Gammon Forum

Timers and counters

Postings by administrators only.

Posted by	**Nick Gammon** Australia (22,250 posts) bio *Forum Administrator*
Date	Tue 17 Jan 2012 01:17 AM (UTC) Amended on Sat 04 Jul 2015 04:40 AM (UTC) by Nick Gammon

Message

This page can be quickly reached from the link: http://www.gammon.com.au/timers

The Atmega328 (as on the Arduino Uno) has three timers/counters on-board the chip.

Timer 0 is set up by the init() function (which is automatically called by the code generated by the IDE, before setup() is called). This is used to count approximately every millisecond. This provides you with the figure that the millis() function returns.

You can use these timers easily enough by using the analogWrite function - that just generates a PWM (pulse width modulated) output on the various pins that the timer hardware supports.

But for a more in-depth analysis, let's look at using the timers/counters in our own way. :)

The example code below provides a "frequency counter" which counts the number of events which cause a rising edge on digital pin D5 during a specified interval.

For example, if you put a 5 kHz signal on pin D5, and time it for one second, the count will be 5000. You could also time it for 1/10 of a second (giving you a count of 500) and then multiply the result by 10, again giving you a figure of 5 kHz.

A longer timing period will give higher accuracy, and also average out any small variations during the sample time. However, of course, a longer timing period takes longer to execute.

Counter 1 - used to count pulses

In the code below Timer 1 is configured to count the number of times that a leading edge (rising pulse) is detected on D5. Each event increments the internal counter in the timer. When the 16-bit timer overflows an overflow interrupt is executed which counts the number of overflows.

When the time is up, the number of counts is the current counter contents of timer 1, plus the number of overflows multiplied by 65536.

Timer 2 - used to work out a timing interval

The counts are meaningless unless we know over what interval they occurred, which is what we use Timer 2 for. It is set up to take the internal clock (normally 16 MHz on a Uno), and "pre-scale" it by dividing it by 128. The pre-scaled clock will then "tick" every 8 microseconds (since the clock itself runs with a period of 1/16000000 or 62.5 ns).

So we configure Timer 2 to count up to 125 and then generate an interrupt. This interrupt gives us a chance to see if our counting period is up. Since 8 µs times 125 gives 1000 µs, that means we get interrupted exactly every 1 ms.

Note that Timer 2 has a higher priority than Timers 0 and 1, so neither the millis() timer, nor the Timer 1 counter will take precedence over this interrupt.

In the Timer 2 interrupt we see if time is up (basically whether the required number of milliseconds is up). Is not, we just keep going. If time is up, we turn off both Timers 1 and 2, calculate the total count (by multiplying the number of overflows by 65536 and adding in the remaining counts) and exit.

Frequency Counter sketch for Atmega328

```
// Timer and Counter example
// Author: Nick Gammon
// Date: 17th January 2012

// Input: Pin D5

// these are checked for in the main program
volatile unsigned long timerCounts;
volatile boolean counterReady;

// internal to counting routine
unsigned long overflowCount;
unsigned int timerTicks;
unsigned int timerPeriod;

void startCounting (unsigned int ms)
  {
  counterReady = false;         // time not up yet
  timerPeriod = ms;             // how many 1 ms counts to do
  timerTicks = 0;               // reset interrupt counter
  overflowCount = 0;            // no overflows yet

  // reset Timer 1 and Timer 2
  TCCR1A = 0;
  TCCR1B = 0;
  TCCR2A = 0;
  TCCR2B = 0;

  // Timer 1 - counts events on pin D5
  TIMSK1 = bit (TOIE1);   // interrupt on Timer 1 overflow

  // Timer 2 - gives us our 1 ms counting interval
  // 16 MHz clock (62.5 ns per tick) - prescaled by 128
  //  counter increments every 8 µs.
  // So we count 125 of them, giving exactly 1000 µs (1 ms)
  TCCR2A = bit (WGM21) ;   // CTC mode
  OCR2A  = 124;            // count up to 125  (zero relative!!!!)

  // Timer 2 - interrupt on match (ie. every 1 ms)
  TIMSK2 = bit (OCIE2A);   // enable Timer2 Interrupt

  TCNT1 = 0;      // Both counters to zero
  TCNT2 = 0;

  // Reset prescalers
  GTCCR = bit (PSRASY);        // reset prescaler now
  // start Timer 2
  TCCR2B =  bit (CS20) | bit (CS22) ;  // prescaler of 128
  // start Timer 1
  // External clock source on T1 pin (D5). Clock on rising edge.
  TCCR1B =  bit (CS10) | bit (CS11) | bit (CS12);
  }  // end of startCounting

ISR (TIMER1_OVF_vect)
  {
  ++overflowCount;               // count number of Counter1 overflows
  }  // end of TIMER1_OVF_vect


//******************************************************************
//  Timer2 Interrupt Service is invoked by hardware Timer 2 every 1 ms = 1000 Hz
//  16Mhz / 128 / 125 = 1000 Hz

ISR (TIMER2_COMPA_vect)
  {
  // grab counter value before it changes any more
  unsigned int timer1CounterValue;
  timer1CounterValue = TCNT1;  // see datasheet, page 117 (accessing 16-bit registers)
  unsigned long overflowCopy = overflowCount;

  // see if we have reached timing period
  if (++timerTicks < timerPeriod)
    return;  // not yet

  // if just missed an overflow
  if ((TIFR1 & bit (TOV1)) && timer1CounterValue < 256)
    overflowCopy++;

  // end of gate time, measurement ready

  TCCR1A = 0;    // stop timer 1
  TCCR1B = 0;

  TCCR2A = 0;    // stop timer 2
  TCCR2B = 0;

  TIMSK1 = 0;    // disable Timer1 Interrupt
  TIMSK2 = 0;    // disable Timer2 Interrupt
```

```
  // calculate total count
  timerCounts = (overflowCopy << 16) + timer1CounterValue;  // each overflow is 65536 more
  counterReady = true;              // set global flag for end count period
  }  // end of TIMER2_COMPA_vect

void setup ()
  {
  Serial.begin(115200);
  Serial.println("Frequency Counter");
  } // end of setup

void loop ()
  {
  // stop Timer 0 interrupts from throwing the count out
  byte oldTCCR0A = TCCR0A;
  byte oldTCCR0B = TCCR0B;
  TCCR0A = 0;    // stop timer 0
  TCCR0B = 0;

  startCounting (500);  // how many ms to count for

  while (!counterReady)
     { }  // loop until count over

  // adjust counts by counting interval to give frequency in Hz
  float frq = (timerCounts *  1000.0) / timerPeriod;

  Serial.print ("Frequency: ");
  Serial.print ((unsigned long) frq);
  Serial.println (" Hz.");

  // restart timer 0
  TCCR0A = oldTCCR0A;
  TCCR0B = oldTCCR0B;

  // let serial stuff finish
  delay(200);
  }   // end of loop
```

