Using AVR Timer/Counters: Pulse Width Modulation Modes

Two excellent tutorials have already been written about the AVR timer/counters, and I want to fully acknowledge this excellent work. The tutorials are by abcminiuser (aka Dean Camera) on the this forum. It's not my purpose to steal any of Dean's thunder, since I know he plans to explain PWM and has already started that section of his tutorial. It seems to me that PWM is complex enough that it can bear explaining twice, and it seems that Dean's approach is to explain it from the audio and analog waveform creation viewpoint. My viewpoint will deal much more with motor and appliance control techniques with PWM.

While Dean's tutorials very thoroughly cover Normal and CTC Modes of the AVR timer/counters, I want to recap these modes briefly so we're all on the same page. The discussion may seem redundant, but will serve as a useful refresher and makes sure that all the basics are covered to provide the background needed for this tutorial.

We're going to be looking at some figures which will help explain the workings of PWM mode. For these figures to make sense, we'll develop them one step at a time from the simplest timer/counter modes to the more complicated. Some timer/counter basics need to be explained so we can build on them as well. I hope you find this approach will make the PWM modes easier to understand.

Note that everything Dean tells us about the Atmega16 timer/counter 1 applies to timer/counter 1 on the ATtiny2313. The only difference is the actual pin numbers of the output pins.

Let's get started!

Timer/Counter Basics

Let's start with a basic definition. What is a timer/counter? A *counter* is one of many registers in a modern microprocessor and simply counts events, that is, electrical pulses. Each time a pulse is applied to a counter, the value in the counter increments by one. If the pulses are applied at a known, constant rate, then the counter becomes a timer. Knowing the pulse rate and a count, we can easily compute a time value. For example, if the pulse rate is 100 pulses per second (each pulse is $1/100^{th}$ of a second long) and our counter has counted 20 pulses, then 0.20 seconds has elapsed. (20 pulses) / (100 pulses/second) = 0.20 seconds.

To understand timer/counters we must know their limits. There are two limits: the pulse properties (minimum pulse width and duration), and the maximum value that can be counted to. When using the internal clock sources (those provided by the processor), we don't have to worry about the minimum pulse properties. (However, these must be considered if we're driving the counter with an external signal. We'll just use the internal clocks and worry about external clocking another time.) The other limit, maximum count value, is determined by the width of the register in bits. An eight bit wide counter can

count to 2^8-1 or 255, while a 16 bit wide counter can count to 2^16-1 or 65,535. On the 256th count (or 65,536th count), the counter starts over at zero. We say that the counter rolls over, or overflows, when this happens.

If we don't stop the timer/counter before it reaches its maximum value, it overflows, or starts counting again from zero. Unless we keep track of each time an overflow occurs, we can't know the real time. Effectively, we extend the width of the counter register using software. Dean explains the method of doing this in his excellent tutorial.

More commonly, we don't really care to keep track of the total elapsed time. We just want an event to happen repeatedly at a known (programmable) interval. Timer/counters are admirably suited for doing this and most of this tutorial is about how to do exactly that.

Let's define a couple of more terms. **BOTTOM** is the value 0. This is the minimum value that the counter can have. Should be pretty obvious – can't count lower than this. **TOP** is the value the counter reaches when it starts over. In the simplest case, that value is the maximum value that can be in the counter register. This is determined by the width of the counter register in bits. An eight bit wide counter has the maximum value of 2^8 – 1 or 255. For a sixteen bit wide counter, the maximum value is 65,535. This value is also known as **MAX**. In other timer/counter modes, values other than MAX can be TOP and cause the counter to start over at zero, or *overflow*, before MAX is reached.

We can draw a useful diagram to help us understand timer/counter behavior. See Figure 1, below. Understanding this figure will help you understand the figures in the Atmel data sheets. I'll also use this type of figure as we explore more advanced capabilities of the timer/counters. **COUNT** is the value in the timer/counter register. Assuming that pulses are supplied at a constant rate, COUNT starts at BOTTOM and increases linearly, as **TIME** passes, to TOP, then starts over at BOTTOM. In the figure, increasing COUNT is shown as a ramp rising as TIME passes. MAX is shown for reference. In this figure, TOP and MAX are the same. The event when COUNT reaches TOP and starts over at BOTTOM is known as *overflow*. In the AVR processors, a special flag is set (the Timer Overflow Flag) when this occurs and an interrupt can be generated (if interrupts are enabled). This figure shows the AVR timer/counter in Normal Mode. Dean's tutorial provides several examples of using timer/counters in Normal Mode; I'm assuming you've all read this and understand it.

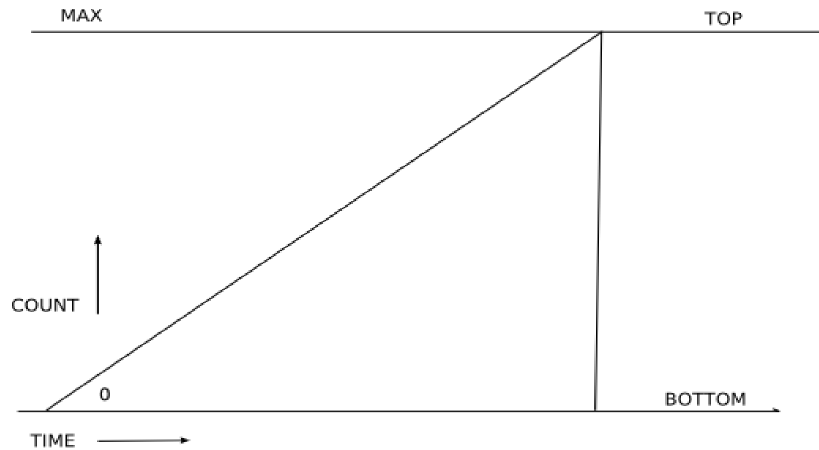


Figure 1 - Basic Timer Operation (Normal Mode)

Timer Control Registers

In addition to the register that holds COUNT (TCNT1 for timer/counter 1) and the register that holds the overflow flags (TIFR – shared between timer/counter 0 and timer/counter 1), there are additional registers that control the behavior of the timer/counters. These are introduced in Dean's tutorial, but I want to mention them since I'll discuss them in greater length as we go along. When appropriate, I'll use timer/counter 1 of the ATtiny2313 for specific examples.

