Newbie's Guide to AVR Timers

Dean Camera

March 15, 2015

****

Contents

Chapter 1

Introduction

The timer systems on the AVR series of Microcontrollers are complex beasts. They have a myriad of uses ranging from simple delay intervals right up to complex Pulse Width Modulation (PWM) signal generation. However, despite the surface complexity, the function of the timer subsystem can be condensed into one obvious function: to measure time.

We use timers every day—the most simple one can be found on your wrist. A simple clock will time the seconds, minutes and hours elapsed in a given day—or in the case of a twelve hour clock, since the last half-day. AVR timers do a similar job, measuring a given time interval.

The AVR timers are very useful as they can run asynchronous to the main AVR core. This is a fancy way of saying that the timers are separate circuits on the AVR chip which can run independent of the main program, interacting via the control and count registers, and the timer interrupts. Timers can be configured to produce outputs directly to pre-determined pins, reducing the processing load on the AVR core.

One thing that trips those new to the AVR timer is the clock source. Like all digital systems, the timer requires a clock in order to function. As each clock pulse increments the timer's counter by one, the timer measures intervals in periods of one on the input frequency:

$$\text{Timer Resolution} = \frac{1}{\text{Input Frequency}}$$

This means the smallest amount of time the timer can measure is one period of the incoming clock signal. For instance, if we supply a 100Hz signal to a timer, our period becomes:

$$\begin{aligned} \text{Timer Resolution} &= \frac{1}{\text{Input Frequency}} \\ &= \frac{1}{100Hz} \\ &= .01s \end{aligned}$$

For the above example, our period becomes .01 seconds, thus our timer will be able to measure times that are a multiple of this duration. If we measure a delay to be 45 of our 100Hz timer periods, then our total delay will be 45 times .01 seconds, or .45 seconds.

For this tutorial, I will assume the target to be an ATMEGA16, running at at 1MHz clock. This is a nicely featured AVR containing certain timer functionality we'll need later on. As modern AVRs come running off their internal $\approx$ 1MHz RC oscillator by default, you can use this without a problem (although do keep in mind the resultant timing measurements will be slightly incorrect due to the RC frequency tolerance).

In the sections dealing with toggling a LED, it is assumed to be connected to PORTB, bit 0 of your chosen AVR (pin 1 of DIP AVRMEGA16).

To start off, we will deal with basic timer functionality and move on from there.

Chapter 2

Timers running at Fcpu

We'll start with a simple example. We'll create a simple program to flash a LED at about 10Hz. Simple, right? Making a LED blink at 10Hz requires us to both turn it on and off again, so we need to perform a toggle of the LED's output state every $\frac{1}{20}$th of a second.

First, let's look at the pseudocode required to drive this example:

```
Set up LED hardware
Set up timer

WHILE forever
  IF timer value IS EQUAL TO OR MORE THAN 1/20 sec THEN
    Reset counter
    Toggle LED
  END IF
END WHILE
```

We're just starting out, so we'll use the polled method of determining the elapsed time - we'll put in an IF statement in our code to check the current timer value, and act on it once it reaches (or exceeds) a certain value. Before we start on the timer stuff, let's create the skeleton of our project:

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

int main (void)
{
   // TODO: Set up LED hardware

   // TODO: Set up timer

   for (;;)
   {
      // TODO: Check timer value, reset and toggle LED when count matches 1/20 of a second
   }
}
```

Extremely simple. I'm going to assume you are familiar with the basics of setting up AVR ports as well as bit manipulation (if you're uncertain about the latter, refer to this excellent tutorial). With that in mind, I'll add in the LED-related code and add in the IF statement:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   // TODO: Set up timer

   for (;;)
   {
      // TODO: Check timer value in if statement, true when count matches 1/20 of a second
      if ()
      {
         PORTB ^= (1 << 0); // Toggle the LED
```

```
            // TODO: Reset timer value
        }
    }
}
```

Now we need to start dealing with the timer. We want to do nothing more than start it at 1MHz, then check its value later on to see how much time has elapsed. We need to deviate for a second and learn a little about how the timer works in its most basic mode.

The AVR timer circuits come in two different widths, 8 and 16 bit. While the capabilities of the two timer types differ, at the most basic level (simple counting), the only difference is the maximum amount of time the timer can count to before overflowing and resetting back to zero. Those familiar with C will know that an unsigned eight bit value can store a value from 0 to $2^8 - 1$, or 255, before running out of bits to use and becoming zero again. Similarly, an unsigned 16 bit value may store a value from 0 to $2^{16} - 1$, or 65535 before doing the same.

As the name suggests, an 8 bit timer stores its value as an eight bit value in its count register, while the 16 bit timer stores its current count value in a pair of eight bit registers. Each advancement of the counter register for any AVR timer indicates that one timer period has elapsed.

Our project needs a fairly long delay, of $\frac{1}{20}$ of a second. That's quite short to us humans, but to a microcontroller capable of millions of instructions per second it's a long time indeed!

Our timer will be running at the same clock speed as the AVR core to start with, so we know that the frequency is 1MHz. One Megahertz is $\frac{1}{1000000}$ of a second, so for each clock of the timer only one millionth of a second has elapsed! Our target is $\frac{1}{20}$ of a second, so let's calculate the number of timer periods needed to reach this delay:

$$\begin{aligned}
\text{Target Timer Count} &= \frac{1}{\text{Target Frequency}} / \frac{1}{\text{Timer Clock Frequency}} - 1 \\
&= \frac{1}{20} / \frac{1}{1000000} - 1 \\
&= \frac{.05}{0.000001} - 1 \\
&= 50000 - 1 \\
&= 49999
\end{aligned}$$

The AVR timers will only update their count on each timer input clock tick, thus it takes one tick to get from a count of zero to one, or from the timer's maximum value back to zero. As a result, we need to decrement the number of ticks in our calulation by one as shown in the above formula.