[EDIT] Amended 25 April 2012 to make more accurate by allowing for overflows in Timer 1 during the interrupt service routine, and by stopping Timer 0.

[EDIT] Amended 28 June 2013 to fix bug where I was testing for TIFR1 & TOV1 rather than TIFR1 & _BV (TOV1).

[EDIT] Amended 31 August 2013 to change _BV() to bit().

Accuracy

Pumping in a 5 MHz signal from a signal generator, the sketch outputs around 5001204 (give or take a couple of counts). The error (assuming the signal generator is accurate) is therefore 1204/500000 or about 0.02% error.

Trying with a 5 kHz signal, the sketch outputs around 5000 to 5002, an error of 2/5000 or 0.04% error.

So, pretty accurate. Tests on my Arduino clock showed that the clock itself was around 0.2% wrong, so we can't really expect better accuracy than that.

Range

I measured up to 8 MHz with about 0.5% error. At 5 MHz the error was down to 0.02% as described above. At the other end of the scale, it measured down to 10 Hz without any obvious error. Below that errors crept in, particularly as the sample period is only 500 ms.

More examples of timers and interrupts here:

http://gammon.com.au/interrupts

Frequency Counter sketch for Atmega2560

```
// Timer and Counter example for Mega2560
// Author: Nick Gammon
// Date: 24th April 2012

// input on pin D47 (T5)
```

```
// these are checked for in the main program
volatile unsigned long timerCounts;
volatile boolean counterReady;

// internal to counting routine
unsigned long overflowCount;
unsigned int timerTicks;
unsigned int timerPeriod;

void startCounting (unsigned int ms)
  {

  counterReady = false;         // time not up yet
  timerPeriod = ms;             // how many 1 ms counts to do
  timerTicks = 0;               // reset interrupt counter
  overflowCount = 0;            // no overflows yet

  // reset Timer 2 and Timer 5
  TCCR2A = 0;
  TCCR2B = 0;
  TCCR5A = 0;
  TCCR5B = 0;

  // Timer 5 - counts events on pin D47
  TIMSK5 = bit (TOIE1);   // interrupt on Timer 5 overflow

  // Timer 2 - gives us our 1 ms counting interval
  // 16 MHz clock (62.5 ns per tick) - prescaled by 128
  //  counter increments every 8 µs.
  // So we count 125 of them, giving exactly 1000 µs (1 ms)
  TCCR2A = bit (WGM21) ;   // CTC mode
  OCR2A  = 124;            // count up to 125  (zero relative!!!!)

  // Timer 2 - interrupt on match (ie. every 1 ms)
  TIMSK2 = bit (OCIE2A);   // enable Timer2 Interrupt

  TCNT2 = 0;
  TCNT5 = 0;      // Both counters to zero

  // Reset prescalers
  GTCCR = bit (PSRASY);        // reset prescaler now
  // start Timer 2
  TCCR2B =  bit (CS20) | bit (CS22) ;  // prescaler of 128
  // start Timer 5
  // External clock source on T4 pin (D47). Clock on rising edge.
  TCCR5B =  bit (CS50) | bit (CS51) | bit (CS52);

}  // end of startCounting

ISR (TIMER5_OVF_vect)
{
  ++overflowCount;               // count number of Counter1 overflows
}  // end of TIMER5_OVF_vect


//******************************************************************
//  Timer2 Interrupt Service is invoked by hardware Timer 2 every 1 ms = 1000 Hz
//  16Mhz / 128 / 125 = 1000 Hz

ISR (TIMER2_COMPA_vect)
{
  // grab counter value before it changes any more
  unsigned int timer5CounterValue;
  timer5CounterValue = TCNT5;  // see datasheet, (accessing 16-bit registers)

  // see if we have reached timing period
  if (++timerTicks < timerPeriod)
    return;  // not yet

  // if just missed an overflow
  if (TIFR5 & TOV5)
    overflowCount++;

  // end of gate time, measurement ready

  TCCR5A = 0;    // stop timer 5
  TCCR5B = 0;

  TCCR2A = 0;    // stop timer 2
  TCCR2B = 0;

  TIMSK2 = 0;    // disable Timer2 Interrupt
  TIMSK5 = 0;    // disable Timer5 Interrupt

  // calculate total count
  timerCounts = (overflowCount << 16) + timer5CounterValue;  // each overflow is 65536 more
  counterReady = true;              // set global flag for end count period
}  // end of TIMER2_COMPA_vect


void setup () {
  Serial.begin(115200);
  Serial.println("Frequency Counter");
} // end of setup


void loop () {

  // stop Timer 0 interrupts from throwing the count out
  byte oldTCCR0A = TCCR0A;
```

```
  byte oldTCCR0B = TCCR0B;
  TCCR0A = 0;    // stop timer 0
  TCCR0B = 0;

  startCounting (500);  // how many ms to count for

  while (!counterReady)
     { }  // loop until count over

  // adjust counts by counting interval to give frequency in Hz
  float frq = (timerCounts *  1000.0) / timerPeriod;

  // restart timer 0
  TCCR0A = oldTCCR0A;
  TCCR0B = oldTCCR0B;

  Serial.print ("Frequency: ");
  Serial.println ((unsigned long) frq);

  // let serial stuff finish
  delay(200);

}   // end of loop
```

[EDIT] Amended 25 April 2012 to make more accurate by allowing for overflows in Timer 1 during the interrupt service routine, and by stopping Timer 0.