First is the Timer Counter Interrupt Mask Register (TIMSK). This register is shared between timer/counter 0 and timer/counter 1 (as is TIFR) and controls the behavior of the processor when a flag bit is set in TIFR. To understand what's going on here, let's consider the overflow flag for timer/counter 1 (TOV1). TOV1 is in TIFR and is set whenever timer/counter 1 overflows. If Timer/Counter 1 Overflow Interrupt Enable (TOIE1) in TIMSK is set **and interrupts are enabled** (using the sei instruction), then an interrupt will be generated when timer/counter 1 overflows and sets TOV1. There are several other flags in TIFR controlled by corresponding bits in TIMSK. The events causing the TIFR flags to set vary, but the control by TIMSK is the same. We'll discuss these additional flags at the appropriate points in this tutorial.

The next registers we need to know about are the Timer/Counter Control Registers. For timer/counter 1, these registers are TCCR1A and TCCR1B. Not surprisingly, these registers control the behavior of the timer/counters by selecting modes, clock sources, and behavior of output bits. We'll focus on three sets of bits in the two registers. These will be sufficient for this tutorial. More detail will be given as we go through this tutorial – I just want introduce them now.

Five basic modes of operation may be selected for a timer/counter. These modes are Normal (the most basic mode), Clear Timer on Compare Match (CTC) Mode, Fast Pulse Width Modulation Mode, Phase Correct Pulse Width Modulation Mode, and Phase and

Frequency Correct Pulse Width Modulation Mode. Most of this tutorial will focus on the Pulse Width Modulation modes since the other two are very well covered in the Dean's tutorial. Each of these modes (except Normal) have variations. Including the variations brings the total number of modes to 15 (Mode 13 is Reserved). These modes are selected by appropriately setting bits WGM13, 12, 11, and 10 for timer/counter 1. WGM13 and WGM12 are in TCCR1B, while WGM11 and WGM10 are in TCCR1A.

Each timer/counter has three possible clock sources. Either an internal or external clock source may be selected. Selecting no clock source effectively turns off a timer/counter. (I consider this the third source selection.) The internal clock source may be chosen along with one of four dividers (prescalers) which we'll discuss below. We'll ignore the external clock source in this tutorial. The clock sources for timer/counter 1 are chosen using bits CS12, CS11, and CS10 in TCCR1B.

The third set of control bits in the control registers define the behavior of the output pins associated with each timer/counter. For timer/counter 1, these output bits are OC1A and OC1B. Control of OC1A is done with bits COM1A1 and COM1A0; control of OC1B is done by COM1B1 and COM1B0. These bits are in TCCR1A. We'll discuss all of these bits in much more detail as we progress through this tutorial.

Clock Sources and Clock Control

Before we really get into the timer/counter operational modes, let's first be sure we understand the clock as it applies to timer/counters, and how to control it. A source of pulses at a known rate (pulses per second) is commonly called a clock. The clock pulse rate is called the frequency and is measured in Hertz (abbreviated Hz). One Hertz is one cycle (one pulse) per second. This clock can be supplied by an AVR microprocessor – in fact several options exist for the clock.

In the Attiny2313 (and the Atmega168), the basic clock (the processor clock) is provided by an internal oscillator that runs at 8 MHz; that is eight million cycles per second. (Sorry for the basics here, you probably already know this stuff.) By default, this clock rate is divided by 8 so the processor clock is actually 1 MHz. The divide by 8 is done by a *prescaler* and other values are possible. On the ATtiny2313, nine different prescaler values are available (including 1, or no prescaling). All internal clocks for serial I/O, A/D conversion, and timer/counters are derived from the processor clock. If you change its speed, then all the other clocks will change speed also.

It’s quite easy to change the processor clock rate by selecting a different prescaler value. The process is described on page 30 of the ATtiny2313 data sheet and the possible clock divisors (prescalers) are in Table 12 on page 32. A two step process is required. First, set bit the Clock Prescaler Change Enable (CLKPCE) bit to one and all the other bits to 0. Then write the new value for the Clock Prescaler bits and clear the CLKPCE bit. Here's the code to change the clock to 8MHz (prescaler divider = 1). I've included this code in my example if you want to experiment with it.

```
CLKPR = _BV(CLKPCE);        // Initiate write cycle for clock setting
CLKPR = 0x00;               // 8 MHz clock out
```

One caution is that you must set the CLKPS bits within 4 cycles of setting CLKPCE. Usually this just means doing it as the next statement like in my example, but if you do this while interrupts are enabled, be sure to disable them while setting the Clock Prescaler. In older AVR processors, changing the clock prescaler must be done by changing fuse settings, but is still not difficult.

It's worth noting that the clock signal can be an output as well. On ATtiny2313, the clock output is available on CKOUT (pin 6) and can be enabled by setting the appropriate fuse. If you're using the typical Make file, this is the code you need.

```
#AVRDUDE_WRITE_FLASH += -U lfuse:w:0x24:m #8 Mhz clock, Div by 8, default start up, clk out enab
```

You don't need to do this for this tutorial. The code is in the Makefile for the examples, but is commented out. (That is, it has a # as the first character. Make interprets this as the start of a comment.) Here's the code to turn off this output if you want to restore the pin to its normal operation.

```
#AVRDUDE_WRITE_FLASH += -U lfuse:w:0x64:m #8 Mhz clock, Div by 8, default start up, clk out disab
```

Once you have done a Make with either of these lines of code, you don't need to use either line again unless you want to change the mode. That is, you uncomment the first line and do one Make to enable the clock signal output. You can then re-comment the line and the clock will stay enabled even if you do additional Makes. When you want to return the pin used for the clock output to its normal function, simply uncomment the second line shown above in your Makefile and do another Make. Normal operation is restored and you can re-comment the second line. It's OK to set the fuse to the same value multiple times, but if you do a Make of a different program without changing the fuse setting, the behavior of the clock output pin won't change.

As I mentioned, the default clock rate for the ATtiny2313 is one megahertz. One megahertz is a pretty fast clock for events that humans can observe. If we want to see LEDs blink, then we need speeds no faster than tenths of seconds. Divide one megahertz by one hundred thousand to get a tenth of a second clock! To get clocks for timer/counters at slower rates, the counter/timers typically provide prescalers. The timer/counter prescalers are *in addition* to the prescalers we discussed above which set the processor clock speed. As a reminder, prescalers are simply dividers which divide a clock by a specific factor before passing it on, in this case to the timer/counter. AVR processors have four prescaler values available. The reason for having separate prescalers for the timer/counters is so that the timer/counter speeds can be controlled independently of the processor speed. As mentioned above, bits in TCCR1B select prescale values of 1,

8, 64, 256, or 1024. Specific examples will be given as we move through this tutorial; Dean's tutorial also has examples.

