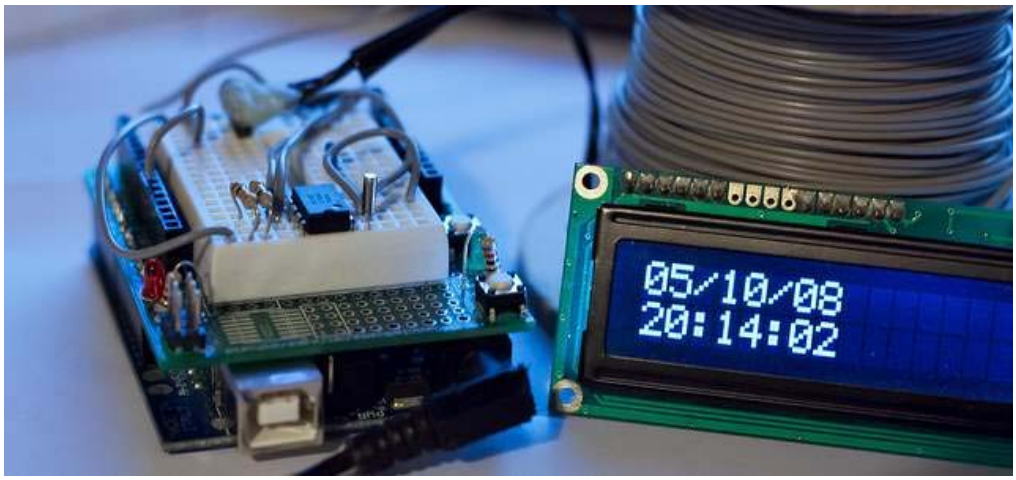
EngBlaze

Arduino, AVR, and hardware hacks, oh my.

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.

This entry was posted in **Arduino**, **AVR**, **Tutorials** and tagged **arduino**, **AVR**, **avr-libc**, **examples**, **interrupts**, **timer**, **tutorial** by **EngBlaze**.

57 THOUGHTS ON "MICROCONTROLLER TUTORIAL SERIES: AVR AND ARDUINO TIMER INTERRUPTS"

Rax
on **January 23, 2012 at 10:27 pm** said:

Nice tutorial, but I have one question.
If I need timer3 on Mega1280 to make interrupt every 40uS, how can I do it?

EngBlaze
on **January 23, 2012 at 10:54 pm** said:

Hi Rax,

Your Mega1280 runs at 16 MHz, the same speed as the processor used in the examples. Timer3 is a 16-bit timer, just like the Timer1 discussed in the tutorial. From the above: at 16MHz, you'll go through one clock cycle every 1/(16*10^6) seconds, or 6.25e-8 s. This is your timer resolution.

Using the equation in the CTC mode section with no prescaler:
(# timer counts) = (target time) / (timer resolution)
(# timer counts) = (40 * 10^-6 s) / (6.25e-8 s) = 640

So you need to generate an interrupt every 640 ticks.

If you set up Timer3 in CTC mode with those parameters using the given example, you should get what you're looking for.

> Rax
> on **January 23, 2012 at 11:22 pm** said:
>
> Without prescaler means I leave out this part:
> Is this correct?
> TCCR1B |= (1 << CS10);
> TCCR1B |= (1 << CS12);
>
> So code will be:
>
> cli();
> TCCR3A = 0;
> TCCR3B = 0;
> OCR3A = 640;
> TCCR3B |= (1 << WGM12);
> TIMSK3 |= (1 << OCIE3A);
> sei();
>
> > **EngBlaze**
> > on **January 24, 2012 at 12:02 am** said:
> >
> > You do need to set the CS10 bit to start the timer running (no prescaler). Using CS11 and CS12 in various combinations is what gets you different prescale values.
> >
> > Otherwise your setup looks good. Just add the "TCCR1B |= (1 << CS10);" line just "sei()" and you should be good to go. Let us know how it works out!
> >
> > > Rax
> > > on **January 24, 2012 at 2:59 am** said:
> > >
> > > I let you know later.
> > > I'm trying to use solid state relays as dimmer.
> > > I'm still noob at this stuff, but I try to learn.
> > >
> > > At the moment I'm busy with my real job. 😛
> > >
> > > Dax
> > > on **January 25, 2012 at 7:21 am** said:

Just remember that 640 clock cycles isn't a whole lot. If your interrupt routine takes more time to process than that, things get funky.