So running at 1MHz, our timer needs to count to 49999 before $\frac{1}{20}$th of a second has elapsed. That's a very large value - too large for an 8 bit value! We'll need to use the 16 bit timer 1 instead.

Firstly, we need to start the timer at the top of our main routine, so that it will start counting. To do this, we need to supply the timer with a clock; as soon as it is clocked it will begin counting in parallel with the AVR's CPU core (this is called synchronous operation). To supply a clock of F_{cpu} to the timer 1 circuits we need to set the CS10 bit (which selects a F_{cpu} prescale of 1 - more on that later) in the `TCCR1B`, the Timer 1 Control Register B.

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

int main (void)
{
    DDRB |= (1 << 0); // Set LED as output

    TCCR1B |= (1 << CS10); // Set up timer

    for (;;)
    {
        // TODO: Check timer value in if statement, true when count matches 1/20 of a second
```

```
      if ()
      {
         PORTB ^= (1 << 0); // Toggle the LED

         // TODO: Reset timer value
      }
   }
}
```

Now, with only one line of code, we've started the hardware timer 1 counting at 1MHz - the same speed as our AVR. It will now happily continue counting independently of our AVR. However, at the moment it isn't very useful, we still need to do something with it!

We want to check the timer's counter value to see if it reaches $\frac{1}{20}$ of a second, or a value of 49999 at 1MHz as we previously calculated. The current timer value for timer 1 is available in the special 16-bit register, `TCNT1`. In actual fact, the value is in two 8-bit pair registers `TCNT1H` (for the high byte) and `TCNT1L` (for the low byte), however the C library implementation we're using helpfully hides this fact from us.

Let's now add in our check to our code - it's as simple as testing the value of `TCNT1` and comparing against our wanted value, 49999. To prevent against missed compares (where the timer updates twice between checks so our code never sees the correct value), we use the equal to or more than operator, "`>=`".

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= (1 << CS10); // Set up timer

   for (;;)
   {
      // Check timer value in if statement, true when count matches 1/20 of a second
      if (TCNT1 >= 49999)
      {
         PORTB ^= (1 << 0); // Toggle the LED

         // TODO: Reset timer value
      }
   }
}
```

Great! We've only got one more line of code to write, to reset the timer value. We already know the current value is accessed via the `TCNT1` register for Timer 1, and since this is a read/write register, we can just write the value 0 to it once our required value is reached to rest it.

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= (1 << CS10); // Set up timer

   for (;;)
   {
      // Check timer value in if statement, true when count matches 1/20 of a second
      if (TCNT1 >= 49999)
      {
         PORTB ^= (1 << 0); // Toggle the LED

         TCNT1 = 0; // Reset timer value
      }
   }
}
```

And there we have it! We've just created a very basic program that will toggle our LED every $\frac{1}{20}$ of a second at a 1MHz clock. Testing it out on physical hardware should show the LED being dimmer than normal (due to it being pulsed quickly). Good eyesight might reveal the LED's very fast flickering.

Next, we'll learn about the prescaler so we can try to slow things down a bit.

Chapter 3

Prescaled Timers

In chapter 2 of this tutorial we learned how to set up our 16-bit timer 1 for a $\frac{1}{20}$ second delay. This short (to us humans) delay is actually quite long to our AVR - 50,000 cycles in fact at 1MHz. Notice that Timer 1 can only hold a value of 0-65535 - and we've almost reached that! What do we do if we want a longer delay?

One of the easiest things we can do is to use the timer's prescaler, to trade resolution for duration. The timer prescaler is a piece of timer circuitry which allows us to divide up the incoming clock signal by a power of 2, reducing the resolution (as we can only count in 2^n cycle blocks) but giving us a longer timer range.

Let's try to prescale down our F_{cpu} clock so we can reduce the timer value and reduce our delay down to a nice 1Hz. The Timer 1 prescaler on the ATMEGA16 has divide values of 1, 8, 64, 256 and 1024 - so let's re-do our calculations and see if we can find an exact value.

Our calculations for our new timer value are exactly the same as before, except we now have a new prescaler term. We'll look at our minimum resolution for each first:

$$\begin{aligned} \text{Timer Resolution} &= \frac{1}{\text{Input Frequency/Prescale}} \\ &= \frac{\text{Prescale}}{\text{Input Frequency}} \end{aligned}$$

For a 1MHz clock, we can construct a table of resolutions using the available prescaler values and a F_{cpu} of 1MHz.

Prescaler Value	Resolution @ 1 MHz
1	1 µs
8	8 µs
64	64 µs
256	256 µs
1024	1024 µs

If you recall our equation for calculating the timer value for a particular delay in chapter 2 of this tutorial, you'll remember it is the following:

$$\text{Target Timer Count} = \frac{1}{\text{Target Frequency}} / \frac{1}{\text{Timer Clock Frequency}} - 1$$

However, as we've just altered the prescaler term, the latter half is now different. Substituting in our new resolution equation from above we get:

$$\text{Target Timer Count} = (\frac{1}{\text{Target Frequency}} / \frac{\text{Prescale}}{\text{Input Frequency}}) - 1$$

Or, rearranged:

$$\text{Target Timer Count} = (\frac{\text{Input Frequency}}{\text{Prescale} \times \text{Target Frequency}}) - 1$$

Now, we want to see if there is a prescaler value which will give an *exact* delay of 1Hz. One Hertz is equal to one cycle per second, so we want our compare value to be one second long, or 1000000uS. Let's divide that by each of our resolutions and put the results in a different table:

Prescaler Value	Target Timer Count
1	999999
8	124999
64	15624
256	3905.25
1024	975.5625

The results are interesting. Of the available prescaler values, we can immediately discount 256 and 1024 - they do not evenly divide into our wanted delay period. They are of course usable, but due to the rounding of the timer count value the resultant delay will be slightly over or under our needed delay. That leaves us with three possible prescaler values that will suit our purposes; 1, 8 and 64.

Our next task is to remove the values that aren't possible given our hardware constraints. On an 8-bit timer, that means discounting values of more than $2^8 - 1$, or 255, as the value won't fit into the timer's 8-bit count register. For our 16-bit timer, we have a larger range of 0 to $2^{16} - 1$, or 65535. Only one of our prescaler values satisfies this requirement — a prescale of 64 — as the other two possibilities require a timer count value of more bits than our largest 16-bit timer is capable of storing.

Let's go back to our original timer program and modify it to compare against our new value of 15624, which we've found to be 1 second at a prescale of 64 and a F_{cpu} of 1MHz:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   // TODO: Set up timer at Fcpu/64

   for (;;)
   {
      // Check timer value in if statement, true when count matches 1 second
      if (TCNT1 >= 15624)
      {
         PORTB ^= (1 << 0); // Toggle the LED

         TCNT1 = 0; // Reset timer value
      }
   }
}
```