CTC Mode - Briefly

Let's continue with the next timer/counter mode, CTC Mode. CTC stands for Clear Timer on Compare Match Mode. We'll continue to use timer/counter 1 as a specific example. CTC mode works nearly identically to Normal Mode, but allows us to change the value of TOP. Instead of being limited to MAX (maximum counter value), we can put a value in another register, OCR1A. The behavior of the timer/counter is the same as in Normal Mode, but the value of TOP can be set arbitrarily. Dean provides a complete example of this mode as Part Four of his tutorial. That's why my discussion of CTC mode is brief – just long enough to show another figure. Figure 2 shows the behavior in this mode. Note that TOP is no longer the same as MAX, it is determined by the value in a register named OCR1A. But the behavior when COUNT reaches TOP is still the same: COUNT starts over at BOTTOM and an overflow event occurs.

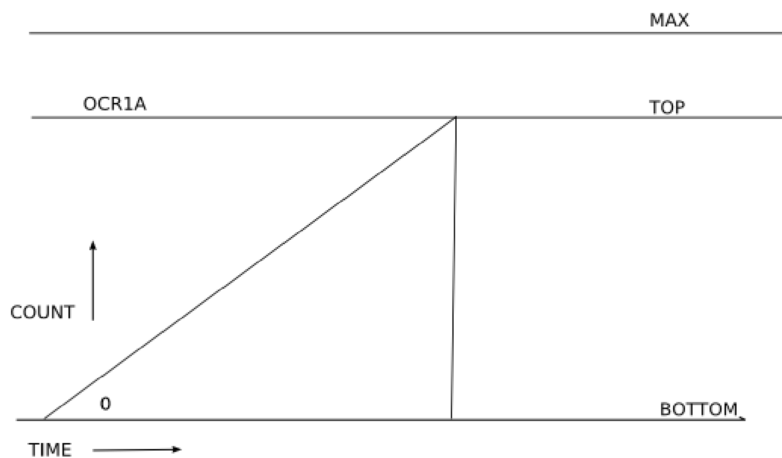


Figure 2 - Timer Operation with OCR1A as TOP (CTC Mode)

The value of TOP can also be changed while the timer/counter is running, although this can have unintended consequences. (While the data sheet explains this, my advice is to simply not do it. There are other modes you can use which don't have this problem. We'll get to those in a moment.)

An additional capability of CTC Mode is to allow the timer/counter to directly control an output bit (OC1A for timer/counter 1). Dean explains all about this in Part Six of his tutorial. Depending on the setting of the COM1A1 and COM1A0 bits in the TCCR1A register, OC1A will go high or low, or will toggle when TOP is reached and the timer resets to 0 (clears). Note that the OC1A bit must be configured as an output by setting bit PB3 in the Data Direction Register DDRB. (This detail applies to ATtiny2313.) Note also that the timer/counter behavior overrides normal port behavior. So if the clock selection is made 0 (meaning no clock selected), then we can directly control the value of OC1A

and it behaves as a normal I/O port bit – in fact, it **is** a normal I/O port bit. Conversely, selecting a clock source (with bits CS12, CS11, and CS10 in TCCR1B) enables the timer/counter and it will now control the value on OC1A. By choosing toggle, we can generate a nice square wave. This mode is the mode I used to drive a small speaker and generate a beeper tone in this Instructable.
http://www.instructables.com/id/Reading_Switches_with_ATtiny2313/
Dean explains the details completely; Figure 3 adds the behavior of OC1A to show the resulting waveform for a given value of TOP (OCR1A). Note that OC1A toggles, or changes its value each overflow. A square wave is thus generated.

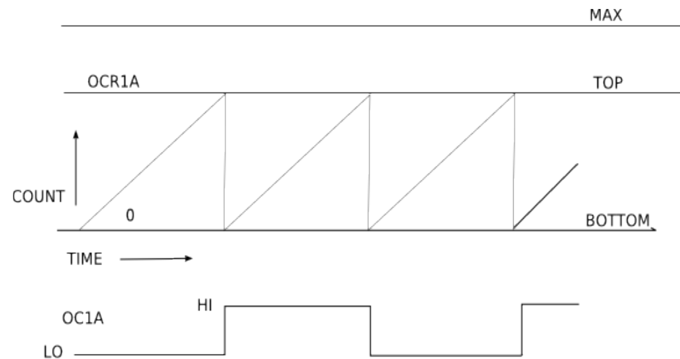


Figure 3 - OC1A output with OCR1A as TOP (CTC Mode)

On to the PWM Modes!

While CTC mode allows us to generate simple waveforms, we're mostly restricted to just creating a square wave. For greater control of waveforms, we need to use the timer/counter PWM modes. These modes are the *real* point of this tutorial. The term "PWM" seems to me to be a poor choice. These modes open up the power of the timer/counters and should be considered the actual normal modes of operation (IMHO).

PWM stands for Pulse Width Modulation. Googling for this term will lead you to a large amount of information and many thorough explanations. Dean's latest addition to his Timer/Counter Tutorial begins to describe PWM. While the PWM modes of the timer/counters will indeed do a very good job of PWM, they can do a lot more than the term PWM might imply. The PWM modes are the most useful and powerful modes available for the AVR timer/counters.

The first PWM mode we'll look at is called Fast PWM. Let's consider the additional capabilities of Fast PWM Mode one step at a time. Fast PWM Mode works very much like CTC mode, but adds another register for controlling the pulses on the OC1A pins. Fast PWM allows us to use the Input Capture Register (ICR1) as TOP. Now the value placed in the OCR1A register will be continuously compared to COUNT (the timer/counter value) as it increments. When the COUNT is equal to the value in OCR1A, the output on OC1A will go high. As soon as the count reaches TOP, OC1A will be

cleared. So we can again generate a square wave, but now we can easily vary the *duty cycl*e. The *duty cycl*e is simply the ratio of the time the output is high divided by the total length of one cycle. Figure 4 shows this new behavior, how ICR1 defines TOP, and how OCR1A controls the output duration. We can easily invert the output on OC1 by setting bits in TCCR1A.