It would probably be better to use the built-in PWM function. It works in a similiar way using timers, but instead of triggering an interrupt it directly toggles a pin without bothering the processor.

Rax
on **January 25, 2012 at 1:00 pm** said:

But I need to trigger output HIGH in sync with AC current waveform. When I detect zero crossing, I set counter 0. That's all I have in interrupt code.
I haven't tested it yet, but on the weekend I try to get there.
I couldn't use PWM, because it only made lights flicker when I was trying to dim down the lights. analogWrite(2, 255) was full on, analogWrite(2, 10) was full on with flicker…

Matt
on **April 12, 2012 at 2:03 pm** said:

In reply to using a relay as a dimmer, that's just about the fastest way to burn out the contacts.

Cheesy
on **September 24, 2012 at 6:20 am** said:

He did say SOLID STATE relay, as far as I am aware, the contacts on one of those are far more robust.

Doug
on **October 26, 2012 at 9:38 pm** said:

Mat, he said solid state relay….. They do not have contacts…….

Segura
on **January 25, 2012 at 4:06 pm** said:

Hey, great tutorial. Amazing! I had just started a clock project (http://www.youtube.com/watch?v=1vVnXmgl0vc) a week ago and I managed to configure the Timer2 but, with 8 bits, I had to use a counter, and it ended up accumulating too much delay. Also I didn't know that tone() uses Timer2 so.. I will use Timer1. I hope it's more precise, its quite crucial in a clock :/

Do you know what can I do to make it more precise? Using an external clock or a 555 timer would be a better option?

PS: On the big code example you give the ISR function is "ISR(TIMER1_OVF_vect)" but as you mention before it should be ISR(TIMER1_COMPA_vect). Also, I didn't have to use a volatile variable, I'm increasing the seconds variable and I get no trouble being "byte seconds;"

EngBlaze
on **January 25, 2012 at 9:15 pm** said:

Thanks for the kind words Segura! Glad to hear you found the tutorial useful. Also, great catch on the wrong interrupt vector – we edited the code to reflect that.

As for not making variables volatile, you might not see adverse affects depending on your code, but there's definitely a possibility. It's good practice to always mark as volatile any variable that will be modified in an ISR. Don't take it from us, take it from the guys that wrote the book: http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_volatile

Best of luck with the clock project!

Rax
on **January 25, 2012 at 6:25 pm** said:

So I tried, but no luck… I need to get Zero Crossing and in this exact moment I need to switch output high for some time, so I could dim the lights.
I thought this would work, but it doesn't… Can You point where I'm wrong?

```
volatile int counterValue;
volatile int requestValue;

void setup(){
pinMode(3, OUTPUT);
attachInterrupt(0, resetCounter, RISING);
cli(); //Disable global interrupts
TCCR3A = 0; //Set register to 0
TCCR3B = 0; //Set register to 0
OCR3A = 640; //Compare match register to get desired timer count
TCCR3B |= (1 << WGM12); //Turn on CTC mode
TIMSK3 |= (1 << OCIE3A); //Enable timer compare interrupt
TCCR1B |= (1 << CS10); //Start timer
sei(); //Enable global interrupts
}

ISR(TIMER1_OVF_vect){
counterValue++;
if (counterValue==requestValue) {
digitalWrite(3, HIGH);
} else {
digitalWrite(3, LOW);
}
}
```

```
void resetCounter(){
counterValue=0;
}

void loop(){
requestValue=25;
}
```

Rax
on **January 25, 2012 at 6:32 pm** said:

Copied the old version, sorry.

