Beyond the Arduino 7

It's All in the Timers

I have to own up – we've been writing terribly inefficient code in this series. That's right. Every time we've blinked an LED or read a temperature or logged an event, we've been wasteful. Thankfully, that is all about to change ...

It's All in the Timing

Time dictates how we live our lives. Most of us need to conform to a schedule specified by our working hours, meetings, train timetables, and kid's bath times. Even our recreation time is governed by the clock on the wall: the starting time of a sports game, arriving at the cinema in time for a film, not missing a dinner reservation.

It isn't that much of a jump to expect that our microcontroller projects are even *more* dependent on timing – albeit at a time scale that is magnitudes smaller than ours. Many of our projects so far have relied on this – baud rates, I²C frequencies, even the simplest blinking of an LED all rely on accurate timing.

Given the importance of timing, we could reasonably expect that our microcontroller should have some fairly clever mechanisms to allow us to manage timing in our projects – and we would be right. This mechanism comes in the form of something called a timer.

A Timer! Of Course!

It sounds so logical – a timer to control the timing on your projects. So, why do we need a dedicated timer? We were doing just fine with our *_delay_ms()* functions that we were calling in earlier projects. Why would we want to complicate our lives further with another peripheral on our microcontroller?

To answer that question, let's ignore the other functionality that the timer provides and just focus on the ability to implement a delay for a specific period. The answer lies in the implementation of the *_delay_ms()* function. It is essentially a loop that "burns up" clock cycles in order to cause a delay for a specified period of time. So, if you ask for a one second delay, the MCU sits there spinning its wheels for one second counting out clock cycles – and all the while not able to perform any other function.

While it may sound like that is exactly what we want – the microcontroller to halt all other operations for that one second – it isn't really that efficient. Wouldn't it make much more sense to have the microcontroller carry on with other tasks, and then just tell us when the one second is up?

Often, the microcontroller may not have any other tasks to carry on with, in which case we could let it go to sleep (dramatically reducing its power usage) and then wake up when the one second is over. This is where timers come into play.

When I first made the transition from the Arduino environment to raw AVR programming, the concept of a timer made complete sense – until I started thinking through the mechanics: How does it keep time? How do I use it (Do I need to keep checking the time?)? What unit of time does it keep? What if I'm operating at a different clock speed?

So, of course, I headed to the datasheet, which (as was typical in my earlier days) threw me into total confusion. I poured myself a scotch and called it a night, with a plan to revisit the topic more cautiously the following day.

A Cautious Approach to Timers

The following day, I set the datasheet aside and began researching timers without getting into the specifics of Atmel's timers. I think that's a good place to start here too.

So, what exactly is a timer? It couldn't really get any simpler: A timer is a *counter* which increments (or in certain cases, decrements) in time with the CPU clock. The current count of the timer is stored in a register that we can access from our code. Sound useful? Not really!

By Andrew Retallack
www.crash-bang.com

Off the back of their basic counter functionality, timers generally offer a range of features that actually do make them useful.

Prescalers

We've dealt with prescalers in the past; they are like a gearbox for the microcontroller's clock, ratcheting down a fast clock into speeds that are more manageable. A 16 MHz clock ticks 16 million times a second, resulting in a time-slice that is far too small to be practical for most applications. A 1024 prescaler slows that down by a factor of (you guessed it) 1024, resulting in an achieved frequency of 15.6 kHz – far more useful (although still at a pretty high resolution). We'll be using the prescaler at its maximum value for this article.

Interrupts

Yes, interrupts come into play once again as an important tool. Timers are usually able to generate a number of different types of interrupts; the two most common being an overflow and compare interrupt. Let's look at the overflow first, as this is the simplest.

Remember how the current value of the timer is stored in a register and that the registers we've dealt with so far have been eight-bit registers that store a value from 0 to 255? So, what happens when the counter reaches 255? As with most registers (and variables in many languages), it "overflows" back to a value of zero and starts counting up again. You can configure the timer so that it generates an interrupt each time this overflow occurs.