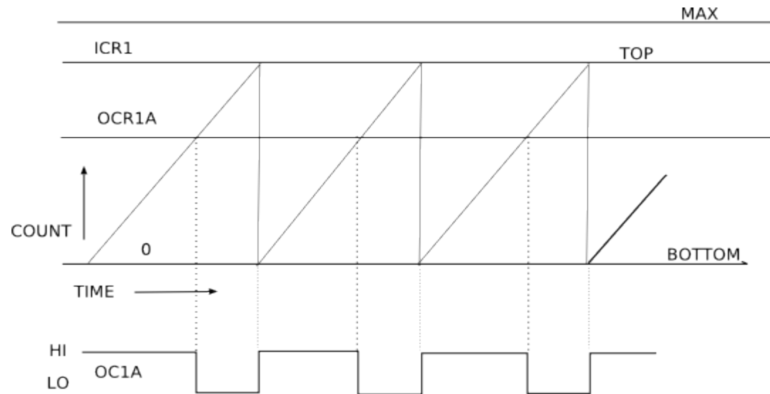


Figure 4 - OC1A output with OCR1A controlling PWM and ICR1 as TOP (Fast PWM Mode)

“Fine,” you say, “but I want the same level of control of the cycle time that CTC mode gave me.” And you shall have it. To do this, we use another register, OCR1B, along with OCR1A. By setting WGM bits appropriately, the cycle time is controlled by OCR1A and the duty cycle is controlled by OCR1B. Figure 5 shows the operation now. Note that OCR1A is changing and OCR1B is held constant. In fact, both of them can change and you can see an example in this Instructable.
http://www.instructables.com/id/Extreme_Surface_Mount_Soldering/

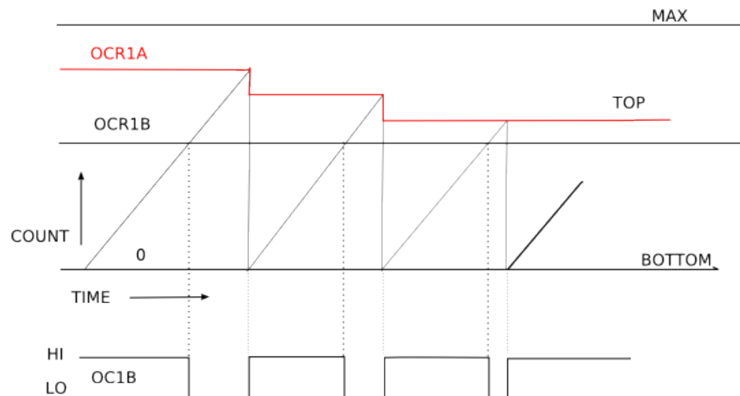


Figure 5 - OC1B output with OCR1B controlling PWM and OCR1A as TOP (Fast PWM Mode)

Remember my caution about changing the value of TOP while the timer/counter is running? You may rightly be concerned about this since Figure 5 suggests doing just this. What gives? Happily, the PWM modes add a level of protection when changing the value in OCR1A or OCR1B. When we enter a new value for OCR1A or OCR1B, that value is stored automatically, but the actual update of OCR1A or OCR1B is delayed until TOP or BOTTOM is reached (depends on the exact mode). Then the update occurs. This ensures that a new value of OCR1A or OCR1B will be seen correctly and not skipped. This technique is known as double buffering. That's why we can do what is shown in Figure 5.

“Great,” you say, “but I really need two outputs, not just one. I notice that there’s an OC1B output pin. Can I use that?” Good question! The answer is Yes! (Atmel seems to think of everything, don’t they?) Change the WGM bits and ICR1 will be used as TOP, and **both** OC1B and OC1A are available as outputs so two waveforms can be generated from the same counter. The polarity of the outputs is set independently. Figure 6 shows the results.

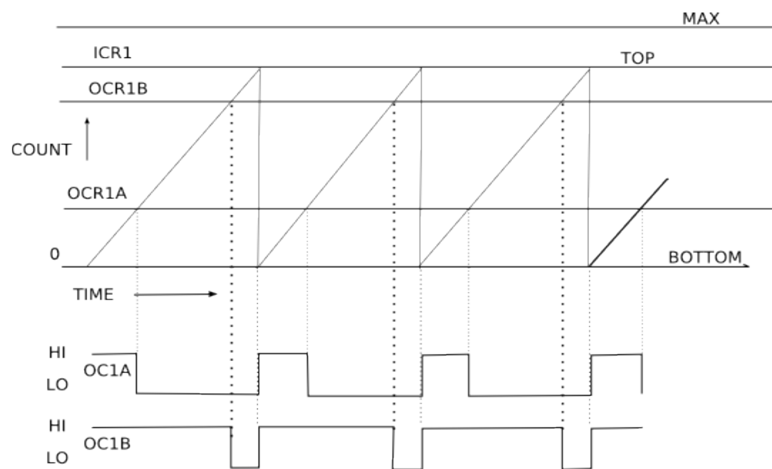


Figure 6 - OC1A & OC1B output with OCR1A & OCR1B controlling PWM, and ICR1 as TOP (Fast PWM Mode)

Unfortunately, even Atmel is forced eventually to impose some restrictions. ICR1 is not double buffered and cannot safely be changed while the timer/counter is running without great care. My advice is to simply not do this, but if you must, then please refer to the AVR data sheets for details since this is beyond the scope of this tutorial. We haven’t talked about Input Capture, but note that using the ICR1 register in this manner means you can’t use it for Input Capture.

The Real Deal: Phase and Frequency Correct PWM

Most of you are probably thinking, “Wow, this is great! What more could I possibly want from timer/counters?” Well, if you’re into controlling stepper motors, there is a problem with Fast PWM Mode. Part Nine of Dean's tutorial (the new section) discusses the problem. Dean illustrates the problem in his diagram showing Standard PWM and Phase

Correct PWM. Figure 7 (below) shows more detail of how the problem arises. Shown in the figure are the waveforms that result with two different values for OCR1A. Note that the leading edge of the pulse on OC1A always occurs at the same point. This causes the **center** of the pulse to shift. (Technically, causing a phase shift.) For motor control, we want the center of the pulse to be constant (no phase shifts).

