the Serial Monitor window to see the results. It should behave a lot like the previous sketch. Take some voltage readings to make sure things are still in order.

The main difference in this automatic version of the voltmeter sketch starts in the `setup()` function. The ADC peripheral is configured to generate an interrupt when it completes a conversion, by setting the `ADIE` bit in the `ADCSRA` register to `1`. In addition, the free-running mode of the ADC is enabled through the combination of the ADC Auto Trigger Enable bit (`ADATE`), which is also in the `ADCSRA` register, and the lower bits of the `ADCSRB` register, which you set to all zeros. This operational mode instructs the ADC to automatically start a new conversion after the previous conversion completes. Other conversion triggers are available as well, including interrupts from the AC, the external interrupt request 0, and select timer/counter interrupts. The free-running mode ensures the fastest conversion turnover, triggering a new conversion immediately after the previous conversion is finished.

Because you're using oversampling to augment the limited resolution of the ADC, it's possible to speed up the ADC clock by choosing a smaller prescale divisor without losing too much accuracy. Selecting the divide-by-64 divisor effectively doubles the sample rate while reducing the effective resolution by one bit.

Shortened conversion cycles means less available time to handle the end-of-conversion interrupt. This leads to another important observation about the example sketch. The optimum interrupt handler gets in quickly, does its job, and exits promptly. Performing a lot of floating-point math, formatting reports, and sending data via the relatively slow serial port aren't the kinds of things you want in a production-ready interrupt handler. Ideally, the interrupt handler aggregates the data and sets a flag, indicating to the foreground process that a report is ready.

External Interrupts

The Arduino software has a little support for using interrupts in your sketches. It uses interrupts internally for timing functions. It makes provisions for enabling and disabling interrupts globally, using the `interrupts()` and `noInterrupts()` macro definitions, which resolve to the `SEI` and `CLI` machine-language instructions, respectively.

The only other support present in the Arduino software for interrupts is the `attachInterrupt()` and `detachInterrupt()` functions. These functions allow you to connect a function of your own devising to the available *external interrupts* of the AVR core. These interrupts are tied directly to the core, and their inputs share pins with the other general-purpose I/O lines in both the ATmega328 and the ATmega2560. See Table 7-6 (n/c stands for not connected).

Table 7-6: External Interrupts

Open table as spreadsheet

External Interrupt	Arduino Uno		Arduino Mega 2560	
	Port	Pin	Port	Pin
INT0	PD2	2	PD0	21
INT1	PD3	3	PD1	20
INT2			PD2	19
INT3			PD3	18
INT4			PE4	2
INT5			PE5	3
INT6			PE6	n/c
INT7			PE7	n/c

The ATmega328 has two external interrupts, INT0 and INT1. They share pins with PD2 and PD3, respectively. These I/O pins can be configured as inputs (with or without enabling the built-in pullup resistors) or as outputs.

The ATmega2560 has eight external interrupts, INT0-INT7. They share pins with I/O lines from Ports D and E. Only six of the available external interrupts, INT0-5, are connected to anything on the Arduino Mega 2560 I/O Board.

External interrupts work in a manner similar to the pin-change interrupts discussed earlier. External interrupts, although more scarce, are more flexible in their configurations. Each of the external interrupts can be individually enabled or disabled, just like the pin-change interrupts. However, each external interrupt has its own interrupt vector. You can configure the external interrupts to detect pin changes, like the pin-change interrupts, but also program them to only respond to rising edges, falling edges, or low levels. This allows a bit more discrimination in determining what external signal should generate an interrupt.

Let's take a look at this signal-filtering capability of the external interrupts, using the Arduino-supplied functions; see Listing 7-11. Just add the bold statements to the Blink example sketch.

Listing 7-11: Exploring External Interrupt Trigger Configurations

```
void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
  Serial.begin(9600);
  attachInterrupt(0, tattle, CHANGE); // jumper D2 to D13
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}

void tattle(void) {
  Serial.println("I'm telling!");
}
```

In the `setup()` function, you additionally initialize the serial port and attach the function `tattle()` to external interrupt 0, INT0, to trigger on any `CHANGE` in the incoming signal. Notice that the second parameter of the `attachInterrupt()` function doesn't contain any sort of special punctuation, such as parentheses. In C, this doesn't invoke or call the function, but serves as a reference to its address in the program space. The Arduino-supplied `attachInterrupt()` function uses this address as a destination to jump to when one of the predefined external interrupt vectors is executed.