The overflow interrupt is great, but only if you want to measure time in multiples of 1/61 of a second (that's how long it takes the timer to count up to 255 off a 16 MHz clock). What if you want to measure a different value? The answer is that you can set the timer to generate an interrupt at *any* value between 0 and 255 by setting a value in a compare register (on the ATmega328P, this is called an Output Compare register). If you set a value of 127 into this register, an interrupt will fire every time the timer count reaches 127.

Now, we've gained a bit of control and are actually getting somewhere!

Controlling a Pin

In addition to generating an interrupt, many timers can directly change the level of an output pin. This can be really useful if you want, for example, to have an LED flash away without any interruption or variation. Without putting any strain at all on your microcontroller, you can simply "set and forget" and the timer will make it happen.

Configure the registers and away you go. On the ATmega328P, there are only certain pins that this can happen on; those marked OC0A, OC1A, OC2A, OC0B, OC1B, and OC2B.

Timing an Event

Not only can the timer affect a change on a pin, it can also be used to detect a change on a pin. When a change on a pin occurs, the value of the timer is stored in a "capture" register, and an interrupt is generated to let you know that there's a value waiting to be processed. You could think of this functionality as a stopwatch for microcontrollers – useful for timing very small events; for example, measuring the frequency or time-based levels of an external component (ever wanted to build a basic digital oscilloscope?).

Generating PWM Signals

Timers are also able to generate PWM (pulse-width modulation) signals; in fact, this is how many microcontrollers implement PWM functionality. In the world of Arduino, we used the *analogWrite()* function to achieve this. Behind the scenes, however, it was all being run by the timers.

It is a little more complex in the "raw" world of AVR microcontrollers, but an understanding of the concepts will allow you to work with a much broader range of microcontrollers than the Arduino ecosystem allows.

Time to Talk Frequency

When I first started exploring microcontrollers in general and timers in particular, I was taken back to my high school days as I began to delve into frequency and period. Perhaps a refresher will help you in the same way it did me.

Frequency is the number of times an event occurs in a specific time period. In the world of computing, we measure frequency in Hertz (Hz), which refers to the number of events per second. Our microcontroller runs at 16 MHz, meaning that its clock ticks 16 million times in a second. In our LED projects, we typically blink the LED once a second; in other words, at a frequency of 1 Hz.

Period (or the time taken for one clock tick to complete) is the reciprocal of frequency. If you divide "1" by a frequency, the result given is a duration in seconds; 16 MHz results in a period of 62.5 nanoseconds, or 0.0000000625 seconds. These are pretty short slivers of time!

A clock prescaler divides the frequency of a clock source (e.g., the main clock of your microcontroller) by a specific number. So, the 1024 prescaler will generate a "tick" for every 1024 cycles of the main clock. The result is that the frequency decreases to 15.625 kHz (16,000,000 / 1024). Remember that period is the inverse of frequency, so the decrease in frequency results in an increase in period. We now have a period of 64 microseconds (or 0.000064 seconds).

Feature	Timer0	Timer1	Timer2
Precision	8-bit	16-bit	8-bit
Maximum Value	255	65,535	255
Output Compare	Yes — 2	Yes — 2	Yes — 2
Input Capture	No	Yes	No
PWM — Fast	Yes	Yes	Yes
PWM — Phase Correct	Yes	Yes	Yes
PWM — Phase & Frequency Correct	No	Yes	No
Asynchronous Operation	No	No	Yes

Table 1: Core timer features.

The Watchdog Timer

Finally (for now), we have the oddly-named watchdog timer. The main purpose of the watchdog is to keep an eye on your project, and initiate a reset if the project seems to be "hanging" in a non-responsive state. This is a good way to restart your project if it gets caught in an infinite loop — although, it shouldn't be relied on as a substitute for poor design and testing.

To implement the watchdog timer, you need to set a *timeout* value. The watchdog timer needs to be reset ("stroked" as it's commonly known) before the *timeout* value is reached. Otherwise, the microcontroller will be reset. There's more to watchdogs, but we won't touch on that right now.