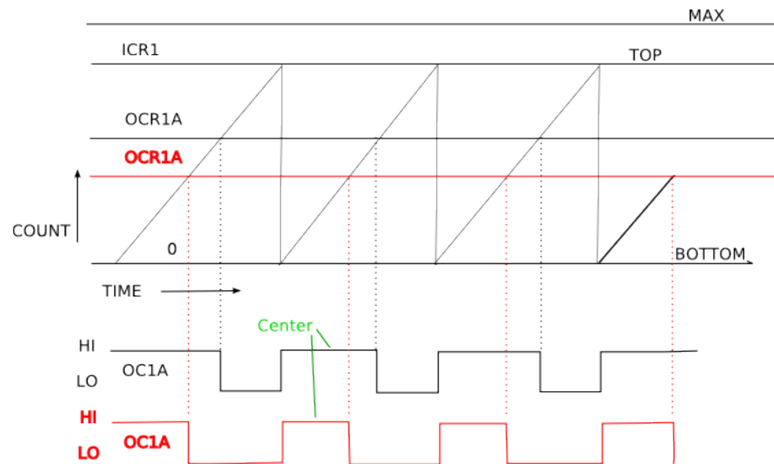


Figure 7 - OC1A output with OCR1A controlling PWM and ICR1 as TOP for Two Levels of OCR1A (Fast PWM Mode)

Cleverly, Atmel has found a way to generate just such waveforms using the timer/counters. By *decrementing* the count (reversing) when TOP is reached and comparing the values in the OCRs on the way up **and** on the way down, it is possible to do just what we want. Let's work through this additional complexity one step at a time.

Let's consider Phase & Frequency Correct PWM Mode in its simplest form. TOP will be set by ICR1, and OCR1A will define the waveform on OC1A. Note the change in behavior. No switching occurs at TOP. OC1A goes high when OCR1A = COUNT on the way up (incrementing) and goes low when OCR1A = COUNT on the way down (decrementing). The output will be as shown in Figure 8.

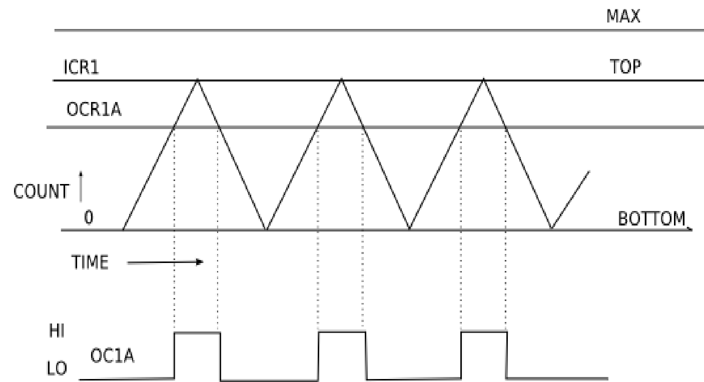


Figure 8 - OC1A output with OCR1A controlling PWM and ICR1 as TOP (P&F PWM Mode)

Here's the pseudo code. Since this is a pure hardware mode, similar to Part Six of Dean's tutorial, the code's pretty simple.

```
Set up Output Pins (LEDs)
Set Time Control Values
  1 Second Cycle
  1/10th Second On Time
Set up Timer in P&F Correct PWM Mode
  Mode with ICR1 as TOP, OCR1A Compare
  OC1A HI when COUNT = OCR1A up counting
  Clock prescaler divides by 1024
Loop Forever
```

Since each line of pseudo code here translates to just one or two lines of actual code, things can't get much simpler. I chose a prescaler value of 1024, so my timer clock tick (one count) will be (1 Sec/Hz) / (1MHz / 1024) = 1.024 milliseconds. So a one second cycle should be about 976 ticks (counts) long. One tenth of a second will be about 98 ticks long. I'll actually use 980 for one second so it divides evenly into tenths. Dean discusses how to get exact time durations, but this is close enough for the examples here. Note that we divide the times in two since we're counting up *and down* in this mode. To get a cycle time of 980 counts, we put 490 into ICR1. A cycle consists of 490 counts up, then 490 counts down.

To put timer/counter 1 into Phase and Frequency Correct PWM Mode, we look at Table 46 on page 110 of the ATtiny2313 data sheet. There are two possible modes, we'll select the one that uses ICR1 as TOP. This is Mode 8, and we only need to set bit WGM13.

Here's the code:

```
#include <avr_io.h>
```

```
int main(void)
{
	DDRB |=  _BV(CT1A) | _BV(CT1B) ;   /* Enable CT1 output pins */
	PORTB &=  ~ (_BV(CT1A) | _BV(CT1B));        /* Set them to lo - LEDs off */

	// Set ICR1 (TOP) and OCR1A
	ICR1 = 490;		// 1 second cycle time
	OCR1A = 451;		// Gives 1/10 second pulse

	// Configure Timer/Counter 1
	// Select P & F Correct PWM Mode, and ICR1 as TOP. Use WGM Bits.
	// WGM Bits are in TCCR1A and TCCR1B. Any previous settings are cleared.
	TCCR1A = 0;
	TCCR1B = _BV(WGM13);
	// OC1A HI w/ COUNT = OCR1A up counting, LO w/ COUNT = OCR1A down.
	TCCR1A |= _BV(COM1A1) | _BV(COM1A0);
	// Select Internal Clock Divided by 1024. This starts the Timer/Counter.
	TCCR1B |= _BV(CS12) | _BV(CS10);

	while (TRUE) // Do this forever
	{
		// nothing to do – timer/counter does the work!!
	}
}
```

If we want to change the duty cycle, we simply change the value in OCR1A just as we've done previously. But there is an important change in the behavior which is illustrated in Figure 9. Here I show the waveforms on OC1A for two different settings of OCR1A. This figure can be directly compared to Figure 7. Because the waveforms are centered about TOP, their centers stay aligned and there is no phase shift as the duty cycle is changed. Thus we have Phase Correct PWM! OCR1A (OCR1B also) is double buffered as before and only updated at BOTTOM. This maintains the phase correctness.

