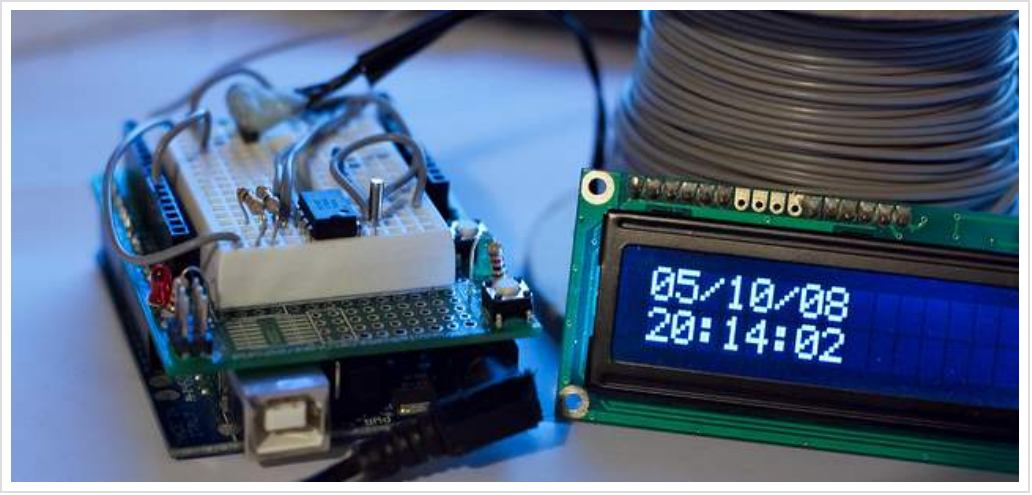
Microcontroller tutorial series: AVR and Arduino timer interrupts



Does your program seem like it's trying to do too much at once? Are you using a lot of `delay()` or `while()` loops that are holding other things up? If so, your project is a good candidate to use timers. In this tutorial, we'll discuss AVR and Arduino timers and how to use them to write better code.

In our prior article, we covered interrupt basics and how to use external interrupts that are triggered by a pin change or similar event. Check it out if you're looking to brush up on interrupts in general.

This chapter moves on to timer interrupts and talks about their applications in Arduino projects or custom AVR circuits. Almost all Arduino boards are powered by AVR 8-bit processors, so to experience the full power of timers you'll use the same techniques no matter which platform you're on. Here's the tutorial's table of contents:

What is a timer?

You're probably familiar with the general concept of a timer: something used to measure a given time interval. In microcontrollers, the idea is the same. You can set a timer to trigger an interrupt at a certain point in the future. When that point arrives, you can use the interrupt as an alert, run different code, or change a pin output. Think of it as an alarm clock for your processor.

The beauty of timers is that just like external interrupts, they run asynchronously, or independently from your main program. Rather than running a loop or repeatedly calling `millis()`, you can let a timer do that work for you while your code does other things.

For example, say you're building a security robot. As it roams the halls, you want it to blink an LED every two seconds to let potential intruders know they'll be vaporized if they make a wrong move. Using normal code techniques, you'd have to set a variable with the next time the LED should blink, then check constantly to see if that time had arrived. With a timer interrupt, you can set up the interrupt, then turn on the timer. Your LED will blink perfectly on cue, even while your main program executes its complicated `terminateVillian()` routine.

How do timers work?

Timers work by incrementing a counter variable, also known as a *counter register*. The counter register can count to a certain value, depending on its size. The timer increments this counter one step at a time until it reaches its maximum value, at which point the counter *overflows*, and resets back to zero. The timer normally sets a flag bit to let you know an overflow has occurred. You can check this flag manually, or you can also have the timer trigger an interrupt as soon as the flag is set. Like any other interrupt, you can specify an Interrupt Service Routine (ISR) to run code of your choice when the timer overflows. The ISR will reset the overflow flag behind the scenes, so using interrupts is usually your best option for simplicity and speed.

In order to increment the counter value at regular intervals, the timer must have access to a *clock source*. The clock source generates a consistent repeating signal. Every time the timer detects this signal, it increases its counter by one.