TCCR1B is actually TCCR3B
TIMER1_OVF_vect is actually TIMER3_OVF_vect
attachInterrupt(0, resetCounter, RISING); is actually FALLING

Rax
on **January 27, 2012 at 11:08 am** said:

Got it working. Thanks, guys. 🙂
I'll share my sketch in case anybody else needs to make a dimmer for random fire solid state relays.

```
volatile int counterValue;
volatile int requestValue;

void setup(){
Serial.begin(9600);
pinMode(3, OUTPUT);
attachInterrupt(0, resetCounter, CHANGE);
cli();
TCCR3A = 0;
TCCR3B = 0;
OCR3A = 640;
TCCR3B |= (1 << WGM12);
TIMSK3 |= (1 << OCIE3A);
TCCR3B |= (1 << CS10);
sei();
}

ISR(TIMER3_COMPA_vect){
counterValue++;
counterValue &= 0xff;
if (counterValue==requestValue) {
digitalWrite(3, HIGH);
}
else {
digitalWrite(3, LOW);
}
```

```
}

void resetCounter(){
counterValue=0;
}

void loop(){
requestValue=1;
}
```

David Tegerdine
on **January 28, 2012 at 5:13 am** said:

Nice tutorial. I have one correction – when using CTC mode you need to load the OCR1A register with your calculated value minus 1. The number represents the final value of the TCNT register, so if set to 15625 you actually get 15626 cycles per interrupt.

EngBlaze
on **January 29, 2012 at 3:44 pm** said:

Hi David, you're absolutely right. The tutorial has has been updated. Thanks for the correction!

Seeker
on **January 28, 2012 at 4:34 pm** said:

Thank you for having the first microcontroller tutorial on timers that is actually logical, readable, and well-explained! I've tried reading off and on about how to use the timers, but it was never very clear, as they would use terms like watchdog timers and prescalars without bothering to define their purposes. I *finally* have the basic knowledge I needed to use the timers and a starting point to understand the datasheet specifics. I can't tell you how much I appreciate this tutorial!

EngBlaze
on **January 29, 2012 at 3:45 pm** said:

Hi Seeker, very glad you found it useful! Let us know if there are any other topics that you'd like to see, we always want to serve our readers as best we can.

David Tegerdine
on **January 29, 2012 at 6:38 pm** said:

Sorry, ignore this. I thought my earlier comment had gone astray.

Seetron Tech
on **February 7, 2012 at 4:00 pm** said:

Great work! I just linked to this tutorial from my site, where I converted the once-a-second ISR into a clock demo for a display.

> 
>
> **EngBlaze**
> on **February 17, 2012 at 1:55 pm** said:
>
> Thanks for the link! Your clock project looks awesome. That display is so crisp, we might have to pick up one of those bad boys for the lab.

Matt
on **April 12, 2012 at 2:01 pm** said:

Hey, I just wanted to thank you for such a wonderful tutorial! I successfully used it to create IR messages. My library (named PFIR, with the timing based on the concepts discussed in this tutorial) is on my blog: http://mattallen37.wordpress.com/arduino-libraries/ (second library on that page).

> 
>
> **EngBlaze**
> on **April 12, 2012 at 11:07 pm** said:
>
> Hi Matt, glad you found it useful! The IR library looks awesome, we can think of a few projects where it could come in really handy. In fact, that might be something we would want to write about – would you mind if we mentioned it in a short post sometime (with full credit and linking back to your blog of course)?

Ronny Janssen
on **April 15, 2012 at 10:10 am** said:

Thanks.. just what I needed to get rid of the delay(1000) at the end of my loop and to actually 'trigger' the loop() to do it's stuff every second. With reasonable (good enough for me) accuracy that is.

tommaso
on **May 10, 2012 at 3:43 am** said:

i suppose you may help . is there away to read high frequency pulses on pin 5 (timer 1 connected) without stop every time the

counter to see which number of pulses it reached . is possible to use top variabiles to receive an interrupt when the number of pulses are reached ? is it enaugh if i set up external clock source instead of prescaling in your example?

EngBlaze
on **May 13, 2012 at 7:45 pm** said:

Tommaso, I'm not 100% sure what you're asking here. Are you trying to count high frequency pulses, and then read the count without interrupting new reads? That should be possible by incrementing the counter variable with a hardware interrupt. You would have to go through extra lengths if you wanted to ensure that the read was atomic, i.e. making sure the variable doesn't change during a read or other manipulations.

For your second question, what do you mean by top variables? And finally, yes, you can use an external clock source instead of prescaling whenever required.

leonardo
on **May 13, 2012 at 5:58 am** said:

Hi, I have the following issue: I need to develop a program on an Arduino UNO that after it will receive a trigger on an external analog input port, will periodically recall a function. The period will be 1/170 second with hight accuracy.
It's possible to develop? Have I to use an external clock?