Note I've removed the timer setup line, as it is no longer valid. We want to set up our timer to run at $\frac{1}{F_{cpu} \times 64}$ now. To do this, we need to look at the datasheet of the ATMEGA16 to see which bits need to be set in which control registers.

Checking indicates that we need to set both the `CS10` and `CS11` prescaler bits in `TCCR1B`, so let's add that to our program:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Set up timer at Fcpu/64

   for (;;)
   {
      // Check timer value in if statement, true when count matches 1 second
      if (TCNT1 >= 15624)
      {
         PORTB ^= (1 << 0); // Toggle the LED

         TCNT1 = 0; // Reset timer value
      }
   }
}
```

Compile it, and we're done! Remember that our timer runs as soon as it gets a clock source, so with the above configuration our program will now work and the LED will now flash. The LED is toggled at a rate of 1Hz via our timer code, and two toggles are required for a full on-off cycle, giving a flashing frequency of .5Hz.

Chapter 4

Long Timer Delays in Firmware

So far, we've learned how to use the timers in their most basic counting mode to delay a specified duration. However, we've also discovered a limitation of the timers — their maximum duration that their timer count registers can hold. We've managed to get a 1Hz delay out of a prescaled 16-bit timer with a prescale, but what if we want a delay of a minute? An hour? A week or year?

The answer is to create a sort of prescaler of our own in software. By making the hardware timer count to a known delay — say the 1Hz we created earlier — we can increment a variable each time that period is reached, and only act after the counter is reached a certain value. Let's pseudocode this so we can get a better understanding of what we want to do:

```
Set up LED hardware
Set up timer
Initialise counter to 0

WHILE forever
  IF timer value IS EQUAL TO 1 sec THEN
    Increment counter
    Reset timer
    IF counter value IS EQUAL TO 60 seconds THEN
      Toggle LED
    END IF
  END IF
END WHILE
```

The above pseudocode will build on our last experiment - a timer with a one second count - to produce a long delay of one minute (60 seconds). It's very simple to implement - all we need extra to our last example is an extra IF statement, and a few variable-related lines. First off, we'll re-cap with our complete code as it stands at the moment:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Set up timer at Fcpu/64

   for (;;)
   {
      // Check timer value in if statement, true when count matches 1 second
      if (TCNT1 >= 15624)
      {
         PORTB ^= (1 << 0); // Toggle the LED

         TCNT1 = 0; // Reset timer value
      }
   }
}
```

We need some code to create and initialise a new counter variable to 0, then increment it when the counter reaches one second as our pseudocode states. We also need to add in a test to see if our new variable reaches the value of 60, indicating that one minute has elapsed.

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

int main (void)
{
   // TODO: Initialise a new counter variable to zero

   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Set up timer at Fcpu/64

   for (;;)
   {
      // Check timer value in if statement, true when count matches 1 second
      if (TCNT1 >= 15625)
      {
         TCNT1 = 0; // Reset timer value
         // TODO: Increment counter variable

         // TODO: Check here to see if new counter variable has reached 60
         if ()
         {
            // TODO: Reset counter variable

            PORTB ^= (1 << 0); // Toggle the LED
         }
      }
   }
}
```

Now that we have our new program's structure, replacing the `TODO`s becomes very simple. We want a target count of 60, which is well within the range of an 8-bit unsigned char variable, so we'll make our counter variable of type `unsigned char` to give us a range of up to 255 ($2^8 - 1$) seconds. The rest of the code is fairly straightforward, so I'll add it all in at once:

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

int main (void)
{
   unsigned char ElapsedSeconds = 0; // Make a new counter variable and initialise to zero

   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Set up timer at Fcpu/64

   for (;;)
   {
      // Check timer value in if statement, true when count matches 1 second
      if (TCNT1 >= 15624)
      {
         TCNT1 = 0; // Reset timer value
         ElapsedSeconds++;

         if (ElapsedSeconds == 60) // Check if one minute has elapsed
         {
            ElapsedSeconds = 0; // Reset counter variable

            PORTB ^= (1 << 0); // Toggle the LED
         }
      }
   }
}
```

Compile and run, and the LED should toggle once per minute. By extending this technique, we can produce delays of an arbitrary duration. One point of interest is to note that any timing errors compound - so if the timer input frequency is 1.1MHz rather than 1.0MHz our one minute timer will be sixty times that small error out in duration. For this reason it is important to ensure that the timer's clock is as accurate as possible, to reduce long-term errors as much as possible.