Because timers are dependent on the clock source, the smallest measurable unit of time will be the period of this clock. For example, if we provide a 1 MHz clock signal to a timer, we can calculate our timer resolution (or timer period) as follows:

```
T = timer period, f = clock frequency

T = 1 / f
T = 1 / 1 MHz = 1 / 10^6 Hz
T = (1 * 10^-6) s
```

Our timer resolution is one millionth of a second. You can see how even relatively slow processors can break time into very small chunks using this method.

You can also supply an external clock source for use with timers, but in most cases the chip's internal clock is used as the clock source. This means that your minimum timer resolution will be based on your processor speed (either 8 or 16 MHz for most 8-bit AVRs).

Types of timers

If you're using any of the standard Arduino variants or an 8-bit AVR chip, you have several timers at your disposal. In this tutorial, we'll assume you're using a board powered by the AVR ATmega168 or ATmega328. This includes the Arduino Uno, Duemilanove, Mini, any of Sparkfun's Pro series, and many similar designs. You can use the same techniques on other AVR processors like those in the Arduino Mega or Mega 2560, you'll just have to adjust your pinout and check the datasheet for any differences in the details.

The ATmega168 and ATmega328 have three timers: Timer0, Timer1, and Timer2. They also have a watchdog timer, which can be used as a safeguard or a software reset mechanism. However, we don't recommend messing with the watchdog timer until you get comfortable with the basics. Here are a few details about each timer:

TIMER0

Timer0 is an 8-bit timer, meaning its counter register can record a maximum value of 255 (the same as an unsigned 8-bit byte). Timer0 is used by native Arduino timing functions such as `delay()` and `millis()`, so you Arduino users shouldn't mess with it unless you're comfortable with the consequences.

TIMER1

Timer1 is a 16-bit timer, with a maximum counter value of 65535 (an unsigned 16-bit integer). The Arduino Servo library uses this timer, so be aware if you use it in your projects.

TIMER2

Timer2 is an 8-bit timer that is very similar to Timer0. It is utilized by the Arduino `tone()` function.

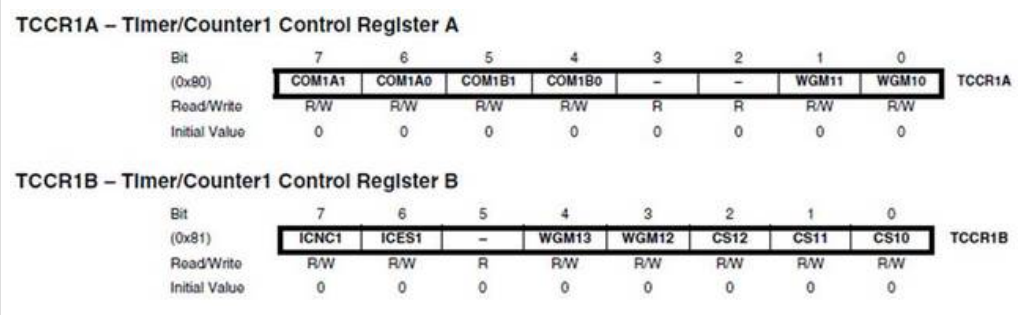
TIMER3, TIMER4, TIMER5

The AVR ATmega1280 and ATmega2560 (found in the Arduino Mega variants) have an additional three timers. These are all 16-bit timers,

and function similarly to Timer1.

Configuring and running the timer

In order to use these timers, we need to set them up, then make them start running. To do this, we'll use built-in registers on the AVR chip that store timer settings. Each timer has a number of registers that do various things. Two of these registers hold setup values, and are called TCCRxA and TCCRxB, where x is the timer number (TCCR1A and TCCR1B, etc.). TCCR stands for *Timer/Counter Control Register*. Each register holds 8 bits, and each bit stores a configuration value. Here are the details, taken from the ATmega328 datasheet:



TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

To start using our timer, the most important settings are the last three bits in TCCR1B, CS12, CS11, and CS10. These dictate the timer clock setting. By setting these bits in various combinations, we can tell the timer to run at different speeds. Here's the relevant table from the datasheet:

Table 16-5. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

By default, these bits are set to zero. Let's use a simple example, and say that we want to have Timer1 run at clock speed, with one count per clock cycle. When it overflows, we'll run an Interrupt Service Routine (ISR) that toggles a LED tied to pin 2 on or off. We'll write Arduino code for this example, though we'll use avr-libc routines wherever they don't make things overly complicated. AVR pros can adapt as they see fit.