[EDIT] Amended 4 September 2013 to change _BV() to bit().

Timer ready reckoner

To help work out what prescaler/count you need for setting up timers, consult this table:

Pre-scaler	Count	Frequency Hz	Period nS	Period uS	Period mS	Period (Sec)	
1	1	16,000,000	62.5	0.0625	0.0000625	0.0000000625	
8	1	2,000,000	500	0.5	0.0005	0.0000005	
32	1	500,000	2,000	2	0.002	0.000002	Timer 2 only
64	1	250,000	4,000	4	0.004	0.000004	
128	1	125,000	8,000	8	0.008	0.000008	Timer 2 only
256	1	62,500	16,000	16	0.016	0.000016	
1024	1	15,625	64,000	64	0.064	0.000064	
1	10	1,600,000	625	0.625	0.000625	0.000000625	
8	10	200,000	5,000	5	0.005	0.000005	
32	10	50,000	20,000	20	0.02	0.00002	Timer 2 only
64	10	25,000	40,000	40	0.04	0.00004	
128	10	12,500	80,000	80	0.08	0.00008	Timer 2 only
256	10	6,250	160,000	160	0.16	0.00016	
1024	10	1,563	640,000	640	0.64	0.00064	
1	50	320,000	3,125	3.125	0.003125	0.000003125	
8	50	40,000	25,000	25	0.025	0.000025	
32	50	10,000	100,000	100	0.1	0.0001	Timer 2 only
64	50	5,000	200,000	200	0.2	0.0002	
128	50	2,500	400,000	400	0.4	0.0004	Timer 2 only
256	50	1,250	800,000	800	0.8	0.0008	
1024	50	313	3,200,000	3,200	3.2	0.0032	
1	100	160,000	6,250	6.25	0.00625	0.00000625	
8	100	20,000	50,000	50	0.05	0.00005	
32	100	5,000	200,000	200	0.2	0.0002	Timer 2 only
64	100	2,500	400,000	400	0.4	0.0004	
128	100	1,250	800,000	800	0.8	0.0008	Timer 2 only
256	100	625	1,600,000	1,600	1.6	0.0016	
1024	100	156	6,400,000	6,400	6.4	0.0064	
1	200	80,000	12,500	12.5	0.0125	0.0000125	
8	200	10,000	100,000	100	0.1	0.0001	
32	200	2,500	400,000	400	0.4	0.0004	Timer 2 only
64	200	1,250	800,000	800	0.8	0.0008	
128	200	625	1,600,000	1,600	1.6	0.0016	Timer 2 only
256	200	313	3,200,000	3,200	3.2	0.0032	
1024	200	78	12,800,000	12,800	12.8	0.0128	
1	256	62,500	16,000	16	0.016	0.000016	
8	256	7,813	128,000	128	0.128	0.000128	
32	256	1,953	512,000	512	0.512	0.000512	Timer 2 only
64	256	977	1,024,000	1,024	1.024	0.001024	
128	256	488	2,048,000	2,048	2.048	0.002048	Timer 2 only
256	256	244	4,096,000	4,096	4.096	0.004096	
1024	256	61	16,384,000	16,384	16.384	0.016384	
Timer 1 only:							
1	1000	16,000	62,500	62.5	0.0625	0.0000625	
8	1000	2,000	500,000	500	0.5	0.0005	
64	1000	250	4,000,000	4,000	4	0.004	
256	1000	63	16,000,000	16,000	16	0.016	
1024	1000	16	64,000,000	64,000	64	0.064	

All these figures assume a 16 MHz clock, and thus a clock period of 62.5 ns.

The "count" column is the number of counts you use for the Output Compare Register (eg. for OCR2A, OCR2B and so on) when counting up to a "compare" value. The count of 256 can also be used to see how long until Timer 0 and Timer 2 overflow (Timer 1 is a 16-bit timer and overflows after 65536 counts).

Remember that the count registers are **zero-relative**, so to get 100 counts, you actually put 99 into the register.

So for example, with a prescaler of 64, Timer 0 will overflow every 1.024 ms (which in fact it normally does for use by the millis() function).

To set some other frequency choose a prescaler which appears reasonably close, and then apply this formula:

```
count = frequency_from_table / target_frequency
```

For example, if we wanted to flash an LED at 50 Hz using a prescaler of 1024:

```
count = 15625 / 50 = 312.5
```

Since 312.5 is greater than 256 we could only do that with Timer 1 which is a 16-bit timer. Note that 312.5 has a decimal place, and therefore the flash would not occur every 50 Hz (as it would be rounded down). You could choose a prescaler of 256 instead:

```
count = 62500 / 50 = 1250
```

Now, we have a whole number, so the frequency would be accurate.