Let's Get Specific

After that long list of features, let's take a look at how timers are implemented on our ATmega328P microcontroller. For this article, we're going to focus on the compare and overflow functionality of the ATmega328P timer; in future articles, we'll take a look at the capture and PWM features as I'd be doing them an injustice with a cursory discussion!

Three Timers to Play With

The ATmega places three different timers at our disposal: Timer0, Timer1, and Timer2 — meaning that we're able to have three sets of "timing" functionality running simultaneously. While in essence the three timers perform the same function, there are a few differences that may lead you to choose a specific timer for your project. Two of the timers (0 and 2) are eight-bit timers; they cover a shorter time period, overflowing after a value of 255. Timer 1 is a 16-bit timer, so it only overflows after 65,535. If we're running at 16 MHz with a 1024 prescaler, this translates into a possible time period of a little over four seconds.

Table 1 gives a quick comparison of the three timer's features.

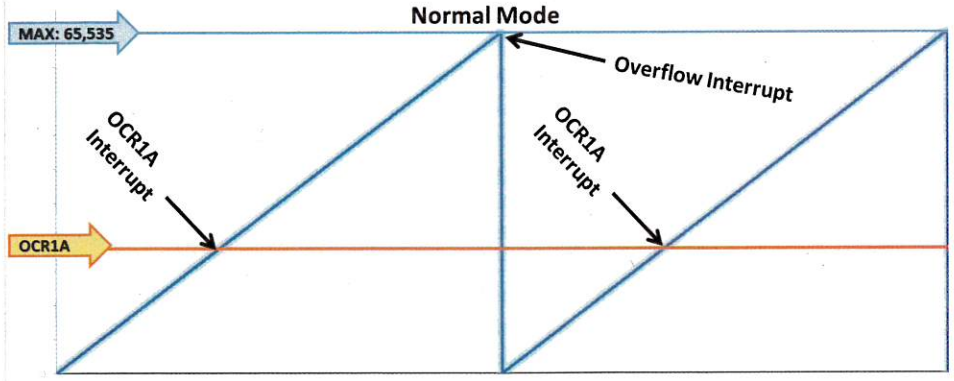

Figure 1: Timing diagram showing timer operating in Normal mode.

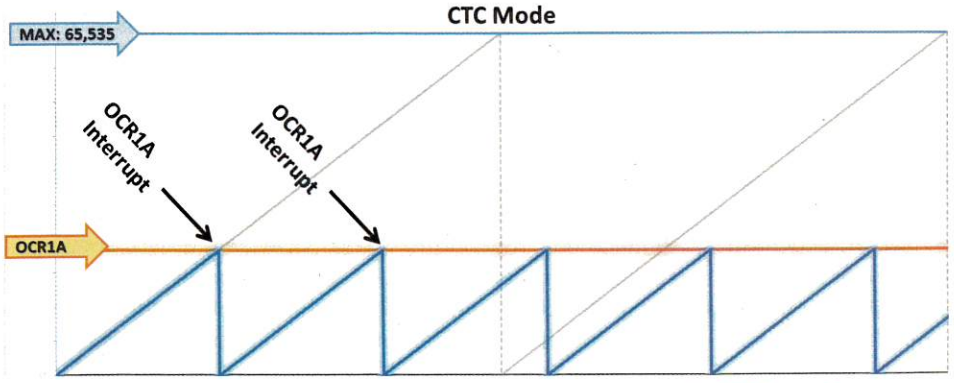

Figure 2: Timing diagram showing timer operating in CTC mode.

Deciding on the Mode

There are a number of modes that we can set the timers to run in, as **Table 1** hinted at. For this article, we'll focus on the two most common modes in order to establish a firm base; we'll then move on to others in future articles.

Normal Mode:

This is the simplest mode, starting at zero and counting up to the maximum values of 255 for eight-bit timers or 65,535 for 16-bit timers. Once the timer reaches the top, it simply overflows back to zero and starts counting up again, and again, and again.