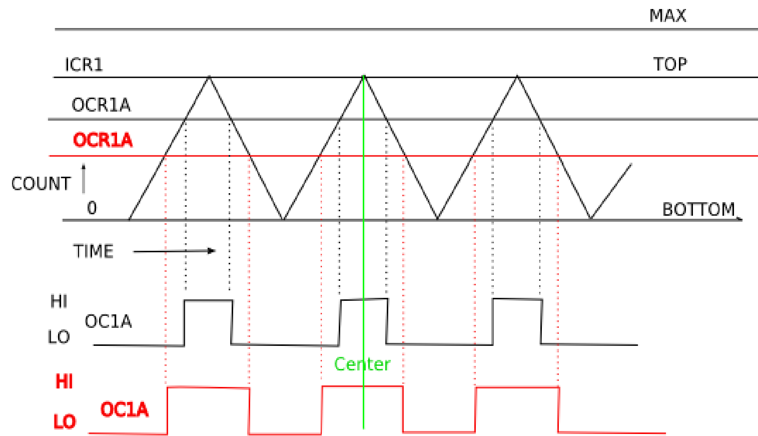


Figure 9 - OC1A output with OCR1A controlling PWM
and ICR1 as TOP for Two Levels of OCR1A (P&F PWM Mode)

Just like Fast PWM Mode, we can use OCR1A as TOP and let OCR1B control OC1B. Since OCR1A is double buffered, it's possible to change the cycle length (and thus the frequency) and have the change only occur at BOTTOM. Doing this means that only frequencies we select will ever be used – no short or long cycles and occur during changes. Hence, Phase and Frequency Correct PWM Mode. This latter point is not really that big a deal. Phase correctness is what's most important; but since Frequency correctness is also provided, we might as well use it. (I'll talk about timer/counter 0 briefly later, but note that it has only Phase Correct PWM Mode besides Fast PWM Mode. Phase and Frequency Correct PWM Mode is only slightly preferable; Phase Correct PWM Mode works very nearly as well.)

Finally (and anticipating your question), ICR1 can be used for TOP as long as you don't need to change it during operation (or change it very carefully). OCR1A and OCR1B can both control outputs on OC1A and OC1B. (I use this mode in a soon-to-be-posted Instructable to deliver full-power microstepping control to a stepper motor.) The resulting behavior is illustrated in Figure 10.

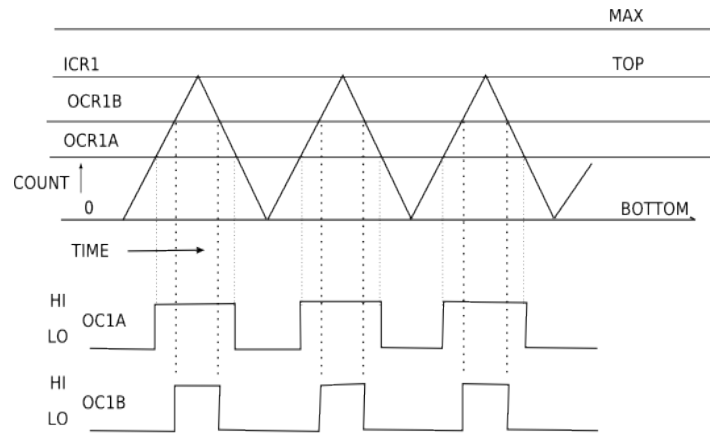
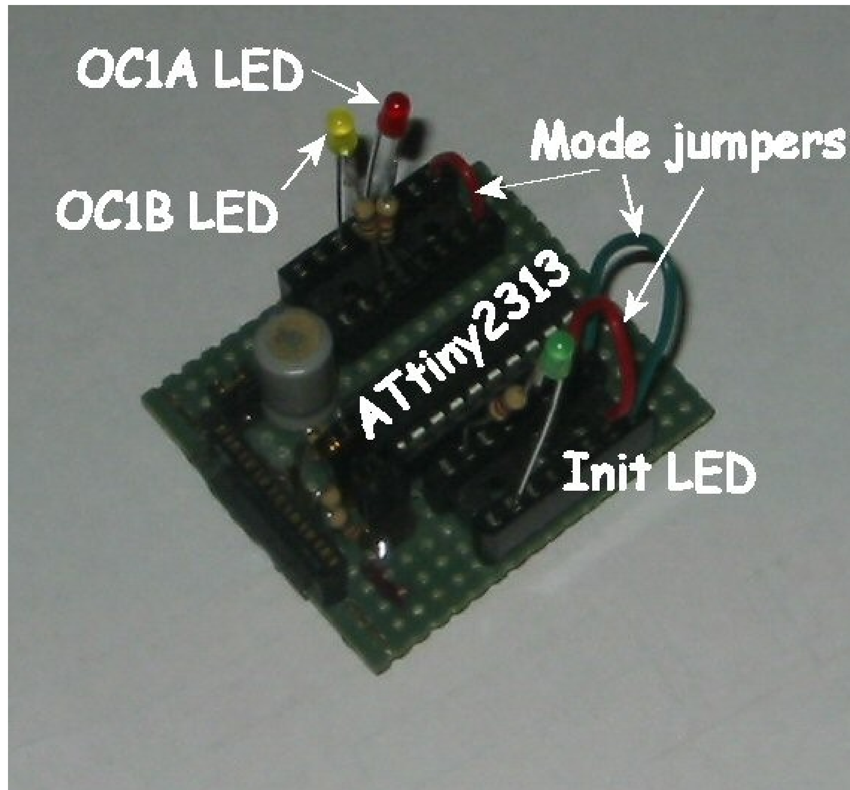


Figure 10 - OC1A & OC1B output with OCR1A & OCR1B controlling PWM, and ICR1 as TOP (P&F PWM Mode)

This mode of operation of the timer/counters is known as Phase and Frequency Correct PWM Mode. This is the mode I think should be called Normal and used for almost all applications of the timer/counters. There is also a mode known as Phase Correct PWM, but it appears to offer no advantage over Phase and Frequency Correct PWM and can be safely ignored, IMHO. (But it's just fine to use and the only phase correct mode available on 8 bit timer/counters like timer/counter 0 and 2.)

Useful Examples with Code

The example code is written for an ATtiny2313. Let's download and compile it now. Besides the CT_Examples.c file, there's a Makefile as well. Create a new directory and place them both in it. Then use WinAVR (or your weapon of choice) to compile, link and load it into an ATtiny2313. (Note the Makefile has the necessary lines to enable the clock output if you should so desire. They're commented out.) Connect blinkenlights to pins PB3 and PB4 and PD1; put jumpers to ground on PD4, PD5 and PD6 pins. (If you're unfamiliar with blinkenlights, a blinkenlight is simply an LED in series with a resistor - 220 to 1K ohm - not real critical. I make them in red, yellow, and green flavors.) The blinkenlight on PD1 is an initialization signal. It's just a friendly signal that the program has started – handy as you experiment. The jumpers on PD4-6 are used to select which example to actually run. The blinkenlight on PB3 is for OC1A while the one on PB4 is OC1B. Here's a picture of my setup.



Take a look at the code and you'll notice a few details. The first couple of lines in main() will change the clock rate if you wish to experiment with it. By uncommenting those lines, the processor clock will be set to 8MHz (by selecting a prescaler value of 1) and your pulses from the timer/counters will be 8 times as fast. This is an optional exercise.