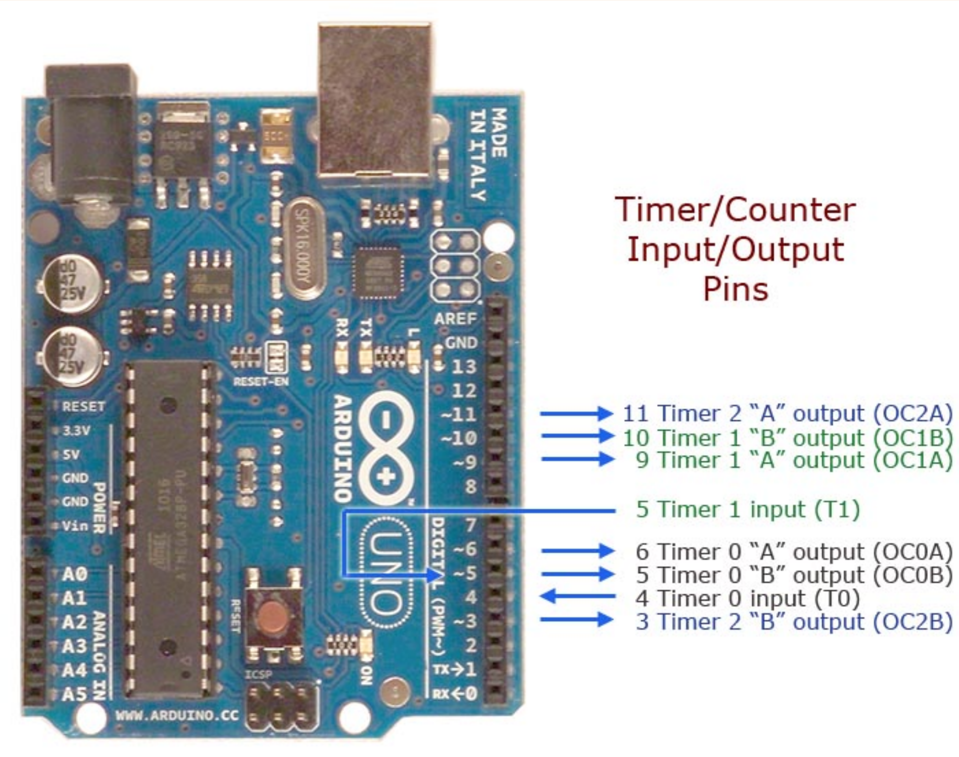
Bear in mind, too, that if you are using the hardware to toggle a pin, it needs to get toggled at twice the target frequency (because 100 Hz is counting 100 times a complete cycle, not half a cycle).

Timer hardware input/output

The table below shows the relationship between various pins (with the Arduino pin number in brackets) and the respective timers.

For example, to count an external source with Timer 1, you connect that to Arduino pin D5 (pin 11 on the Atmega328).

```
Timer 0

input     T0     pin  6  (D4)

output    OC0A   pin 12  (D6)
output    OC0B   pin 11  (D5)

Timer 1

input     T1     pin 11  (D5)

output    OC1A   pin 15  (D9)
output    OC1B   pin 16  (D10)

Timer 2

output    OC2A   pin 17  (D11)
output    OC2B   pin  5  (D3)
```



- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,250 posts) bio *Forum Administrator*

Date Reply #1 on Fri 10 Feb 2012 04:41 AM (UTC)

Amended on Sat 04 Jul 2015 04:42 AM (UTC) by Nick Gammon

Message

Another frequency counter

The frequency counter below works a bit differently. The one in the earlier post used a timer to count the number of "ticks" in a given interval, so it was literally counting the frequency.

The sketch below turns that around, and uses a timer to work out the interval between two consecutive rising edges on pin D2. This time we use a "rising" interrupt on D2 to notice the leading edge. We also set up a high-precision timer (Timer 1) which is a 16-bit timer.

By using no prescaler, Timer 1 counts 1 for every clock cycle (say, every 62.5 ns at 16 MHz). By multiplying the number of counts between the leading edges by 62.5, and then taking the inverse, we can deduce the frequency.

The advantage of this method is that we get a very quick calculation. For example, at 10 kHz the period is 1/10000, namely 100 µs, so we get our result 100 µs later.

```
// Frequency timer
// Author: Nick Gammon
// Date: 10th February 2012

// Input: Pin D2

volatile boolean first;
volatile boolean triggered;
volatile unsigned long overflowCount;
volatile unsigned long startTime;
volatile unsigned long finishTime;

// here on rising edge
void isr ()
{
  unsigned int counter = TCNT1;  // quickly save it

  // wait until we noticed last one
  if (triggered)
    return;

  if (first)
    {
    startTime = (overflowCount << 16) + counter;
    first = false;
    return;
    }

  finishTime = (overflowCount << 16) + counter;
  triggered = true;
  detachInterrupt(0);
}  // end of isr

// timer overflows (every 65536 counts)
ISR (TIMER1_OVF_vect)
{
  overflowCount++;
}  // end of TIMER1_OVF_vect


void prepareForInterrupts ()
  {
  // get ready for next time
  EIFR = bit (INTF0);  // clear flag for interrupt 0
  first = true;
  triggered = false;  // re-arm for next time
  attachInterrupt(0, isr, RISING);
  }  // end of prepareForInterrupts


void setup ()
  {
  Serial.begin(115200);
  Serial.println("Frequency Counter");

  // reset Timer 1
  TCCR1A = 0;
  TCCR1B = 0;
  // Timer 1 - interrupt on overflow
  TIMSK1 = bit (TOIE1);   // enable Timer1 Interrupt
  // zero it
```

```
  TCNT1 = 0;
  overflowCount = 0;
  // start Timer 1
  TCCR1B =  bit (CS10);  //  no prescaling

  // set up for interrupts
  prepareForInterrupts ();

  } // end of setup

void loop ()
  {

  if (!triggered)
    return;

  unsigned long elapsedTime = finishTime - startTime;
  float freq = F_CPU / float (elapsedTime);  // each tick is 62.5 ns at 16 MHz

  Serial.print ("Took: ");
  Serial.print (elapsedTime);
  Serial.print (" counts. ");

  Serial.print ("Frequency: ");
  Serial.print (freq);
  Serial.println (" Hz. ");

  // so we can read it
  delay (500);

  prepareForInterrupts ();
}   // end of loop
```

Note that due to the time taken to service the interrupts on the data's leading edges, the maximum achievable frequency you can sample is around 100 kHz (which would mean the ISR is taking around 10 µs).

[EDIT] See below (reply #12) for a modified version that uses the Input Capture Unit to find the moment that the time is up, which lets you count up to 200 kHz.

31 August 2013: Changed _BV() to bit().

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,250 posts) bio *Forum Administrator*

Date Reply #2 on Tue 21 Feb 2012 07:12 PM (UTC)

Amended on Sat 04 Jul 2015 04:46 AM (UTC) by Nick Gammon

Message

Timer setup

Below are some namespaces for easily setting up timers. They can be a bit tedious to get the various bit combinations right for the various modes.

The sketch has three "namespaces" (Timer0, Timer1, Timer2) which inside have a table of modes, and some enums for the various settings).