Thx a lot!

EngBlaze
on **May 13, 2012 at 7:41 pm** said:

Leonardo, using a timer interrupt to trigger a function every 1/170 of a second shouldn't be a problem. If you use the methods described in the tutorial to correctly set your interrupt interval, you'll be all set.

You do need to check and make sure your function can complete within that time interval. If the function hasn't returned by the time the interrupt triggers again, your program won't operate the way you want.

matt
on **May 14, 2012 at 7:03 am** said:

Hi there, great explanation of the Timer regs… But you gloss over /ignore TCCR1A… My project requires a pin to be turned on at a specific number of "ticks" and then turned off after a certain number more.

I notice that there are two compare registers; OCR1A and OCR1B… So I need an interrupt to be called when the timer matches OCR1A (to turn the pin on) and then a second interrupt to be called when the timer matches OCR1B, the timer then needs to be reset… To start the process again… This isn't PWM as the values are variable and could be altered at any time by the main code 🙂

Looking at the AVR datasheet, it seems the timer can be set up using TCCR1A to allow this… I will play layer to confirm, but it would be nice if you could include more details of this register in your examples 🙂

matt
on **May 14, 2012 at 11:20 am** said:

Appologies for two posts in a row, but I have been reading and it seems that if you set WGM13 and WGM12, then the value of ICR1 will trigger the timer reset, rather than the OCR1A and OCR1B interrupts (which will be called as normal when the timer reaches them). This behaviour is ideal for my project, if it works. I hope this helps someone else 🙂

ArduinoAlien
on **June 8, 2012 at 9:43 am** said:

Wonderful thanks for the contribution !!!!!!!!!!!!!!!!! 🙂

ricky
on **June 12, 2012 at 9:47 am** said:

good tutorial. like this…

Diane Williams
on **July 16, 2012 at 7:07 pm** said:

Thanks for the great tutorial. You make the CTC and overflow timers so easy to understand. Especially like the way you broke down the formula into pseudocode and followed through with consistent examples, as well as superbly documented, working program code.

pradeep
on **July 25, 2012 at 1:23 pm** said:

nice work brother…can you explain how to use three external interrupts in same pic .Here I used push button to make interrupts.plz help me….I used pic 18F4685.

EngBlaze
on **November 26, 2012 at 10:40 pm** said:

> We're not as familiar with PIC as we are with AVR, so I'm not sure if the process would be similar. You might want to try checking out some other resources such as http://www.engineersgarage.com/embedded/pic-microcontroller-projects/pic-external-hardware-interrupts-circuit

Michael Shimniok (Bot Thoughts)
on **September 5, 2012 at 4:02 pm** said:

Excellent tutorial, thanks! I needed a quick reference for not only which timers are being used in Arduino but how to quickly set up a timer-based interrupt handler for doing some scheduling. This did the trick perfectly.

Michael Scott
on **September 11, 2012 at 11:12 am** said:

I was wondering how an interrupt could also be brought in to pause the LED during its blinking?

EngBlaze
on **November 26, 2012 at 10:56 pm** said:

Hi Michael, I'm not sure what you mean. Do you want to use interrupts to begin blinking an LED, but then set another interrupt to stop the blinking at some point?

Greg S
on **September 17, 2012 at 9:05 pm** said:

Great tutorial. I want to generate (16) 32usec wide pulses with programmable delays between the pulse trains for IR emitter/receiver app. The following modification to your final sample code for CTC gave me the right pulse width (continuous stream of pulses, no gating of pulse trains yet):

```
// set compare match register to desired timer count:
OCR1A = 3;
// turn on CTC mode:
TCCR1B |= (1 << WGM12);
// Set CS10 and CS12 bits for 1024 prescaler:
TCCR1B |= (1 << CS10);
TCCR1B |= (1 << CS11);
```

however the pulse train has some jitter. Every 4th rising edge jumps back and forth several usec. I expected zero jitter.