The `tattle()` function is called when the external interrupt fires. If you attach a jumper wire between D13 (the LED output) and D2 (the external interrupt input), you should see some output from the serial port, both in the Serial Monitor window (the preferred method) and also on the TX LED on the Arduino I/O Board. This should coincide with the alternating of the LED.

> **Note** The `tattle()` function in the previous exercise is *not* an interrupt handler, per se. It's simply a function, like any other function. The Arduino software has already provided the appropriate interrupt handlers for all the external interrupts—`INT0_vect` and so on—and these functions in turn call the user-written handler function when enabled by the `attachInterrupt()` function.

The other possible conditions acceptable to the external interrupts are `RISING`, `FALLING`, and `LOW`. Try them now, in place of the `CHANGE` parameter of the `attachInterrupt()` function. `RISING` should only produce a tattle as the LED begins to illuminate, but not when it's extinguished. `FALLING` should produce the opposite effect.

When you try `LOW`, however, something seems to go wrong. The LED quits blinking, and a steady stream of tattling ensues. What happened?

The `LOW` setting is somewhat special. When the input signal is low, the interrupt fires and *continues* to fire as long as the input stays low. In this case, the external interrupt is detected, and execution proceeds to your little `tattle()` function. The `tattle()` function sends out a message via the serial port and then returns. The low condition persists, however, because your foreground program doesn't get a chance to change the state of the LED before the external interrupt again is invoked—and it never will.

Try not to code yourself into this particular corner, if you can help it. Don't interrupt on a low-level signal that won't eventually

change of its own volition, or none of your other programs will ever get a chance to run again. The problem is compounded because the external interrupts have the highest priority in the interrupt structure of the AVR, so even another, active interrupt handler won't be able to execute while this condition persists.

It *is* possible, on the other hand, to disable the external interrupt in the interrupt handler routine, thus knocking the computer chip out of its rut. Add a `detachInterrupt(0);` statement to the end of the `tattle()` function. This solves the problem of the endlessly reinterrupting interrupt, but it also disables the intended function of the sketch.

Interrupt Reference

Table 7-7 and Table 7-8 provide lists of all the possible interrupt sources for both the Arduino Uno and the Arduino Mega 2560.

Table 7-7: Interrupt Vectors for the ATmega328

Open table as spreadsheet

Vector	Interrupt	Handler	Description
1	INT0	INT0_vect	External interrupt 0
2	INT1	INT1_vect	External interrupt 1
3	PCINT0	PCINT0_vect	Pin-change interrupt 0
4	PCINT1	PCINT1_vect	Pin-change interrupt 1
5	PCINT2	PCINT2_vect	Pin-change interrupt 2
6	WDT	WDT_vect	Watchdog timer (when used as an interrupt)
7	TIMER2_COMPA	TIMER2_COMPA_vect	Timer/Counter 2 compare match A
8	TIMER2_COMPB	TIMER2_COMPB_vect	Timer/Counter 2 compare match B
9	TIMER2_OVF	TIMER2_OVF_vect	Timer/Counter 2 overflow
10	TIMER1_CAPT	TIMER1_CAPT_vect	Timer/Counter 1 capture
11	TIMER1_COMPA	TIMER1_COMPA_vect	Timer/Counter 1 compare match A
12	TIMER1_COMPB	TIMER1_COMPB_vect	Timer/Counter 1 compare match B
13	TIMER1_OVF	TIMER1_OVF_vect	Timer/Counter 1 overflow
14	TIMER0_COMPA	TIMER0_COMPA_vect	Timer/Counter 0 compare match A
15	TIMER0_COMPB	TIMER0_COMPB_vect	Timer/Counter 0 compare match B
16	TIMER0_OVF	TIMER0_OVF_vect	Timer/Counter 0 overflow
17	SPI	SPI_STC_vect	SPI serial transfer complete
18	USART_RX	USART_RX_vect	USART receive complete
19	USART_UDRE	USART_UDRE_vect	USART data register empty
20	USART_TX	USART_TX_vect	USART transmit complete
21	ADC	ADC_vect	ADC conversion complete
22	EE_READY	EE_READY_vect	EEPROM ready
23	ANALOG_COMP	ANALOG_COMP_vect	Analog comparator triggered
24	TWI	TWI_vect	Two-wire interface (I2C) event
25	SPM_READY	SPM_READY_vect	Self-programming event

Table 7-8: Interrupt Vectors for the ATmega2560

Open table as spreadsheet

Vector	Interrupt	Handler	Description
1	INT0	INT0_vect	External interrupt 0
2	INT1	INT1_vect	External interrupt 1
3	INT2	INT2_vect	External interrupt 2
4	INT3	INT3_vect	External interrupt 3
5	INT4	INT4_vect	External interrupt 4
6	INT5	INT5_vect	External interrupt 5