Next is the initialization of the ports to enable OC1A and OC1B as outputs. Then we read the inputs PD4, PD5, and PD6 to select which example to actually run. With the three jumpers that the code reads, eight examples can be included. Currently, there's only five. Hook all the jumpers to ground to select Example 0, disconnect the jumper on PD4 to select Example 1, etc.

Each example uses the same basic structure. Here's the pseudo code for the examples. The values put in the control registers control the timing of the outputs. The values put in the timer/counter control registers select the mode using WGM bits 10 and 11 (in TCCR1A) and WGM bits 12 and 13 (in TCCR1B). Bits COM1A1 and COM1A0 (in TCCR1A) control the behavior of OC1A while bits COM1B1 and COM1B0 (in TCCR1A) control the behavior of OC1B. Bits CS12, CS11, and CS10 in TCCR1B control the clock source and prescaler selection. The settings for these registers and the bits in them are discussed for each example and are shown in the example code.

```
void TimerCounterControlModule (void)
{
        Define TOP and Compare Register values
```

```
	Configure TCCR1A and TCCR1B
	while(TRUE)
	{
		wait for timer overflow then reset flag bit
		if (count >= 10)		// arbitrary
		{
			swap time values in control register
			count = 0
		}
		else
		{
			count++
		}
	}
}
```

Example 0

This example shows how to use Phase and Frequency Correct PWM Mode (I'll abbreviate this to *P&F PWM* from now on.) in the simplest possible way. (I actually lied earlier in this section when I said that P&F PWM Mode with ICR1 as TOP was the simplest mode. This is even simpler; just not as commonly used.) It simply generates a square wave on OC1A by toggling every time OCR1A is equal to COUNT. Since OCR1A is TOP, then the toggle is at TOP. The value in OCR1A is changed every ten times to change the frequency of the square wave. Note the settings of the mode, action, and clock select bits. What we've done is duplicate CTC mode, but we've added the capability to have OCR1A double buffered. Just one example of why I think the P&F Correct PWM mode should be the Normal Mode.

Let's summarize the register settings.
Only OCR1A is used. Put 490 into it so it toggles every second (980 counts).
Select Mode 9, P & F PWM Mode To do this, we have to set bits WGM13 and WGM10 as shown in Table 46, page 110 of the ATtiny2313 data sheet. As mentioned above, these bits are spread between TCCR1A and TCCR1B, with WGM10 in TCCR1A and WGM13 in TCCR1B.
Next, we set OC1A to toggle when OCR1A overflows. Table 45 on page 109 explains how to do this. We simply set COM1A0 in TCCR1A.
Finally, we just need to select a clock source and the timer/counter will do the rest. As stated above, I'm using a prescale value of 1024. Table 47 on page 111 shows that we need to set CS12 and CS10 in TCCR1B, and off we go!

Here's the code to do just what we went through.

```
	// Set OCR1A (TOP)
	OCR1A = 49;		// toggles every one tenth second
```

```
// Configure Timer/Counter 1
// Select P & F Correct PWM Mode, and OCR1A as TOP. Use WGM Bits.
// WGM Bits are in TCCR1A and TCCR1B. Any previous settings are cleared.
TCCR1A = _BV(WGM10);
TCCR1B = _BV(WGM13);
// Configure OC1A to Toggle at TOP.
TCCR1A |= _BV(COM1A0);
// Select Internal Clock Divided by 1024. This starts the Timer/Counter.
TCCR1B |= _BV(CS12) | _BV(CS10);
```

That's pretty much it for Example 0. The only remaining item is the FOREVER loop (described in the generic Example code above). Every ten times through the loop, we switch OCR1A between the 49 we started with and 490 so the square wave period switches between one tenth second and one second. There's nothing special about the values I've selected, so experiment to your heart's content.

To run this example, be sure all the jumpers are in place on PD4, PD5, and PD6. This selects Example 0. You should see the LED on PD1 blink once, then the LED on PB3 (OC1A) should blink at a 1/10 second rate (pretty fast) ten times, blink at a 1 second rate ten times, and repeat until you get bored and turn off the power.

Example 1

Here we look at a slightly more complicated (but still pretty simple) configuration of P&F PWM Mode.
From Table 46 we select mode 8 which uses ICR1 as TOP. OCR1A is used as the Compare register and OC1A is the output.
We set the cycle time using ICR1A as 980 (two seconds) and set OCR1A to compare at 931 (which is 49 below 980). This will cause the LED to come on for 1/10th of a second every two seconds.
We configure OC1A to go HI when COUNT = OCR1A up counting, and go LO when COUNT = OCR1A down counting. We select this mode for OC1A by setting bits COM1A1 and COM1A0 in TCCR1A. This is explained in Table 45 on page 109.

Code:

```
// Set ICR1 (TOP) and OCR1A
ICR1 = 980;                 // two second cycle time
OCR1A = 931;                // one tenth second on time

// Configure Timer/Counter 1
// Select P & F Correct PWM Mode, and ICR1 as TOP. Use WGM Bits.
// WGM Bits are in TCCR1A and TCCR1B. Any previous settings are cleared.
TCCR1A = 0;                 // Just to clear the register.
TCCR1B = _BV(WGM13);
// OC1A HI w/ COUNT = OCR1A up counting, LO w/ COUNT = OCR1A down.
TCCR1A |= _BV(COM1A1) | _BV(COM1A0);
```

```
// Select Internal Clock Divided by 1024. This starts the Timer/Counter.
TCCR1B |= _BV(CS12) | _BV(CS10);
```

Every ten times through the loop, we switch OCR1A between the 931 we started with and 49 so the LED on time switches between one tenth second and 1.9 seconds.

To run this example, be sure the jumpers are in place on PD5, and PD6. Remove the one on PD4. This selects Example 1. You should see the LED on PD1 blink once, then the LED on PB3 (OC1A) should blink on for $1/10^{th}$ second every two seconds ten times, blink off for $1/10^{th}$ second every two seconds ten times, and repeat until you get bored and turn off the power.

Example 2

This configuration of P&F PWM is only slightly more complicated. We'll use OCR1A as TOP, as in Example 0, but we'll use OCR1B for the Compare register. Doing this allows us to change both the period and the duty cycle of the waveform. In this example, we'll only change the period, but you can experiment with changing the duty cycle as well. Have a look at the Extreme Surface Mount Soldering Instructable for another example of changing both the period and the duty cycle. I use PWM to control a hot plate for doing SMT soldering.