First, we initialize the timer:

```
// avr-libc library includes
#include <avr/io.h>
#include <avr/interrupt.h>

#define LEDPIN 2

void setup()
{
        pinMode(LEDPIN, OUTPUT);

        // initialize Timer1
        cli();             // disable global interrupts
        TCCR1A = 0;        // set entire TCCR1A register to 0
        TCCR1B = 0;

        // enable Timer1 overflow interrupt:
        TIMSK1 = (1 << TOIE1);
        // Set CS10 bit so timer runs at clock speed:
        TCCR1B |= (1 << CS10);
        // enable global interrupts:
        sei();
}
```

You'll notice that we used a new register, `TIMSK1`. This is the *Timer/Counter1 Interrupt Mask Register*. It controls which interrupts the timer can trigger. Setting the TOIE1 bit tells the timer to trigger an interrupt when the timer overflows. We can also set other bits to trigger other interrupts. More on that later.

Once we set the CS10 bit, the timer is running, and since we've enabled an overflow interrupt, it will call the `ISR(TIMER1_OVF_vect)` whenever the timer overflows.

Next, we can define the ISR:

```
ISR(TIMER1_OVF_vect)
{
    digitalWrite(LEDPIN, !digitalRead(LEDPIN));
}
```

Now we're free to define our `loop()` and our LED will toggle on and off regardless of what's happening in the main program. To turn the timer off, we can set `TCCR1B = 0` at any time.

However, let's think about how this will work. Using the code we've written, how fast will our LED blink?

We've set Timer1 to interrupt on an overflow, and let's assume we're using an ATmega328 with a 16MHz clock. Since Timer1 is 16 bits, it can hold a maximum value of (2^16 – 1), or 65535. At 16MHz, we'll go through one clock cycle every 1/(16*10^6) seconds, or 6.25e-8 s. That means 65535 timer counts will elapse in (65535 * 6.25e-8s) and our ISR will trigger in, oh… about 0.0041 seconds. Then again and again, every four thousandths of a second after that. Oops. At this rate, we probably won't even be able to detect blinking. If anything, we've created an extremely fast PWM signal for the LED that's running at a 50% duty cycle, so it may appear to be constantly on but dimmer than normal. An experiment like this shows the amazing power of microprocessors – even an inexpensive 8-bit chip can process information far faster than we can detect.

Timer prescaling and CTC

Luckily, the good engineers at Atmel thought of this problem, and included some options. It turns out you can also set the timer to use a *prescaler*, which allows you to divide your clock signal by various powers of two, thereby increasing your timer period. For example, let's say we'd rather have our LED blink at one second intervals. Going back to the TCCR1B register, we can use the three CS bits to set a better timer resolution. If we set CS10 and CS12 using `TCCR1B |= (1 << CS10);` and `TCCR1B |= (1 << CS12);`, we divide our clock source by 1024. This gives us a timer resolution of 1/(16*10^6 / 1024), or 6.4e-5 seconds. Now the timer will overflow every (65535 * 6.4e-5s), or 4.194s. Hm, too long. What can we do?

It turns out there's another mode of operation for AVR timers. This mode is called Clear Timer on Compare Match, or *CTC*. Instead of counting until an overflow occurs, the timer compares its count to a value that was previously stored in a register. When the count matches that value, the timer can either set a flag or trigger an interrupt, just like the overflow case.