Clear Timer on Compare (CTC Mode):

This mode is similar to the

Normal mode, except that it allows you to define the maximum value the timer counts up to, using the Output Compare register. This value, of course, needs to be less than or equal to the maximum that the timer can count up to. Once the Output Compare value is reached, the timer overflows to zero and counts up again.

For example, if you set Timer0's Output Compare register to 127, the timer would count up to 127 and then start from zero again. This is the mode that I normally use for any timing-related functionality in my applications, as it gives me control over the timer period.

Let me Inter...rupt You

It's one thing to set the timer's mode, but another to actually act on the timer values. This is where interrupts come into play. I mentioned earlier that we have two timer-related interrupts that we can choose to enable on the ATmega328P: the Overflow interrupt and the Output Compare interrupt.

The Overflow interrupt fires whenever the timer overflows the maximum timer value, i.e., 255 for Timer0 and Timer2, or 65,535 for Timer1. The Output Compare interrupt fires whenever the timer counts to the value in the Output Compare register. It's important to note that the Output Compare interrupt will trigger in both Normal and CTC modes. The difference is that in Normal mode, the timer will keep counting beyond the Output Compare value up to the maximum, whereas in CTC mode it will restart from zero.

If this all sounds a little confusing, then take a look at **Figures 1** and **2**. **Figure 1** shows the timer operating in Normal mode, while **Figure 2** shows the timer in CTC mode. You'll note that in Normal mode, the timer keeps incrementing all the way to 65,535 with both the Output Compare interrupt and the Overflow interrupt firing. In CTC mode, the timer has a shorter period, only counting up to the Output Compare value (OCR1A).

Time Registers

I know, time is marching on and we need to get started with this month's project (well, two projects actually). The final bit of detail we're going to look at is (as you probably expected) an overview of the registers involved. By now, you'll be pretty comfortable navigating the datasheet, but I still find it useful to highlight the registers that we'll be using in our projects. As each of the three timers have slightly different functionality, I've only summarized the registers for the 16-bit timer that we'll be using: Timer1. **Figure 3** contains this summary.

The easiest way to tackle the registers we need is to see them in context, so take a look at **Listing 1** in addition to the summary as we work through these. **Listing 1** contains the code we need to initialize the timer.

Control Registers

So far, we've seen that most peripherals have a number of control registers. Timers are no different. Timer1 has three control registers which configure – among others – the timer mode and the prescaler setting. Take a look at **Listing 1** with reference to **Figure 3**, and you'll see that we only need to use the Control Register B

Additional Timer Features

There are a few features of timers that we won't cover in this article, but are worth touching on to give you more reasons to go exploring! The datasheet covers these items in more detail, but I typically find that I understand things better when I see working examples. Google is indispensable.

Clock Source

We've been driving the timers off the main system clock, but they can alternatively be driven off external sources. This would allow you to, for example, synchronize with another project or module (e.g., a real time clock) that generates a square wave signal. Or, you could hook up a 555 timer at a frequency that you aren't able to generate with your ATmega, and use that to drive your timer. The clock source needs to be at least half the frequency of the system clock (ideally 2.5 times less), due to the way the ATmega samples the frequency.

Asynchronous Operation

Timer2 — which we haven't used yet — allows asynchronous operation. This means that it is able to run at a frequency that isn't linked to the main system clock — completely independently. This would normally be used for attaching an external 32.768 kHz watch crystal (the same type we used last month with our real time clock) which, of course, operates at a much lower (more useful) frequency.

The asynchronous operation is really useful when you start operating in low power modes. If your main clock is shut down to conserve power, your timer can keep operating off the external clock source. We'll cover low power modes in detail in a future article.