So for example, to set Timer 1 into mode 4 (CTC, top = OCR1A) with a prescaler of 1 (ie. no prescaler) and clearing timer output port 1A on compare you would do this:

```
Timer1::setMode (4, Timer1::PRESCALE_1, Timer1::CLEAR_A_ON_COMPARE);
```

That is a lot easier than setting up a lot of bit patterns.

```
/*
 Timer Helpers library.
```

```
Devised and written by Nick Gammon.
Date: 21 March 2012
Version: 1.0

Licence: Released for public use.

See: http://www.gammon.com.au/forum/?id=11504

 Example:

 // set up Timer 1
 TCNT1 = 0;         // reset counter
 OCR1A =  999;       // compare A register value (1000 * clock speed)

 // Mode 4: CTC, top = OCR1A
 Timer1::setMode (4, Timer1::PRESCALE_1, Timer1::CLEAR_A_ON_COMPARE);

 TIFR1 |= bit (OCF1A);    // clear interrupt flag
 TIMSK1 = bit (OCIE1A);   // interrupt on Compare A Match

*/

#ifndef _TimerHelpers_h
#define _TimerHelpers_h

#if defined(ARDUINO) && ARDUINO >= 100
  #include "Arduino.h"
#else
  #include "WProgram.h"
#endif

/* ---------------------------------------------------------------
 Timer 0 setup
 --------------------------------------------------------------- */

namespace Timer0
{
  // TCCR0A, TCCR0B
  const byte Modes [8] [2] =
  {

  { 0,                         0 },            // 0: Normal, top = 0xFF
  { bit (WGM00),               0 },            // 1: PWM, Phase-correct, top = 0xFF
  {               bit (WGM01), 0 },            // 2: CTC, top = OCR0A
  { bit (WGM00) | bit (WGM01), 0 },            // 3: Fast PWM, top = 0xFF
  { 0,                         bit (WGM02) },  // 4: Reserved
  { bit (WGM00),               bit (WGM02) },  // 5: PWM, Phase-correct, top = OCR0A
  {               bit (WGM01), bit (WGM02) },  // 6: Reserved
  { bit (WGM00) | bit (WGM01), bit (WGM02) },  // 7: Fast PWM, top = OCR0A

  };  // end of Timer0::Modes

  // Activation
  // Note: T0 is pin 6, Arduino port: D4
  enum { NO_CLOCK, PRESCALE_1, PRESCALE_8, PRESCALE_64, PRESCALE_256, PRESCALE_1024, T0_FALLING, T0_RISING };

  // what ports to toggle on timer fire
  enum { NO_PORT = 0,

    // pin 12, Arduino port: D6
    TOGGLE_A_ON_COMPARE  = bit (COM0A0),
    CLEAR_A_ON_COMPARE   = bit (COM0A1),
    SET_A_ON_COMPARE     = bit (COM0A0) | bit (COM0A1),

    // pin 11, Arduino port: D5
    TOGGLE_B_ON_COMPARE  = bit (COM0B0),
    CLEAR_B_ON_COMPARE   = bit (COM0B1),
    SET_B_ON_COMPARE     = bit (COM0B0) | bit (COM0B1),
  };


  // choose a timer mode, set which clock speed, and which port to toggle
  void setMode (const byte mode, const byte clock, const byte port)
  {
  if (mode < 0 || mode > 7)  // sanity check
    return;

  // reset existing flags
  TCCR0A = 0;
  TCCR0B = 0;

  TCCR0A |= (Modes [mode] [0]) | port;
  TCCR0B |= (Modes [mode] [1]) | clock;
  }  // end of Timer0::setMode

}  // end of namespace Timer0

/* ---------------------------------------------------------------
 Timer 1 setup
 --------------------------------------------------------------- */

namespace Timer1
{
  // TCCR1A, TCCR1B
  const byte Modes [16] [2] =
  {

  { 0,                         0 },            // 0: Normal, top = 0xFFFF
  { bit (WGM10),               0 },            // 1: PWM, Phase-correct, 8 bit, top = 0xFF
  {               bit (WGM11), 0 },            // 2: PWM, Phase-correct, 9 bit, top = 0x1FF
  { bit (WGM10) | bit (WGM11), 0 },            // 3: PWM, Phase-correct, 10 bit, top = 0x3FF
```

```
  { 0,                          bit (WGM12) },  // 4: CTC, top = OCR1A
  { bit (WGM10),                bit (WGM12) },  // 5: Fast PWM, 8 bit, top = 0xFF
  {              bit (WGM11), bit (WGM12) },  // 6: Fast PWM, 9 bit, top = 0x1FF
  { bit (WGM10) | bit (WGM11), bit (WGM12) },  // 7: Fast PWM, 10 bit, top = 0x3FF
  { 0,                                       bit (WGM13) },  // 8: PWM, phase and frequency correct, top = ICR1
  { bit (WGM10),                             bit (WGM13) },  // 9: PWM, phase and frequency correct, top = OCR1A
  {              bit (WGM11),                bit (WGM13) },  // 10: PWM, phase correct, top = ICR1A
  { bit (WGM10) | bit (WGM11),               bit (WGM13) },  // 11: PWM, phase correct, top = OCR1A
  { 0,                          bit (WGM12) | bit (WGM13) },  // 12: CTC, top = ICR1
  { bit (WGM10),                bit (WGM12) | bit (WGM13) },  // 13: reserved
  {              bit (WGM11), bit (WGM12) | bit (WGM13) },  // 14: Fast PWM, TOP = ICR1
  { bit (WGM10) | bit (WGM11), bit (WGM12) | bit (WGM13) },  // 15: Fast PWM, TOP = OCR1A

  };  // end of Timer1::Modes

  // Activation
  // Note: T1 is pin 11, Arduino port: D5
  enum { NO_CLOCK, PRESCALE_1, PRESCALE_8, PRESCALE_64, PRESCALE_256, PRESCALE_1024, T1_FALLING, T1_RISING };

  // what ports to toggle on timer fire
  enum { NO_PORT = 0,

    // pin 15, Arduino port: D9
    TOGGLE_A_ON_COMPARE  = bit (COM1A0),
    CLEAR_A_ON_COMPARE   = bit (COM1A1),
    SET_A_ON_COMPARE     = bit (COM1A0) | bit (COM1A1),

    // pin 16, Arduino port: D10
    TOGGLE_B_ON_COMPARE  = bit (COM1B0),
    CLEAR_B_ON_COMPARE   = bit (COM1B1),
    SET_B_ON_COMPARE     = bit (COM1B0) | bit (COM1B1),
  };

  // choose a timer mode, set which clock speed, and which port to toggle
  void setMode (const byte mode, const byte clock, const byte port)
  {
  if (mode < 0 || mode > 15)  // sanity check
    return;

  // reset existing flags
  TCCR1A = 0;
  TCCR1B = 0;

  TCCR1A |= (Modes [mode] [0]) | port;
  TCCR1B |= (Modes [mode] [1]) | clock;
  }  // end of Timer1::setMode

}  // end of namespace Timer1

/* ---------------------------------------------------------------
 Timer 2 setup
 --------------------------------------------------------------- */

namespace Timer2
{
  // TCCR2A, TCCR2B
  const byte Modes [8] [2] =
  {

  { 0,                          0 },            // 0: Normal, top = 0xFF
  { bit (WGM20),                0 },            // 1: PWM, Phase-correct, top = 0xFF
  {              bit (WGM21), 0 },            // 2: CTC, top = OCR2A
  { bit (WGM20) | bit (WGM21), 0 },            // 3: Fast PWM, top = 0xFF
  { 0,                          bit (WGM22) },  // 4: Reserved
  { bit (WGM20),                bit (WGM22) },  // 5: PWM, Phase-correct, top = OCR2A
  {              bit (WGM21), bit (WGM22) },  // 6: Reserved
  { bit (WGM20) | bit (WGM21), bit (WGM22) },  // 7: Fast PWM, top = OCR2A

  };  // end of Timer2::Modes

  // Activation
  enum { NO_CLOCK, PRESCALE_1, PRESCALE_8, PRESCALE_32, PRESCALE_64, PRESCALE_128, PRESCALE_256, PRESCALE_1024 };

  // what ports to toggle on timer fire
  enum { NO_PORT = 0,

    // pin 17, Arduino port: D11
    TOGGLE_A_ON_COMPARE  = bit (COM2A0),
    CLEAR_A_ON_COMPARE   = bit (COM2A1),
    SET_A_ON_COMPARE     = bit (COM2A0) | bit (COM2A1),

    // pin 5, Arduino port: D3
    TOGGLE_B_ON_COMPARE  = bit (COM2B0),
    CLEAR_B_ON_COMPARE   = bit (COM2B1),
    SET_B_ON_COMPARE     = bit (COM2B0) | bit (COM2B1),
  };


  // choose a timer mode, set which clock speed, and which port to toggle
  void setMode (const byte mode, const byte clock, const byte port)
  {
  if (mode < 0 || mode > 7)  // sanity check
    return;

  // reset existing flags
  TCCR2A = 0;
  TCCR2B = 0;

  TCCR2A |= (Modes [mode] [0]) | port;
  TCCR2B |= (Modes [mode] [1]) | clock;
  }  // end of Timer2::setMode
```

```
}  // end of namespace Timer2

#endif
```