Vector	Interrupt	Handler	Description
7	INT6	INT6_vect	External interrupt 6
8	INT7	INT7_vect	External interrupt 7
9	PCINT0	PCINT0_vect	Pin-change interrupt 0
10	PCINT1	PCINT1_vect	Pin-change interrupt 1
11	PCINT2	PCINT2_vect	Pin-change interrupt 2
12	WDT	WDT_vect	Watchdog timer (when used as an interrupt)
13	TIMER2_COMPA	TIMER2_COMPA_vect	Timer/Counter 2 compare match A
14	TIMER2_COMPB	TIMER2_COMPB_vect	Timer/Counter 2 compare match B
15	TIMER2_OVF	TIMER2_OVF_vect	Timer/Counter 2 overflow
16	TIMER1_CAPT	TIMER1_CAPT_vect	Timer/Counter 1 capture event
17	TIMER1_COMPA	TIMER1_COMPA_vect	Timer/Counter 1 compare match A
18	TIMER1_COMPB	TIMER1_COMPB_vect	Timer/Counter 1 compare match B
19	TIMER1_COMPC	TIMER1_COMPC_vect	Timer/Counter 1 compare match C
20	TIMER1_OVF	TIMER1_OVF_vect	Timer/Counter 1 overflow
21	TIMER0_COMPA	TIMER0_COMPA_vect	Timer/Counter 0 compare match A
22	TIMER0_COMPB	TIMER0_COMPB_vect	Timer/Counter 0 compare match B
23	TIMER0_OVF	TIMER0_OVF_vect	Timer/Counter 0 overflow
24	SPI	SPI_STC_vect	SPI serial transfer complete
25	USART0_RX	USART0_RX_vect	USART0 receive complete
26	USART0_UDRE	USART0_UDRE_vect	USART0 data register empty
27	USART0_TX	USART0_TX_vect	USART0 transmit complete
28	ANALOG_COMP	ANALOG_COMP_vect	Analog comparator triggered
29	ADC	ADC_vect	ADC conversion complete
30	EE_READY	EE_READY_vect	EEPROM ready
31	TIMER3_CAPT	TIMER3_CAPT_vect	Timer/Counter 3 capture event
32	TIMER3_COMPA	TIMER3_COMPA_vect	Timer/Counter 3 compare match A
33	TIMER3_COMPB	TIMER3_COMPB_vect	Timer/Counter 3 compare match B
34	TIMER3_COMPC	TIMER3_COMPC_vect	Timer/Counter 3 compare match C
35	TIMER3_OVF	TIMER3_OVF_vect	Timer/Counter 3 overflow
36	USART1_RX	USART1_RX_vect	USART1 receive complete
37	USART1_UDRE	USART1_UDRE_vect	USART1 data register empty
38	USART1_TX	USART1_TX_vect	USART1 transmit complete
39	TWI	TWI_vect	Two-wire interface (I2C) event
40	SPM_READY	SPM_READY_vect	Self-programming event
41	TIMER4_CAPT	TIMER4_CAPT_vect	Timer/Counter 4 capture event
42	TIMER4_COMPA	TIMER4_COMPA_vect	Timer/Counter 4 compare match A
43	TIMER4_COMPB	TIMER4_COMPB_vect	Timer/Counter 4 compare match B
44	TIMER4_COMPC	TIMER4_COMPC_vect	Timer/Counter 4 compare match C
45	TIMER4_OVF	TIMER4_OVF_vect	Timer/Counter 4 overflow
46	TIMER5_CAPT	TIMER5_CAPT_vect	Timer/Counter 5 capture event
47	TIMER5_COMPA	TIMER5_COMPA_vect	Timer/Counter 5 compare match A
48	TIMER5_COMPB	TIMER5_COMPB_vect	Timer/Counter 5 compare match B
49	TIMER5_COMPC	TIMER5_COMPC_vect	Timer/Counter 5 compare match C
50	TIMER5_OVF	TIMER5_OVF_vect	Timer/Counter 5 overflow
51	USART2_RX	USART2_RX_vect	USART2 receive complete
52	USART2_UDRE	USART2_UDRE_vect	USART2 data register empty

Vector	Interrupt	Handler	Description
53	USART2_TX	USART2_TX_vect	USART2 transmit complete
54	USART3_RX	USART3_RX_vect	USART3 receive complete
55	USART3_UDRE	USART3_UDRE_vect	USART3 data register empty
56	USART3_TX	USART3_TX_vect	USART3 transmit complete

Previous Next