Listing 1. Timer1 initialization routine.

```
void Timer_Init(void)
{
    TIMSK1 = 0;    //Disable all interrupts on
                   //timer

    TCCR1B |= (1<<CS12) | (1<<CS10);
                   //Set prescaler to 1024

    TCCR1B |= (1<<WGM12);    //Set to Clear-
                             //Timer-on-Compare-Match (CTC) mode

    OCR1A = 15625;
       // CPU Speed = 16MHz; Prescaler = 1024;
       //cycles per sec = 16000000/1024 = 15625

    TIMSK1 |= (1<<OCIE1A);
       //Enable interrupts on Compare
}
```

TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0
Function	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10

Compare Match Output Mode (A and B) — Waveform Generation Mode

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0
Function	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

Input Capture Configuration — Waveform Generation Mode — Clock Select / Prescaler

TCCR1C – Timer/Counter1 Control Register C

Bit	7	6	5	4	3	2	1	0
Function	FOC1A	FOC1B	-	-	-	-	-	-

Force Output Compare

TCNT1 – Timer/Counter1 Register

Bit	7	6	5	4	3	2	1	0
HIGH	TCNT115	TCNT114	TCNT113	TCNT112	TCNT111	TCNT110	TCNT19	TCNT18
LOW	TCNT17	TCNT16	TCNT15	TCNT14	TCNT13	TCNT12	TCNT11	TCNT10

Timer Value

OCR1A – Output Compare Register A

Bit	7	6	5	4	3	2	1	0
HIGH	OCR1A15	OCR1A14	OCR1A13	OCR1A12	OCR1A11	OCR1A10	OCR1A9	OCR1A8
LOW	OCR1A7	OCR1A6	OCR1A5	OCR1A4	OCR1A3	OCR1A2	OCR1A1	OCR1A0

Timer Output Compare Value

TIMSK1 – Timer/Counter Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
Function	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1

Input Capture Interrupt Enable — Output Compare Interrupt Enable — Overflow Interrupt Enable

Figure 3: Summary of Timer1 registers.

(TCCR1B) to set the prescaler to 1024 and the mode to CTC. In setting the mode, we work with the *Waveform Generation Mode* bits (WGMxx). Don't be misled by the somewhat technical-sounding name; I know that I was initially confused.

Timer/Counter Register

The Timer/Counter register (TCNT1) contains the current value of the timer and also allows you to *set* the value of the timer. If for some reason you choose not to use interrupts for your timing and want to work off the underlying timer values, then this is where you'll find them.

You'll note in the datasheet and summary in **Figure 3** that the register has 16 bits. This is what we'd expect, given that Timer1 is a 16-bit timer. What we may struggle with, though, is working out how to set a 16-bit register on an eight-bit microcontroller. The Timer/Counter register is, in fact, made up of two eight-bit registers (TCNT1L and TCNT1H) containing a low byte and a high byte.

You may remember that we came across a similar conundrum back in the third article (*Nuts & Volts*, May 2015) when reading a 10-bit value off the ADC (analog-to-digital converter). Well, Atmel has kindly solved the Timer1 register issue in the same way as they solved the ADC issue: by providing a single register that *acts* as a 16-bit register. So, the answer to our puzzle is incredibly straightforward. We simply use the TCNT1 register for our read/write operations.

Output Compare Register

The Output Compare register contains (you may have guessed) the value that triggers the Output Compare interrupt. When the timer counts up to the value in the Output Compare register (OCR1A), then the interrupt will fire; assuming, of course, that it is configured to fire. This register needs to match the resolution of the timer.

So, for Timer1, it needs to be a 16-bit register. OCR1A is accessed in the same way as the TCNT1 register is: as a virtual 16-bit register and without the need to split the value being written into high and low bytes.

You may have noticed that we actually have two Output Compare registers – OCR1A and OCR1B. This sounds pretty interesting, so can we set two interrupts to be triggered off the same timer? Yes, we can, but only OCR1A can *clear* the timer if you have set it to run in CTC mode. This means that you would need to set OCR1B to a value lower than that of OCR1A.

Interrupt Mask Register

The final register we're going to look at now is the Timer/Counter Interrupt Mask register (TIMSK1). This is the register that we use to control which interrupts we

want firing. By setting the appropriate bit to a 1, we can specify which of the Input Capture, Output Compare (A and B), and Overflow interrupts will be enabled. Very straightforward!