Thanks for extra insight

EngBlaze

on **September 25, 2012 at 5:42 pm** said:

Hm, not sure what might be causing that. What model Arduino do you have? Some models use a ceramic resonator as the clock source, which might be less stable than the quartz crystal found in the official versions. Anything else that might cause noise in the clock source, like varying input voltage?

oldbaritone
on **December 27, 2012 at 11:00 am** said:

since it's every 4th pulse (I'm speculating "every 4th or 5th") I would check to see if another timer interrupt, perhaps Arduino's own timer tick or serial clock, is disabling interrupts and increasing latency because of something that happens outside of your process, but impacts your pulse because it's occurring during the other interrupt. Perhaps each 4th pulse happens to occur during another ISR elsewhere. Probably that would be internal to another Arduino library, so it may take a lot of digging to find it, but I would suspect something like that rather than instability in a crystal oscillator that is apparently stable for the other 3 pulses. A quick-check for the serial timer might be to try initializing serial to 1200 baud and see if the symptoms change. That's just a guess because 32uS * 4 = 1.28 ms, not too far from the roughly 1 ms clock for both 9600 baud and Arduino's millisecond timer. Failing that, it might be something requiring a real-time in-circuit-emulator to catch it.

Miranda
on **September 29, 2012 at 11:21 am** said:

Great tutorial! Finally found an easy way to learn about arduino timers.

I was wondering if there's a way to read the current timer value.
Let say I use timer1 to count pulses on T1 pin. Is it possible to set a second timer to countdown 1 second and set it's ISR to read the timer1 current counting value?

EngBlaze
on **November 26, 2012 at 10:56 pm** said:

Thanks Miranda, glad you found it useful!

To read a timer's current value, you can access the TCNT registers. For example, `if (TCNT1 >= 5000)` will test if the current value of timer 1 is greater than or equal to 5000. For more detail, you can check the datasheet for your chip, but general usage should be equally straightforward.

vivek
on **October 2, 2012 at 12:44 am** said:

i am fairly new to coding with microprocessors , can u help me make a 1 houtr timer that could count down from 59:99 to 0 . i am using a atmel ATmega 168 processor

EngBlaze
on **November 26, 2012 at 10:44 pm** said:

You might find something to help you accomplish your goal here: http://arduino.cc/playground/Main/LibraryList#Timing. The best way to learn is to get familiar with existing code, modify it to make it your own, then eventually, build a custom solution if required.

David Roberts
on **November 14, 2012 at 8:56 pm** said:

Thanks for your help in advance. I am looking for a way of generating TTL outputs with a width of (precisely) 150 microseconds at frequencies ranging from 0 to 600 Hz. The frequencies would need to be adjusted by software every second or so. I have not bought an Arduino yet (so I don't even qualify as a newby) , but judging from the tutorial it seems that this is possible with an Ardunio Uno. Is this correct? Do you see any major hitches? – I don't want to go down this road needlessly. I was hoping to bypass the expense of doing this in LabView and a National Instruments board costing many hundreds of dollars.

EngBlaze
on **November 25, 2012 at 10:40 pm** said:

Hi David,

In general, there shouldn't be any gotchas in using Arduino digital I/O to generate TTL signals. You can find numerous examples of people doing this on the Arduino.cc forums or elsewhere on the web.

The only thing to make sure of is whether the Uno can handle the precision you're looking for. You say you need precisely 150 ms pulses… the Uno can certainly reliably output this interval but the margin of error is controlled by processor speed. If by precisely, you mean +/- several hundred microseconds, and your code won't be handling a lot of other things, you should be just fine. If you need precision on the order of a few microseconds, you'll have to investigate more closely, but for $30, the Uno is a cheap experiment compared to other hardware or spending hours in LabView.

Happy coding!

soulid
on **November 25, 2012 at 7:03 am** said:

Great tutorial…first time I understood something of that bit-moving into register.
I tried to extend the sample with a Timer1_COMPB_vect to switch off the LED at a certain time defined by it, but failed. I suspect the timer will be reseted after COMPA has been reached…is that right? How to disable that?