The above can be downloaded from:

http://gammon.com.au/Arduino/TimerHelpers.zip

Just unzip and put the TimerHelpers folder into your libraries folder.

Example of use:

```
#include <TimerHelpers.h>

/* ---------------------------------------------------------------
   Test sketch
   --------------------------------------------------------------- */

const byte SHUTTER = 9;  // this is OC1A (timer 1 output compare A)

void setup() {
   pinMode (SHUTTER, INPUT);
   digitalWrite (SHUTTER, HIGH);

}  // end of setup

ISR(TIMER1_COMPA_vect)
{
   TCCR1A = 0;        // reset timer 1
   TCCR1B = 0;
}  // end of TIMER1_COMPA_vect

void loop() {
  delay (250);   // debugging

  TCCR1A = 0;        // reset timer 1
  TCCR1B = 0;

  digitalWrite (SHUTTER, HIGH);   // ready to activate
  pinMode (SHUTTER, OUTPUT);

 // set up Timer 1
  TCNT1 = 0;         // reset counter
  OCR1A =  999;       // compare A register value (1000 * clock speed)

  // Mode 4: CTC, top = OCR1A
  Timer1::setMode (4, Timer1::PRESCALE_1, Timer1::CLEAR_A_ON_COMPARE);

  TIFR1 |= bit (OCF1A);    // clear interrupt flag
  TIMSK1 = bit (OCIE1A);   // interrupt on Compare A Match

}  // end of loop
```

One-shot timer

The example code above demonstrates a one-shot timer. This sets up Timer 1 to activate a camera shutter for 62.5 µs (1000 x the clock speed of 62.5 ns), and then the interrupt service routine cancels the timer, so the shutter is only activated once.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,250 posts) bio *Forum Administrator*

Date Reply #3 on Mon 09 Apr 2012 02:48 AM (UTC)

Amended on Mon 22 Feb 2016 08:08 PM (UTC) by Nick Gammon

Message

Operation modes of Timer 0

Every time I go to use the Arduino (Atmega328) timers I wonder what the heck is the difference between the various modes. Do I want normal? CTC? PWM? Fast PWM?

Below are the results of investigating what each mode does, for timer 0.

I'm guessing that Timer 2 (which is also an 8-bit timer) works in a similar way.