Bringing It All Together

Now that we've worked through the main registers we'll need to use, the code in **Listing 1** should make some sense. It is actually an extract from our first project this month: Blink.

"Blink? Surely we've advanced beyond a project that simply blinks an LED?"

If you're thinking along those lines, you're 100% correct. We *have* advanced way beyond blinking an LED. However, through all our projects to-date, we've never blinked an LED *efficiently*. So, I'd like to go back to the roots of this series of articles and make that LED blink the way that all LEDs should blink!

You can download the project "Beyond Arduino 7 – 1 Timer Blink" at the article link. Open it up in Atmel Studio, and we'll take a quick look at how it works. **Figure 4** shows the project on a breadboard (the same setup as we had in April), while **Figure 5** shows an alternative using the Toadstool Mega328 (refer to **Resources**).

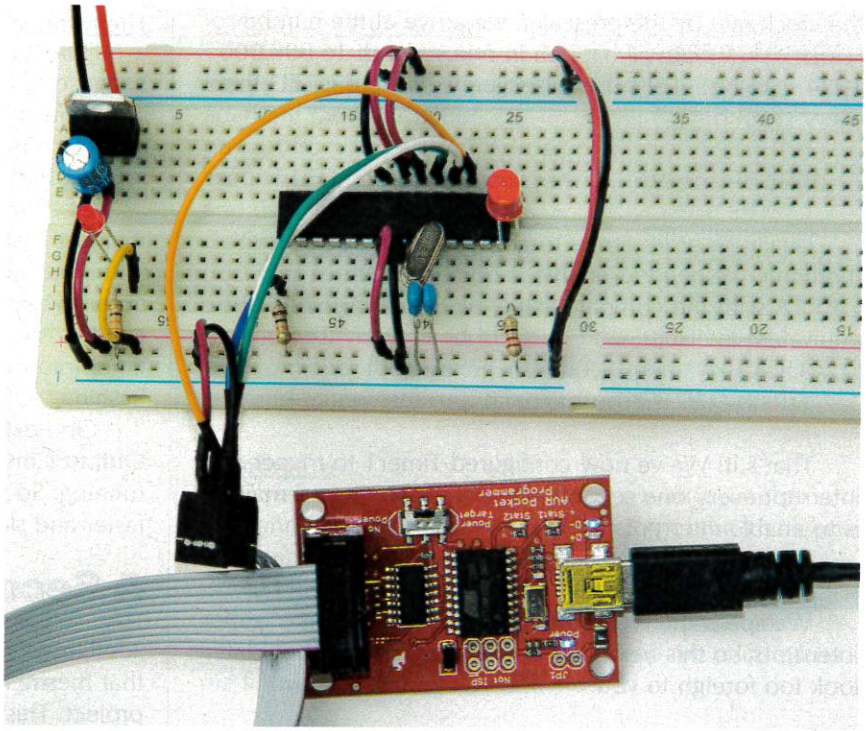

Figure 4: The "Blink" project laid out on a breadboard.

Initializing the Timer

We've spent some time looking at **Listing 1**, so we can run through the logic pretty quickly. As you'll see, this function is called before we enter the *while(1)* loop in our program.

First up, we disable all timer interrupts, just in case they were set previously. We don't want to be disturbed as we configure the timer. As the interrupt bits are all contained in the same register, we simply type:

```
TIMSK1 = 0;
```

The next statement sets the prescaler to 1024 – the maximum value. Take a look at Table 16-5 in the ATmega328 datasheet, and you'll see we need to set bits CS12 and CS10:

```
TCCR1B |= (1<<CS12) | (1<<CS10);
```

Next, we need to set the mode for the timer. From looking at Table 16.4 in the datasheet, it will be apparent that some modes require you to set both TCCR1A and TCCR1B. The CTC mode that we're using only needs WGM12 set:

```
TCCR1B |= (1<<WGM12);
```

With the mode set to Clear on Timer Compare, we, of course, need to set an Output Compare value. By dividing