Here is the code:

```
// avr-libc library includes
#include
#include
```

```
#define LEDPIN 13
int timer;
void setup()
{
pinMode(LEDPIN, OUTPUT);
cli(); // disable global interrupts

// initialize Timer1
// ZERO Timer1
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:initial value changed in loop (LED ON)
OCR1B = 30000; // turn on CTC mode:initial value changed in loop (LED OFF)
TCCR1B |= (1 << WGM12); // Set CS10 and CS12 bits for 1024 prescaler:
TCCR1B |= (1 << CS10); // prescaler 1024
TCCR1B |= (1 << CS12); // prescaler 1024
TIMSK1 |= (1 << OCIE1A); // enable timer compare interrupt
TIMSK1 |= (1 << OCIE1B); // enable timer compare interrupt:
sei(); // Enable global interrupts
}

void loop()
{}

ISR(TIMER1_COMPA_vect){digitalWrite(LEDPIN, HIGH);}
ISR(TIMER1_COMPB_vect){digitalWrite(LEDPIN, LOW);}
```

Vuong Hong
on **December 13, 2012 at 11:45 am** said:

Hi, this is the best tutorial I've ever read.
I wonder whether we have to clear the TCCR1A and B and connect the RESET pin to Vcc for program to work properly?
Thanks.

EngBlaze
on **December 13, 2012 at 10:53 pm** said:

Hi Vuong,

TCCR1A and B control the timer mode and a few other options, so you shouldn't need to clear them for anything to work properly. Just make sure you set them to the mode you need, as mentioned in the "Configuring and running the timer" section of the tutorial.

BioZ
on **December 28, 2012 at 12:03 pm** said:

Hi, amazing tutorial. I am still a bit new, so i have an issue. I need to read 2 digital signals with a frequency of about 500KHz each via the interrupt pins (2 and 3). I have a 2560 mega. sampling is confusing me – i know i can set the timer to overflow using compare register at 500Khz, but how do i make the called interrupt read the pins? i keep thinking that it will have to be in phase for it to be read correctly, and that is what is something i am not sure how to do. Perhaps sample at a faster rate, like 1MHz and use interrupt changing, rising or falling, so it will definatley catch a pulse regardless of phase.
Best regards

> 
> **EngBlaze**
> on **January 13, 2013 at 9:40 pm** said:
>
> Timer interrupts and pin change interrupts are completely different beasts, so you should be picking the correct tool to solve your problem. If you're trying to read a signal and react differently depending on whether the pin is high or low, you'll want to use pin change interrupts. A timer interrupt would require you to be perfectly in phase with the signal, as you said, and executing the interrupt service routine itself would throw this off.
>
> If you're using a pin change interrupt, you don't need to set a rate – the ISR will trigger whenever the signal changes. You can track a counter variable to see how much time has elapsed since the last change and by extension, your pulse length.


Marcos Rodriguez
on **January 4, 2013 at 1:53 pm** said:

Hello I need help…
My problem is the following. I want to use the arduino uno to check an external signal. This signal is always high but it for some reason the signal goes low for more than 3us I need to know by making an led come on.
Thank you very much for you help.

> 
> **EngBlaze**
> on **January 5, 2013 at 11:16 am** said:
>
> Hi Marcos, in this case you probably want to trigger an interrupt when the signal first goes low, which then checks again to see if the signal remains low after 3us. You should be able to do this by adapting the example code, or check out the Arduino playground for more extensive examples: http://playground.arduino.cc/Code/Interrupts


jps1
on **April 30, 2014 at 12:10 am** said:

Thank you for logically and clearly explaining timer basics. Everything was so easy.
Question: will a pinchange interrupt still trigger even if the actual pinchange event happened while a timer interrupt is being run?

> 
> **EngBlaze**
> on **October 31, 2014 at 5:33 pm** said:
>
> Hi jps, glad you found it useful!

The answer to your question is a bit nuanced. AVR processors have two types of interrupts – one type sets an “interrupt flag”, which will be processed and cleared when the chip finishes its current ISR, while the other is processed immediately, but only if no other ISR is currently running. Arduino’s attachInterrupt() function only uses the latter – if an interrupt is triggered while another ISR is already running, it will be ignored (unless the new interrupt is still being triggered after the ISR completes).

If you wish to have the processor queue multiple ISRs, you’ll have to do so using the more advanced AVR methods. You can also change a configuration bit on your processor to enable “nested interrupts”, which means that any enabled interrupt will interrupt a currently running ISR. Check out the “Reset and Interrupt Handling” section of your processor’s datasheet for more detailed info.