Chapter 5

The CTC Timer Mode

Up until now, we've been dealing with the timers in a very basic way - starting them counting, then comparing in our main routine against a wanted value. This is rather inefficient - we waste cycles checking the timer's value every time the loop runs, and slightly inaccurate (as the timer may pass our wanted compare value slightly while processing the loop). What if there was a better way?

Well, there is. The AVR timers usually incorporate a special function mode called "Clear Timer on Compare", or CTC for short. The CTC operating mode does in hardware what we've previously experimented in software; it compares in hardware the current timer value against the wanted value, and when the wanted value is reached a flag in a status register is set and the timer's value reset.

This is extremely handy; because the comparing is done in hardware, all we have to worry about is checking the flag to determine when to execute our LED toggling - much faster than comparing bytes or (in the case of the 16-bit timer) several bytes.

CTC mode is very straightforward. Before we look into the implementation, let's pseudocode what we want to do.

```
Set up LED hardware
Set up timer in CTC mode
Set timer compare value to one second

WHILE forever
  IF CTC flag IS EQUAL TO 1 THEN
    Toggle LED
    Clear CTC flag
  END IF
END WHILE
```

Short and to the point. Note that the name of the mode is *Clear Timer on Compare* - the timer's value will automatically reset each time the compare value is reached, so we only need to clear the flag when the delay is reached. This set-and-forget system is very handy, as once the timer is configured and started we don't need to do anything other than check and clear its status registers.

Now then, we'll grab our previous example, modifying it to fit with our new pseudocode:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   // TODO: Configure timer mode to CTC
   // TODO: Set compare value for a compare rate of 1Hz

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

   for (;;)
   {
      if () // TODO: Check CTC flag
```

```
        {
            PORTB ^= (1 << 0); // Toggle the LED

            // TODO: Clear CTC flag
        }
    }
}
```

Now, we need to flesh out the skeleton code we have. First up we need to configure our timer for CTC mode. As you might be able to guess, we want to configure our timer, thus the bits we want will be located in the timer's Control registers. The table to look for is the one titled *"Waveform Generation Mode Bit Description"*, and is located in the timer control register descriptions for each timer. This table indicates all the possible timer modes, the bits required to set the timer to use those modes, and the conditions each mode reacts to.

You should note that our previous examples have ignored this table altogether, allowing it to use its default value of all mode bits set to zero. Looking at the table we can see that this setup corresponds to the "Normal" timer mode. We want to use the CTC mode of the timer, so let's look for a combination of control bits that will give us this mode.

Interestingly, it seems that two different combinations in Timer 1 of the ATMEGA16 will give us the same CTC behaviour we desire. Looking to the right of the table, we can see that the `TOP` value (that is, the maximum timer value for the mode, which corresponds to the compare value in CTC mode) uses different registers for each. Both modes behave in the same manner for our purposes and differ only by the register used to store the compare value, so we'll go with the first.

The table says that for this mode, only bit `WGM12` needs to be set. It also says that the register used for the compare value is named `OCR1A`.

Looking at the timer control registers (`TCCR1A` and `TCCR1B`) you should notice that the `WGM1x` bits - used to configure the timer's mode - are spread out over both registers. This is a small pain as you need to find out which bits are in which register, but once found setting up the timer becomes very easy. In fact, as we only have one bit to set — `WGM12` — our task is even easier. The ATMEGA16's datasheet says that `WGM12` is located in the `TCCR1B` register, so we need to set that.

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode
   // TODO: Set compare value for a compare rate of 1Hz

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

   for (;;)
   {
      if () // TODO: Check CTC flag
      {
          PORTB ^= (1 << 0); // Toggle the LED

          // TODO: Clear CTC flag
      }
   }
}
```

The second task for this experiment is to set the compare value - the value that will reset the timer and set the CTC flag when reached by the timer. We know from the datasheet that the register for this is `OCR1A` for the MEGA16 in the first CTC timer mode, so all we need is a compare value. From our previous experiment we calculated that 1Hz at 1MHz with a prescaler of 64 needs a compare value of 15624, so let's go with that.

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode
   OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

   for (;;)
   {
      if () // TODO: Check CTC flag
      {
          PORTB ^= (1 << 0); // Toggle the LED

          // TODO: Clear CTC flag
      }
   }
}
```

There, almost done already! Last thing we need is a way of checking to see if the compare has occurred, and a way to clear the flag once its been set. The place to look for the compare flags is in the timer's Interrupt Flag register — an odd place it seems, but the reason will become clear in the next section dealing with timer interrupts. The ATMEGA16's Timer 1 interrupt flags are located in the combined register `TIFR`, and the flag we are interested in is the "Output Compare A Match" flag, `OCF1A`. Note the "A" on the end; Timer 1 on the ATMEGA16 has two CTC channels (named channel A and channel B), which can work independently. We're only using channel A for this experiment.

Checking for a CTC event involves checking the `OCF1A` flag in this register. That's easy - but what about clearing it? The datasheet includes an interesting note on the subject:

> *...OCF1A can be cleared by writing a logic 1 to its bit location*

Very strange indeed! In order to clear the CTC flag, we actually need to set it - even though it's already set. Due to some magic circuitry inside the AVR, writing a 1 to the flag when its set will actually cause it to clear itself. This is an interesting behaviour, and is the same across all the interrupt bits. There's a good reason for the flags to be cleared this way, although it is a little outside the scope of this tutorial.

Despite that, we can now add in our last lines of code to get a working example:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode
   OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

   for (;;)
   {
      if (TIFR & (1 << OCF1A))
      {
          PORTB ^= (1 << 0); // Toggle the LED

          TIFR = (1 << OCF1A); // clear the CTC flag (writing a logic one to the set flag
              clears it)
      }
   }
}
```

And there we have it, a working .5Hz LED flasher using the CTC timer mode!