Table 14-8. Waveform Generation Mode Bit Description

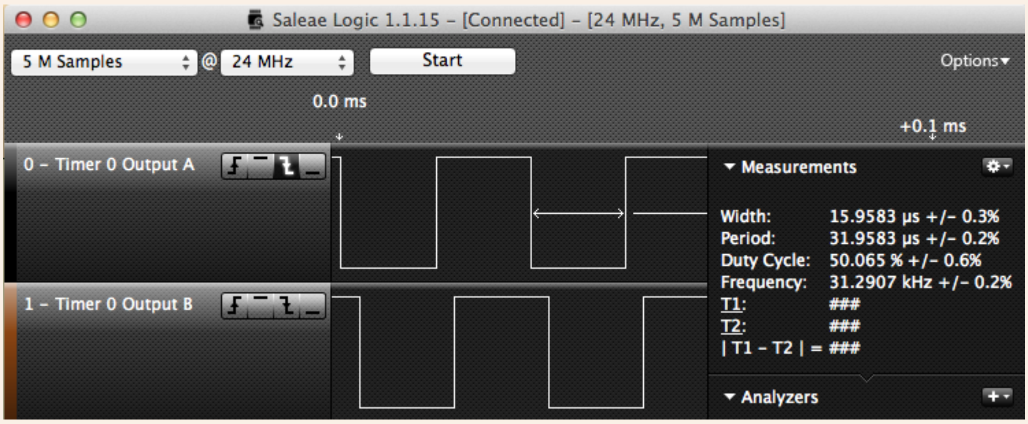
Mode	WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on[(1)(2)]
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	0	1	1	Fast PWM	0xFF	BOTTOM	MAX
4	1	0	0	Reserved	–	–	–
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	–	–	–
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Notes: 1. MAX = 0xFF
2. BOTTOM = 0x00

Timer 0, mode 0 (Normal mode)

This mode just counts to the maximum (0xFF) and wraps around. An interrupt can be set to go off at the wrap-around point. The counters (OCR0A and OCR0B) control at what point in the counting sequence the pins are toggled.

```
#include <TimerHelpers.h>

// Timer 0

// input     T0     pin  6  (D4)
// output    OC0A   pin 12  (D6)
// output    OC0B   pin 11  (D5)

const byte timer0Input = 4;
const byte timer0OutputA = 6;
const byte timer0OutputB = 5;

void setup() {
   pinMode (timer0OutputA, OUTPUT);
   pinMode (timer0OutputB, OUTPUT);
   TIMSK0 = 0;  // no interrupts
   Timer0::setMode (0, Timer0::PRESCALE_1, Timer0::TOGGLE_A_ON_COMPARE | Timer0::TOGGLE_B_ON_COMPARE);
   OCR0A = 150;
   OCR0B = 200;
}  // end of setup

void loop() {}
```

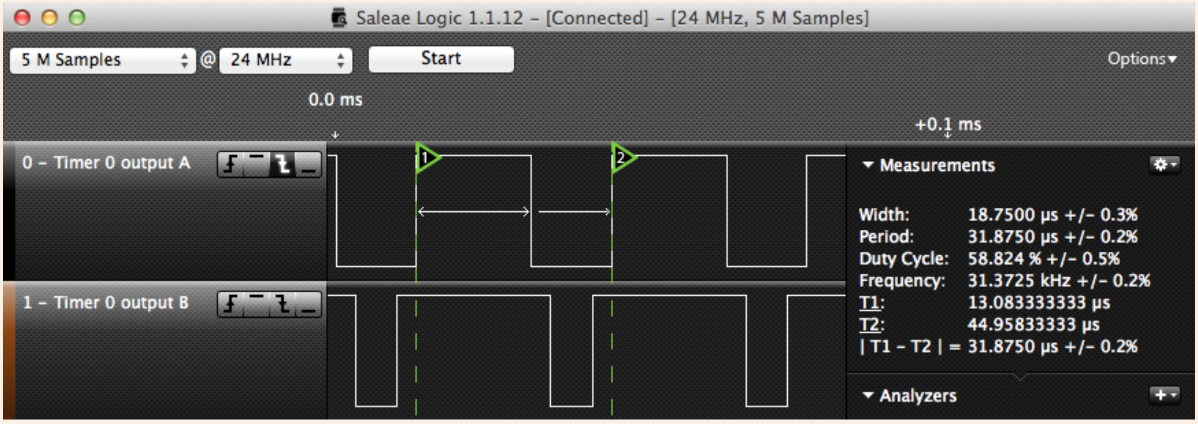


The period here was 16 µs which is 256 * 62.5 ns. Since I set the output pin to toggle on compare, it toggled every 16 µs.

Timer 0, mode 1 (PWM phase correct mode, top at 255)

This mode counts up to the maximum and then down again. On the first cycle (counting up) and if you have CLEAR_A_ON_COMPARE set, then the output is initially high for OCR0A/255 of the period (in the example: 150/255 which is a duty cycle of 58.82%), and then goes low. For the second cycle (counting down) it stays low, and flips back to high when the count is reached.

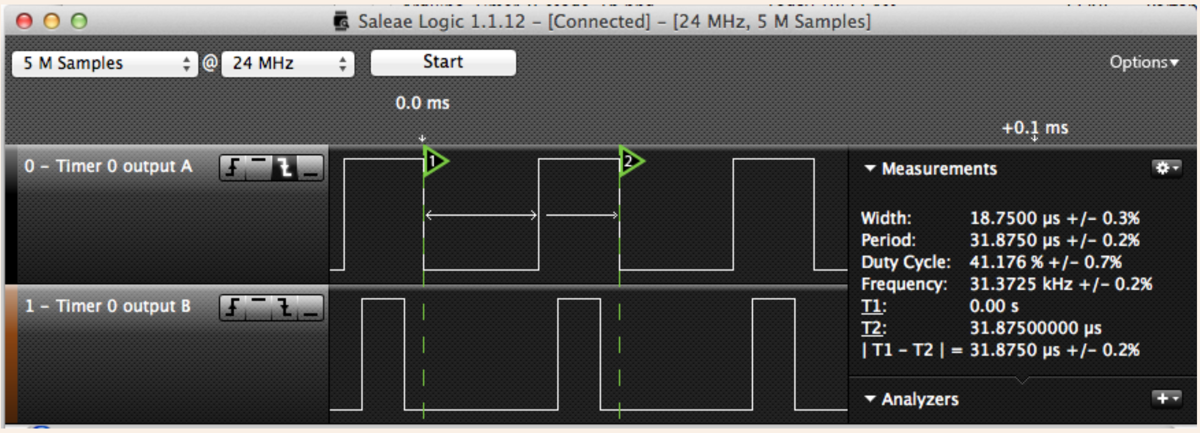
```
#include <TimerHelpers.h>

// Timer 0

// input     T0     pin  6  (D4)
// output    OC0A   pin 12  (D6)
// output    OC0B   pin 11  (D5)

const byte timer0Input = 4;
const byte timer0OutputA = 6;
const byte timer0OutputB = 5;

void setup() {
   pinMode (timer0OutputA, OUTPUT);
   pinMode (timer0OutputB, OUTPUT);
   OCR0A = 150;
   OCR0B = 200;
   TIMSK0 = 0;  // no interrupts
   Timer0::setMode (1, Timer0::PRESCALE_1, Timer0::CLEAR_A_ON_COMPARE | Timer0::CLEAR_B_ON_COMPARE);
}  // end of setup

void loop() {}
```



The total period here is 32 µs (and the duty cycle of the A output is 58.824%, that is: 150/255).