After the last two examples, you should have a pretty clear idea of how the control registers are set up. The code looks very much like what you've already seen, so I won't bore you with details. Look at Table 45 and 46, and at the code in CT_Examples.c.

Every ten times through the loop, we switch OCR1A between the 980 we started with and 98 so the cycle length switches between two seconds and two tenths second.

To run this example, be sure the jumpers are in place on PD4 and PD6. Remove the one on PD5. This selects Example 2. You should see the LED on PD1 blink once, then the LED on PB3 (OC1A) should blink on for $1/10^{th}$ second every two seconds ten times, then the cycle time should change to $2/10^{th}$ seconds while the LED continues to blink on for $1/10^{th}$ second each cycle ten times, and repeat until you get bored and turn off the power.

Example 3

Let's shift gears slightly in this example and learn about Fast PWM Mode. We'll use this example to illustrate the phase shift that comes with Fast PWM. To see this clearly, we'll use both OC1A and OC1B as outputs.

Fast PWM Mode differs from P&F PWM by only counting up. Refer to Figures 4, 5, and 6 earlier in this tutorial as a refresher. As a result, our counts will generally be doubled to yield the same times. That is, a desired two second cycle time will require that a value of 1960 be used for TOP.

Fast PWM Mode setup is very similar to what we've done previously.
Put 1960 in ICR1 to set a 2 second cycle. OCR1A gets 980 to create a one second on time for OC1A. OCR1B gets 490 to give a one half second pulse on OC1B.
Select Fast PWM Mode with ICR1 as TOP by setting the WGM bits according to Table 46.
Table 44 on page 108 shows how to configure OC1A and OC1B to go HI at TOP and LO on compare when up counting.
We keep the clock prescaler at 1024.

Every ten times through the loop, we switch OCR1B between the 490 we started with and 98 so the OC1B on time switches between one half second and two tenths second.

To run this example, be sure the jumper is in place on PD6. Remove the ones on PD4 and PD5. This selects Example 3. You should see the LED on PD1 blink once, then the LED on PB4 (OC1B) should blink on for one half second every two seconds ten times, then the on time should change to 2/10th seconds each cycle ten times, and repeat. At the same time, the LED on PB3 (OC1A) should blink continuously on for one second every two seconds. What you will notice is that both LEDs come on at the same instant each cycle, even though the LED on OC1B is on for less time than the one on OC1A. This illustrates phase incorrectness in this mode.

Example 4

For our final example, we'll shift back to P&F PWM Mode and see exactly what phase correctness is all about. We'll again use both OC1A and OC1B.

Setup is nearly identical to Example 1, with some changes in timing values and the use of OCR1B and OC1B for a second output.

Every ten times through the loop, we switch OCR1B between the 735 we started with and 931 so the OC1B on time switches between one half second and two tenths second.

To run this example, be sure the jumpers are in place on PD4 and PD5. Remove the one on PD6. This selects Example 4. You should see the LED on PD1 blink once, then the LED on PB4 (OC1B) should blink on for one half second every two seconds ten times, then the on time should change to 2/10th seconds each cycle ten times, and repeat. At the same time, the LED on PB3 (OC1A) should blink continuously on for one second every two seconds. What you will notice is that OC1B will turn on in the middle of the OC1A on time. The behavior is in contrast to that in Example 3 and illustrates the phase correctness that give this mode its name.

Conclusion and Final Details

You're probably thinking, “This is good stuff, but how about the other AVR timer/counters? Are they the same as timer/counter 1? Can I use them the same way?” Great questions!

The ATtiny2313 has one additional timer/counter: timer/counter 0. The ATmega168 has two additional timer/counters. Yup, you guessed it: timer/counter 0 and timer/counter 2. Now timer/counter 2 has some extra capabilities (subject of another tutorial?), but in general it's just like timer/counter 0. So let's compare these to timer/counter 1. Timer/counters 0 and 2 are only 8 bits wide. That is, they can only count to 255 before overflowing. They also have no Input Capture (ICR) register, so we're obviously restricted to what registers we can use as TOP. Finally, there is no Phase and Frequency Correct PWM mode; only Phase Correct PWM Mode – still a perfectly satisfactory mode. There are only 8 WGM mode selections.

So what's the bottom line on these guys? Clearly, the times you can use are limited by the 8 bit widths of the registers so the calculations are different. You can handle that. But if you want to have two outputs (OC0A and OC0B for example), you'll have to accept 255 (MAX) for TOP. Timer/counter 0 and 2 are still pretty useful! Just be aware of the restrictions.

“How about Fast PWM Mode?” you ask. “What's Fast about it?” Another great question. Let me try to explain.

Since Fast PWM Mode only counts up, it completes a cycle in half the time that P&F PWM Mode completes a cycle in, *if the same value is use for TOP*. So someone in Marketing at Atmel decided that must mean it's fast. I think this is completely bogus! Using ICR1 or OCR1A as top, we can get nearly any cycle time and duty cycle in P&F PWM Mode that we can get with Fast PWM Mode – and still be phase correct! So Fast PWM Mode buys us nothing and isn't really any faster. Only if you want the very fastest speed does it matter. Fast PWM Mode *may* be able (I haven't proved this yet) to put out a pulse at the processor clock rate, while P&F PWM Mode can only put out a pulse at ½ the processor clock rate. At these speeds, we're not really doing any PWM, just putting out a square wave. So who cares? Forget Fast PWM. It's not really fast.

You might also ask about interrupts since the timer/counters can generate them. Dean does a fine job of discussing these and includes some thorough examples. So I don't see any point in belaboring it. Just follow Dean's examples and make the necessary name changes.

In this tutorial, I have tried to build on the excellent timer/counter tutorials that Dean Camera has created. Hopefully, you all now realize just how powerful and useful Phase and Frequency Correct PWM Mode is. Using the examples I provide, you should understand how to set it up to do the things you want to do with a timer/counter. The figures in this tutorial have illustrated the important concepts and principals of timer/counter operation. You should be on the way to becoming an expert user of the AVR timer/counters.

“What's left?” you ask. “Has everything now been said about timer/counters?” Heavens no! There's still lots of material left for additional tutorials. We haven't talked at all about Input Capture, External Clock Sources, the extra modes of timer/counter 2, analog waveform generation using PWM (Dean seems to be working on this?), and lots more. Stay tuned!