Chapter 6

CTC Mode using Interrupts

For all our previous experiments, we've been using a looped test in our main code to determine when to execute the timer action code. That's fine, but what if we want to shift the responsibility of choosing when to execute the timer code to the AVR hardware instead? To do this, we need to look at the timer interrupts.

Interrupts are events that when enabled, cause the AVR to execute a special routine (called an Interrupt Service Routine, or ISR for short) when the interrupt conditions are met. These interrupts can happen at any time in the program's execution when global interrupts are enabled. When an enabled ISR's condition is met, the main routine is paused while the ISR executes. Once the ISR execution completes, the main routine is resumed until the next interrupt. This is useful for us, as it means we can eliminate the need to keep checking the timer value and just respond to its interrupt events instead.

The AVR timers can have several different Interrupts - typically Overflow, Compare and Capture. Overflow occurs when the timer's value rolls past its maximum and back to zero (for an 8 bit timer, that's when it counts past 11111111 in binary and resets back to 00000000). However, for this section we'll deal with the Compare interrupt, which occurs in CTC mode when the compare value is reached.

Again, we'll pseudocode this to start with:

```
Set up LED hardware
Set up timer in CTC mode
Enable CTC interrupt
Enable global interrupts
Set timer compare value to one second

WHILE forever
END WHILE

ISR Timer Compare
   Toggle LED
END ISR
```

We can start off this by working with our skeleton main code, used in previous examples. I'll skip the details on the parts already discussed in previous sections.

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode

   // TODO: Enable CTC interrupt
   // TODO: Enable global interrupts

   OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64
```

```
    for (;;)
    {

    }
}

// TODO: Add compare ISR here
```

Note how it's a modified version of the non-interrupt driven CTC example covered in the last section. All we need to do is tell the timer to run the compare ISR we define when it counts up to our compare value, rather then us polling the compare match flag in our main routine loop.

We'll start with creating the ISR first, as that's quite simple. In AVR-GCC — specifically, the avr-libc Standard C Library that comes with it — the header file for dealing with interrupts is called (unsurprisingly) "interrupt.h" and is located in the `avr` subdirectory. We need to include this at the top of our program underneath our include to the IO header file. The top of our code should look like this:

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

int main (void)
{
  ...
```

This gives us access to the API for dealing with interrupts. We want to create an ISR for the Timer 1 Compare Match event. The syntax for defining an ISR body in AVRGCC is:

```
ISR(VectorName_vect)
{
    // Code to execute on ISR fire here
}
```

Where *VectorName* is the name of the ISR vector which our defined ISR handles. The place to go to find this name is the "Interrupt" section of the datasheet, which lists the symbolic names for all the ISR vectors that the chosen AVR supports. When writing the vector name into GCC, replace all spaces with underscores, and append "`_vect`" to the end of the vector's name.

Like in chapter 5 we are still dealing with Channel A Compare of Timer 1, so we want the vector named "TIMER1 COMPA". In GCC this is called "`TIMER1_COMPA_vect`", after performing the transformations outlined in the last paragraph. Once the ISR is defined, we can go ahead and write out it's body, adding the LED toggling code.

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

int main (void)
{
    DDRB |= (1 << 0); // Set LED as output

    TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode

    // TODO: Enable CTC interrupt
    // TODO: Enable global interrupts

    OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

    TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

    for (;;)
    {

    }
}
```

```
ISR(TIMER1_COMPA_vect)
{
   PORTB ^= (1 << 0); // Toggle the LED
}
```

Notice how we don't clear the CTC event flag like in chapter 5 — this is automatically cleared by the AVR hardware once the ISR fires. Neat, isn't it!

Running the code so far won't yield any results. This is because although we have our ISR all ready to handle the CTC event, we haven't enabled it! We need to do two things; enable the "TIMER1 COMPA" interrupt specifically, and turn on interrupt handling on our AVR.

The way to turn on our specific interrupt is to look into the second interrupt-related register for our timer, `TIMSK`. This is the Timer Interrupt Mask register, which turns on and off ISRs to handle specific timer events. Note that on the ATMEGA16 this single register contains the enable bits for all the timer interrupts for all the available timers. We're only interested in the Timer 1 Compare A Match interrupt enable bit, which we can see listed as being called `OCIE1A` (Output Compare Interrupt Enable, channel A).

By setting that bit we instruct the timer to execute our ISR upon compare match with our specified compare value. Let's put that line into our program's code and see how it all looks.

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode

   TIMSK |= (1 << OCIE1A); // Enable CTC interrupt

   // TODO: Enable global interrupts

   OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

   for (;;)
   {

   }
}

ISR(TIMER1_COMPA_vect)
{
   PORTB ^= (1 << 0); // Toggle the LED
}
```

Only one more thing to do — enable global interrupts. The AVR microcontrollers have a single control bit which turns on and off interrupt handling functionality. This is used in pieces of code where interrupt handling is not desired, or to disable interrupts while an ISR is already being executed. The latter is done automatically for us, so all we need to do is turn on the bit at the start of our code, and our compare interrupt will start to work.