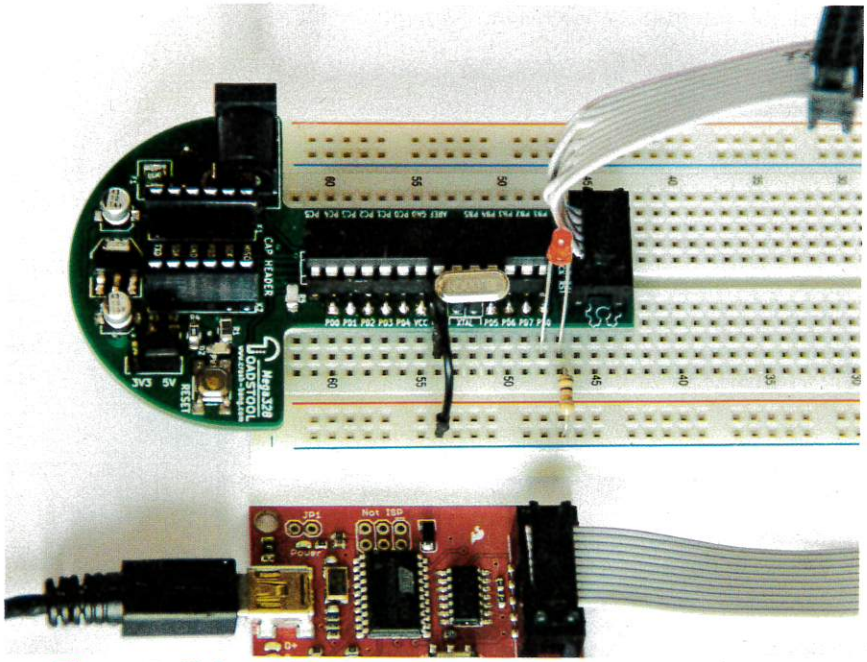

Figure 5: Using a Toadstool Mega328 for the "Blink" project.

the clock rate by the prescaler, we arrive at the number of "clock ticks" that will happen in one second: 16,000,000 / 1024 = 15,625. Let's use this value to make our LED flash every second:

```
OCR1A = 15625;
```

Finally, in order for us to actually do something with the timer, we need to enable an interrupt. As we set OCR1A with the compare value, we need to enable the equivalent interrupt:

```
TIMSK1 |= (1<<OCIE1A);
```

That's it! We've now configured Timer1 to trigger an interrupt every one second. The only other item remaining is to enable interrupts using the *sei()* command.

Handling the Interrupt

We spent the whole July article talking about interrupts, so this next interrupt service routine shouldn't look too foreign to you:

```
ISR(TIMER1_COMPA_vect)
{
    LED_PORT ^= (1<<LED_PIN);
       //Toggle LED
}
```

LED_PORT and *LED_PIN* are simply macros that have been defined as PORTB and PORTB0, respectively. I did this to make the code easier to alter. If I decide to change the pin that the LED is connected to, I only need to make the change once in the initial *#define* statement.

All this interrupt handler does is toggle the LED.

The Main *While(1)* Loop

Huh? Why is the *while(1)* loop empty? It's empty because of the beauty of using timers and their related interrupts. Everything is taken care of by the interrupt service routine which, in turn, is triggered by the timer running in the background. Elegant, don't you think?

Of course, it's unlikely that we'll build many projects that just flash an LED, but having the timer doing its thing in the background means that we don't need to worry about what else we've got to do in the main program. We can focus on the rest of the application, knowing that the timer interrupt will fire like clockwork and keep the LED flashing.

One extra comment: We can change the value of the Output Compare register (OCR1A) while the program is running. So you could, for example, make the LED flash faster and slower in response to some event.

A Second Timer Project

Now that you've managed to tear yourself away from that mesmerizing flashing LED, let's look at another little project. This project is fairly similar, so I've called it "Blink 2" (You'll have to pardon my South African creativity!).