To use CTC, let's start by figuring out how many counts we need to get to our one second interval. Assuming we keep the 1024 prescaler as before, we'll calculate as follows:

```
(target time) = (timer resolution) * (# timer counts + 1)
```

and rearrange to get

```
(# timer counts + 1) = (target time) / (timer resolution)
(# timer counts + 1) = (1 s) / (6.4e-5 s)
(# timer counts + 1) = 15625
(# timer counts) = 15625 - 1 = 15624
```

Why did we add the extra +1 to our number of timer counts? In CTC mode, when the timer matches our desired count it will reset itself to zero. This takes one clock cycle to perform, so we need to factor that into our calculations. In many cases, one timer tick isn't a huge deal, but if you have a time-critical application it can make all the difference in the world.

Now we can rewrite our `setup()` function to configure the timer for these settings:

```
void setup()
{
        pinMode(LEDPIN, OUTPUT);

        // initialize Timer1
        cli();                  // disable global interrupts
        TCCR1A = 0;             // set entire TCCR1A register to 0
        TCCR1B = 0;             // same for TCCR1B

        // set compare match register to desired timer count:
        OCR1A = 15624;
        // turn on CTC mode:
        TCCR1B |= (1 << WGM12);
        // Set CS10 and CS12 bits for 1024 prescaler:
        TCCR1B |= (1 << CS10);
        TCCR1B |= (1 << CS12);
        // enable timer compare interrupt:
        TIMSK1 |= (1 << OCIE1A);
        sei();                  // enable global interrupts
}
```

And we'll need to replace our overflow ISR with a compare match version:

```
ISR(TIMER1_COMPA_vect)
{
    digitalWrite(LEDPIN, !digitalRead(LEDPIN));
}
```

That's all there is to it! Our LED will now blink on and off at precisely one second intervals. And as always, we're free to do anything we want in `loop()`. As long as we don't change the timer settings, it won't interfere with our interrupts. With different mode and prescaler settings, there's no limit to how you use timers.

Here's the complete example in case you'd like to use it as a starting point for your own project. Double click to copy:

```
// Arduino timer CTC interrupt example
// www.engblaze.com

// avr-libc library includes
#include <avr/io.h>
#include <avr/interrupt.h>

#define LEDPIN 2

void setup()
{
        pinMode(LEDPIN, OUTPUT);

        // initialize Timer1
        cli();                  // disable global interrupts
        TCCR1A = 0;             // set entire TCCR1A register to 0
        TCCR1B = 0;             // same for TCCR1B

        // set compare match register to desired timer count:
        OCR1A = 15624;
        // turn on CTC mode:
        TCCR1B |= (1 << WGM12);
        // Set CS10 and CS12 bits for 1024 prescaler:
        TCCR1B |= (1 << CS10);
        TCCR1B |= (1 << CS12);
        // enable timer compare interrupt:
        TIMSK1 |= (1 << OCIE1A);
        // enable global interrupts:
        sei();
}

void loop()
{
    // do some crazy stuff while my LED keeps blinking
}

ISR(TIMER1_COMPA_vect)
{
    digitalWrite(LEDPIN, !digitalRead(LEDPIN));
}
```

Going further

Keep in mind that you can use the built-in ISRs to extend timer functionality. For example, if you wanted to read a sensor every 10 seconds, there's no timer setup that can go this long without overflowing. However, you can use the ISR to increment a counter variable in your program once per second, then read the sensor when the variable hits 10. Using the same CTC setup as in our previous example, our ISR would look something like this:

```
ISR(TIMER1_COMPA_vect)
{
    seconds++;
    if (seconds == 10)
    {
        seconds = 0;
        readMySensor();
    }
}
```

Note that in order for a variable to be modified within an ISR, it must be declared as `volatile`. In this case, we'd need to declare `volatile byte seconds;` or similar at the beginning of our program.

This tutorial covers the basics of timers. As you start to understand the underlying concepts, you'll want to check the datasheet for more information on your particular chip. Datasheets are readily available on Atmel's website. To find them, navigate to the page for your device (8-bit AVRs found here) or do a search for your chip model. There's a lot of information to wade through, but the documentation is surprisingly readable if you have the patience.

Otherwise, experiment and have fun! Check out our other tutorials if you're looking for more knowledge, or sign up for our email newsletter for future AVR and Arduino updates.