The command to do this is called `sei()` in the avr-libc library that ships with WinAVR, and is named to correspond with the assembly instruction which does the same for AVRs (the SEI instruction). That's irrelevant however, as we just need to call the command in our code.

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

int main (void)
{
```

```
    DDRB |= (1 << 0); // Set LED as output

    TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode

    TIMSK |= (1 << OCIE1A); // Enable CTC interrupt

    sei(); //  Enable global interrupts

    OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

    TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

    for (;;)
    {

    }
}

ISR(TIMER1_COMPA_vect)
{
    PORTB ^= (1 << 0); // Toggle the LED
}
```

And our example is finished! Running this will give a nice .5Hz LED flasher using the timer's event interrupts. The nice thing is that the timer operation is now completely handled for us in hardware — once set up, we just need to react to the events we've configured. Notice that our main loop is now empty; if this is the case you may put sleep commands inside the main loop to save power between compares.

Chapter 7

Pure Hardware CTC

You probably think by now that we've improved our example as much as possible — after all, what more improvements are there to make? Well, it's time to finish of the CTC topic by looking at the hardware outputs.

All AVR's pins have alternative hardware functions. These functions (currently non re-routable) when activated interface the IO pins directly to the AVR's internal hardware - for instance the `Tx/Rx` alternative functions which are the direct interface to the AVR's USART subsystem. Alternative pin functions can be very useful; as they can be internally connected straight to a hardware subsystem, the maximum possible performance can be achieved.

In this section, we'll be looking at the Compare Output settings of the AVR timer.

Looking at the timer 1 control registers, we can see a few pairs of bits we've previously ignored, called (for timer 1) `COM1A1/COM1A0` and `COM1B1/COM1B0`. Bonus points to anyone who's linked the "A" and "B" parts of the bit names to the timer compare channels — you're spot on.

These bits allow us to control the hardware behaviour when a compare occurs. Instead of firing an interrupt, the hardware can be configured to set, clear or toggle the `OCxy` (where "x" is the timer number, "y" is the channel letter for timers with more than one channel) hardware pins when a compare occurs. We can use the toggle function with our LED flasher, so that the hardware toggles the LED's state for us automatically, making it a true set-and-forget operation.

Before we do anything else, let's work out which pins of our ATMEGA16 are linked to the Compare Output hardware — we want the pins with alternative functions starting with "`OC`". On our PDIP package version, that maps to:

GPIO Pin	Alternative Function
PB3	OC0
PD4	OC1B
PD5	OC1A

So timer 0 has one Compare Output channel, while timer 1 has two (channels A and B) as we've already discovered. As always we'll just deal with Channel A in our example.

Now we have a problem. All the previous chapters have assumed the LED is attached to `PORTB`, bit 0 - but we'll have to move it for this chapter. As stated above, the alternative pin functions cannot be moved to another pin, so we must move Moses...I mean, our LED, to the pin with the required alternative function.

Timer 1 Channel A's Compare Output is located on `PD5`, so move the LED there for the rest of this example. Now, let's psudocode:

```
Set up LED hardware
Set up timer in CTC mode
Enable timer 1 Compare Output channel A in toggle mode
Set timer compare value to one second
```

```
WHILE forever
END WHILE
```

Amazing how simple it is, isn't it! Well, we can already fill in almost all of this:

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

int main (void)
{
   DDRD |= (1 << 5); // Set LED as output

   TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode

   // TODO: Enable timer 1 Compare Output channel A in toggle mode

   OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

   for (;;)
   {

   }
}
```

All we need is to configure the timer so that it'll toggle our channel A output each time the timer value is equal to our compare value. The datasheet has several descriptions for the functionality of the `COM1Ax` and `COM1Bx` bits, so we need to find the table corresponding to the mode we're using the timer in.

A CTC mode isn't listed explictly; instead the appropriate table is listed as *"Compare Output mode, Non PWM"*. PWM stands for "**P**ulse **W**idth **M**odulation", and is a technique for producing digital output signals that can be modified with external components to produce pseudo-analog waveforms. For now, it is sufficient to know that the CTC mode is not a form of PWM and thus the non-PWM bit description table is the one we're looking for.

To make the channel A Compare Output pin toggle on each compare, the datasheet says we need to set bit `COM1A0` in `TCCR1A`. That's our missing line, so let's add it in!

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

int main (void)
{
   DDRD |= (1 << 5); // Set LED as output

   TCCR1B |= (1 << WGM12); // Configure timer 1 for CTC mode

   TCCR1A |= (1 << COM1A0); // Enable timer 1 Compare Output channel A in toggle mode

   OCR1A   = 15624; // Set CTC compare value to 1Hz at 1MHz AVR clock, with a prescaler of 64