This project is not so much a new project as an adaptation of the existing one. You can download it from the article link, but I'd prefer if you just went along with me to make the changes here. Firstly, why *another* blink project? Is this not getting a little tedious? Well, I'd like to demonstrate the ability for timers to interact directly with specific pins on the microcontroller, without the need for interrupts. Let's see that in action.

1. Move the LED.

In order to have real "hands-free" operation of the

```
#include <avr/io.h>

//Define the LED port and pin
#define LED_PORT        PORTB
#define LED_PIN         PORTB1
#define LED_PORT_DIR    DDRB

//Function Prototypes
void Timer_Init(void);

int main(void)
{

    //Initialise the IO Pins
    LED_PORT_DIR |= (1<<LED_PIN);
            //Set pin to Output
    LED_PORT |= (1<<LED_PIN);
            //Turn the LED on

    Timer_Init();

    while(1)
    {
    }
}
```

Listing 2. Initialize Timer1 to toggle OC1A.

```
void Timer_Init(void)
{

    TIMSK1 = 0;
     //Disable all interrupts on timer

    TCCR1B |= (1<<CS12) | (1<<CS10);
     //Set prescaler to 1024

    TCCR1B |= (1<<WGM12);
     //Set to Clear-Timer-on-Compare-Match
     //(CTC) mode

    TCCR1A = (1<<COM1A0);
     //Toggle OC1A on Compare Match

    OCR1A = 15625;
     // CPU Speed = 16MHz; Prescaler = 1024;
     //cycles per sec = 16000000/1024 = 15625

}
```

LED, we need to move the LED to pin **PB1**. Why PB1? Take a look at the Pin Configuration diagrams in section 1 of the datasheet, and you'll see that PB1 has the following in parentheses: **(OC1A/PCINT1)**.

We're interested in the OC1A portion of this, which means that this pin can be directly controlled by Timer1's Output Compare functionality. So, move the LED, and also change the definition of *LED_PIN* to:

```
#define LED_PIN     PORTB1
```

2. Re-initialize the Initialization Function.

We need to change a couple of things in the *Timer_Init()* function. Firstly, as we won't be using interrupts, you can delete the last line of the function that enables them:

```
TIMSK1 |= (1<<OCIE1A);
```

Secondly, we need to "connect" the timer to pin PB1. We do this using the Control Register A (TCCR1A). A look at Table 16-1 in the datasheet tells us we need the following line of code:

```
TCCR1A = (1<<COM1A0);
    //Toggle OC1A on Compare Match
```

3. No Interruptions, Please.

As we aren't using interrupts any longer, we can:

1. Delete the interrupt handler:
   `ISR(TIMER1_COMPA_vect)`
2. Delete the line that enables global interrupts:
   `sei();`
3. Delete the reference to the interrupt header file:
   `#include <avr/interrupt.h>`

Hands-Free Operation

We're done! **Listing 2** shows the final result. Our timer is now configured to flash the LED without needing any code to manipulate the I/O pin. While the code may not be as short as the initial Blink project we did back in April, it's infinitely more efficient and doesn't clutter up our *while(1)* loop.

What's Next?

I hope that you've found this discussion of timers to be interesting. Timers are very useful in embedded systems, and I incorporate them into my projects often. While we haven't delved into all areas, I think we've covered enough ground to get you up and running.

A Homework Challenge

If you want to explore timers more fully, why not try adapting the first "Blink" project from this article? Add a second LED into the project, and use Timer1's second Output Compare register (OCR1B) to make the LED flash twice as fast as the original one on pin PB0. You'll need to amend the initialization function, alter the existing interrupt handler for OCR1A, and add a new interrupt handler for OCR1B.

Coming Up

I've had some great conversations with a number of you, and find it really interesting learning of the challenges that you face and the projects that you're working on. While I have a number of topics I still want to cover in this series, I'd love to hear if there are specific areas you'd be interesting in learning about. Please drop me a line to let me know your thoughts! **NV**

Resources

Author's website:
www.crash-bang.com

Toadstool Mega328:
www.crash-bang.com/toadstool
http://store.nutsvolts.com