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Start timer at Fcpu/64

   for (;;)
   {

   }
}
```

Simple, isn't it! We've now created the simplest (code-wise) LED flasher possible using pure hardware functionality. Running this will cause the LED to flash at .5Hz, without any code other than the timer initialization!

Chapter 8

The Overflow Event

Well, now that we've had fun creating a LED flasher via a variety of software and hardware CTC methods, we'll move on to one last LED flashing program. This time we'll be using a different Timer event to manage the toggling of the LED: the overflow.

As previously stated, timers store their values into internal 8 or 16 bit registers, depending on the size of the timer being used. These registers can only store a finite number of values, resulting in the need to manage the timer (via prescaling, software extension, etc) so that the interval to be measured fits within the range of the chosen timer.

However, what has not been discussed yet is what happens when the range of the timer is exceeded. Does the AVR explode? Does the application crash? Does the timer automatically stop?

The answer is simple, if rather boring. In the event of the timer register exceeding its capacity, it will automatically roll around back to zero and keep counting. When this occurs, we say that the timer has "overflowed".

When an overflow occurs, a bit is set in one of the timer status registers to indicate to the main application that the event has occurred. Just like with the CTC hardware, there is also a corresponding bit which can enable an interrupt to be fired each time the timer resets back to zero.

So why would we need the overflow interrupt? Well, I leave that as an exercise to the reader. However, we can demonstrate it here in this tutorial — via another LED flasher, of course!

Calculating the frequency of the flashing is a little different to our previous examples, as now we have to calculate in reverse (to find the frequency from the timer count and timer resolution rather than the timer count from a known frequency and timer resolution). We'll still be working with our 16-bit timer 1 for this example, to be consistent with previous chapters.

Let's go back to our one of the timer equations we used back in chapter 3:

$$\text{Target Timer Count} = (\frac{\text{Input Frequency}}{\text{Prescale} \times \text{Target Frequency}}) - 1$$

We want to determine the Target Frequency from the other two variables, so let's rearrange:

$$(\text{Target Timer Count} + 1) \times \frac{\text{Prescale}}{\text{Input Frequency}} = \frac{1}{\text{Target Frequency}}$$

And swap the left and right hand sides to get it into a conventional form:

$$\frac{1}{\text{Target Frequency}} = (\text{Target Timer Count} + 1) \times \frac{\text{Prescale}}{\text{Input Frequency}}$$

Since we know that for the overflow equation, the "Target Timer Count" becomes the maximum value that can be held by the timer's count register, plus one (as the overflow occurs after the

count rolls over from the maximum back to zero). The formula for the maximum value that can be held in a number of bits is:

$$\text{Max Value} = 2^{\text{Bits}} - 1$$

But we want one more than that to get the number of timer counts until an overflow occurs:

$$\text{Counts Until Overflow} = 2^{\text{Bits}}$$

Change "Max Value" to the more appropriate "Target Timer Count" in the first timer equation:

$$\frac{1}{\text{Target Frequency}} = (\text{Target Timer Count} + 1) \times \frac{\text{Prescale}}{\text{Input Frequency}}$$

And substitute in the formula for the counts until overflow to get the timer period equation. Since frequency is just the inverse of period, we can also work out the frequencies of each duration as well:

$$\text{Target Period} = \frac{1}{\text{Target Frequency}}$$

$$\text{Target Period} = 2^{\text{Bits}} \times \frac{\text{Prescale}}{\text{Input Frequency}}$$

Which is a bit complex, but such is life. Now's the fun part - we can now work out the overflow frequencies and periods for our 16-bit Timer 1 running at different prescales of our AVR's 1MHz system clock:

$$\text{Target Period} = 2^{16} \times \frac{\text{Prescale}}{\text{Input Frequency}}$$

Prescaler Value	Overflow Frequency	Overflow Period
1	15.259 Hz	65.5 ms
8	1.907 Hz	0.524 s
64	.2384 Hz	4.195 s
256	.0596 Hz	16.78 s
1024	.0149 Hz	67.11 s

Note how our frequency decreases (and period increases) as our prescaler increases, as it should. Because we have a reasonably slow main system clock, and a large timer count register, we end up with frequencies that are easy to see with the naked eye (with the exception of the case where no prescaler is used). Unlike the CTC method however, we are limited to the frequencies above and cannot change them short of using a smaller timer, different prescaler or different system clock speed — we lose the precision control that the CTC modes give us.

For this example, we'll use a prescaler of 8, to give a 1.8Hz flashing frequency, and a period of about half a second.

Almost time to get into the code implementation. But first, pseudocode! I'm going to extrapolate on the preceding chapters and jump straight into the ISR-powered example, rather than begin with a polled example. It works in the same manner as previous polled experiments, except for the testing of the overflow bit rather than the CTC bit.

```
Set up LED hardware
Set up timer overflow ISR
Start timer with a prescale of 8

WHILE forever
END WHILE

ISR Timer Overflow
   Toggle LED
END ISR
```

Let's get started. As always, we'll begin with the skeleton program:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   // TODO: Enable overflow interrupt
   // TODO: Enable global interrupts

   // TODO: Start timer at Fcpu/8

   for (;;)
   {

   }
}

// TODO: Add overflow ISR here
```

Let's begin with filling in the bits we can already do. The ISR code is easy; we can use the same ISR as chapter 6, except we'll be changing the compare vector to the overflow vector of timer 1.

Looking in the ATMEGA16 datasheet, the overflow interrupt for timer 1 is obvious — it's listed as "TIMER1 OVF" in the Interrupts chapter. Just like in chapter 6, we need to replace the spaces in the vector name with underscores, and add the "`_vect`" suffix to the end.

We can also fill in the "Enable global interrupts" line, as that is identical to previous chapters and is just the `sei()` command from the `<avr/interrupt.h>` header file.

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   // TODO: Enable overflow interrupt
   sei(); // Enable global interrupts

   // TODO: Start timer at Fcpu/8

   for (;;)
   {

   }
}

ISR(TIMER1_OVF_vect)
{
   PORTB ^= (1 << 0); // Toggle the LED
}
```

Next, we need to figure out how to enable the overflow vector, so that our ISR is run each timer the overflow occurs. The datasheet's 16-bit Timer/Counter section comes to our rescue again, indicating that it is the bit named `TOIE1` located in the Timer 1 Interrupt Mask register, `TIMSK`:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TIMSK |= (1 << TOIE1); // Enable overflow interrupt
   sei(); // Enable global interrupts

   // TODO: Start timer at Fcpu/8

   for (;;)
   {

   }
}

ISR(TIMER1_OVF_vect)
{
   PORTB ^= (1 << 0); // Toggle the LED
}
```

The last thing we need to do, is start the timer with a prescaler of 8. This should be easy for you to do — if not, refer back to chapter 3.

The ATMEGA16 Datasheet, Timer 1 section tells us that for a timer running with a prescaler of 8, we need to start it with the bit `CS11` set in the control register `TCCR1B`. Adding that to our code finishes our simple program:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TIMSK |= (1 << TOIE1); // Enable overflow interrupt
   sei(); // Enable global interrupts

   TCCR1B |= (1 << CS11); // Start timer at Fcpu/8

   for (;;)
   {

   }
}

ISR(TIMER1_OVF_vect)
{
   PORTB ^= (1 << 0); // Toggle the LED
}
```

All done! This simple program will cause the LED to toggle each time the overflow occurs and serves as a practical use of the overflow interrupt.

Chapter 9

Overflow as CTC

One neat application of the overflow event is for creating a CTC timer on AVRs which don't support true hardware CTC. It's not as neat as the pure hardware CTC discussed in chapter 7, but faster than the pure software CTC discussed in chapter 2.

CTC works by having a fixed `BOTTOM` value — that's the timer's minimum value — of zero, and a variable `TOP` value, the value at which resets the timer and fires the event. However, with the overflow event we seemingly have a fixed `BOTTOM` of again zero, and a fixed `TOP` of the maximum timer's value. Not so; using a small trick we can adjust the `BOTTOM` value to give us the equivalent of a CTC implementation standing on its head.

This technique is called timer reloading. When configured, we preload the timer's count register (which is both readable and writeable) with a value above zero. This shortens the time interval before the next overflow event, although only for a single overflow. We can get around that by again reloading the timer's value to our non-zero value inside the overflow event for a hybrid software/hardware CTC.

Pseudocode time!

```
Set up LED hardware
Set up timer overflow ISR
Load timer count register with a precalculated value
Start timer with a prescale of 8

WHILE forever
END WHILE

ISR Timer Overflow
   Toggle LED
   Reload timer count register with same precalculated value
END ISR
```

Let's examine the maths again. From chapter 3, we know that the formula for determining the number of timer clock cycles needed for a given delay is:

$$\text{Target Timer Count} = (\frac{\text{Input Frequency}}{\text{Prescale} \times \text{Target Frequency}}) - 1$$

Which works for a fixed `BOTTOM` of zero, and a variable `TOP`. We've got the "upside-down" implementation of a fixed `TOP` of 0 to $2^{\text{Bits}} - 1$ and a variable `BOTTOM`. Our `BOTTOM` value becomes the `TOP` minus the number of clock cycles needed for the equivelent CTC value, so the formula becomes:

$$\text{Reload Timer Value} = (2^{\text{Bits}} - 1) - (\frac{\text{Input Frequency}}{\text{Prescale} \times \text{Target Frequency}})$$

Let's go with the previous example in chapter 3: a .5Hz flasher, using a 1MHz clock and a prescale of 64. We found the timer count to be 15625 for those conditions. Plugging it into the above Reload Timer Value formula gives:

$$\begin{aligned} \text{Reload Timer Value} &= (2^{\text{Bits}} - 1) - 15625 \\ &= (2^{16} - 1) - 15625 \\ &= 65535 - 15625 \\ &= 49910 \end{aligned}$$

So we need to preload and reload our overflow timer on each overflow with the value 49910 to get our desired 1Hz delay. Since we've already gone over the code for an interrupt-driven overflow example in chapter 8, we'll build upon that.

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TIMSK |= (1 << TOIE1); // Enable overflow interrupt
   sei(); // Enable global interrupts

   // TODO: Preload timer with precalculated value

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Set up timer at Fcpu/64

   for (;;)
   {

   }
}

ISR(TIMER1_OVF_vect)
{
   PORTB ^= (1 << 0); // Toggle the LED
   // TODO: Reload timer with precalculated value
}
```

Both the reloading and preloading of the timer takes identical code - we just need to set the timer's count register to the precalculated value. The timer's current count value is available in the `TCNTx` register, where x is the timer's number. We're using the 16-bit timer 1, so the count value is located in `TCNT1`.

We'll now finish of the example, replacing the unfinished segments:

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

int main (void)
{
   DDRB |= (1 << 0); // Set LED as output

   TIMSK |= (1 << TOIE1); // Enable overflow interrupt
   sei(); // Enable global interrupts

   TCNT1 = 49910; // Preload timer with precalculated value

   TCCR1B |= ((1 << CS10) | (1 << CS11)); // Set up timer at Fcpu/64

   for (;;)
   {

   }
}

ISR(TIMER1_OVF_vect)
{
   PORTB ^= (1 << 0); // Toggle the LED
   TCNT1  = 49910; // Reload timer with precalculated value
}
```

And done! This is less preferable to the pure hardware CTC mode, as it requires a tiny bit more work on both the programmer and the AVR to function. However, it will work just fine for AVRs lacking the complete CTC timer functionality.