This could be regarded as "slow PWM" because an entire cycle takes 512 clock cycles (256 up, 256 down). All you get to adjust is the point in that cycle where the output is toggled.

If you change CLEAR to SET then the output is inverted, like this:



Now, the higher the counter, the longer the output is low (because it is set when the counter is reached).

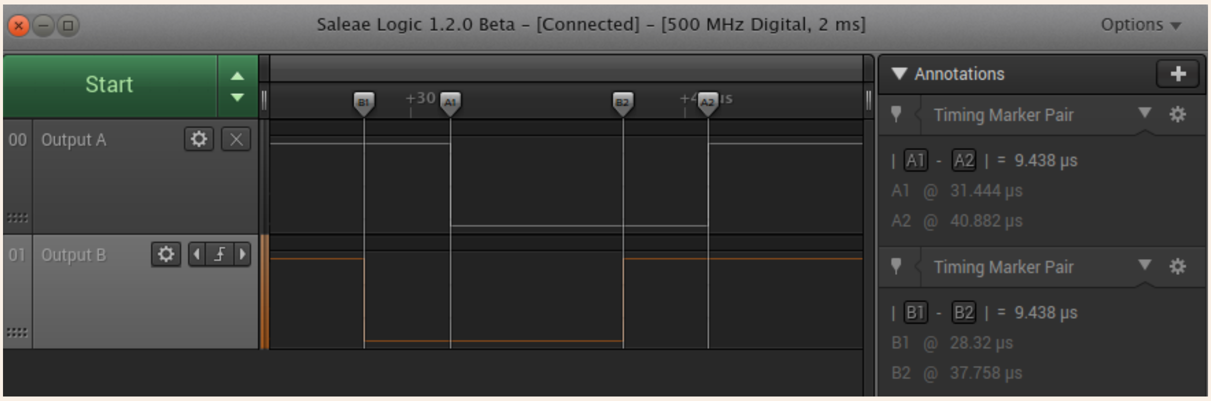
Timer 0, mode 2 (CTC mode)

This mode lets you control the timer frequency. CTC is Clear Timer on Count.

Modes 0 and 1 simply counted up to 256 or 512 respectively, thus giving a fixed frequency output. You could only alter the frequency by changing the timer prescaler.

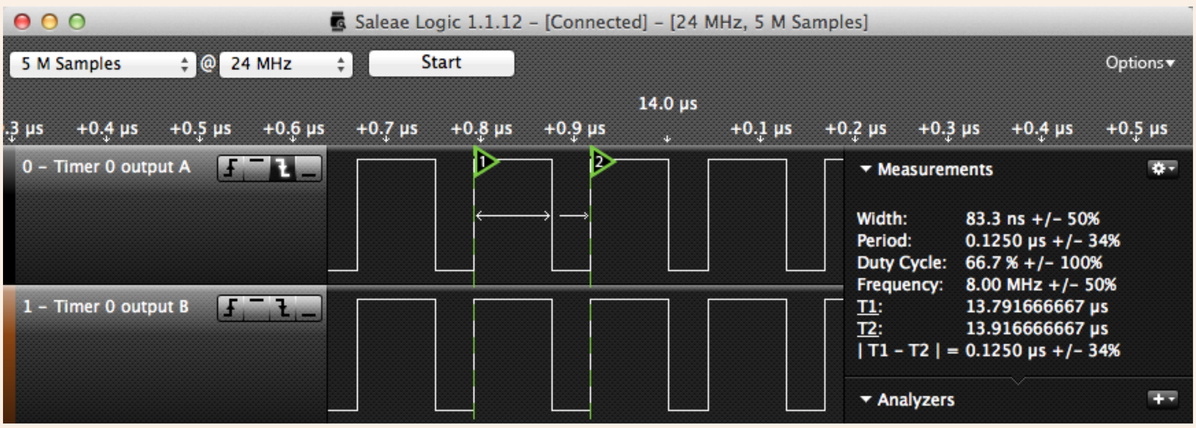
However in CTC mode the timer resets when it reaches the count. In the example, the period is 151 * 62.5 ns which is 9.438 µs. You multiply by 151 and not 150 because the timer is zero-relative (the first count is zero).

```
#include <TimerHelpers.h>

// Timer 0

// input     T0     pin  6  (D4)
// output    OC0A   pin 12  (D6)
// output    OC0B   pin 11  (D5)

const byte timer0Input = 4;
const byte timer0OutputA = 6;
const byte timer0OutputB = 5;

void setup() {
   pinMode (timer0OutputA, OUTPUT);
   pinMode (timer0OutputB, OUTPUT);
   TIMSK0 = 0;  // no interrupts
   Timer0::setMode (2, Timer0::PRESCALE_1, Timer0::TOGGLE_A_ON_COMPARE | Timer0::TOGGLE_B_ON_COMPARE);
   OCR0A = 150;
   OCR0B = 100;
}  // end of setup

void loop() {}
```



Notice that setting the B pin to toggle only works if OCR0B is not greater than OCR0A. However as you can see from the screen shot, OC0B toggles at the OC0A rate (because the timer counter stops at OCR0A). However OC0B is **offset** from OC0A by the value in OCR0B (100 counts).

Also note that you need to set the counter after starting the timer (as shown above) or you get rather strange results like this:



The timer for the above screenshot was started like this:

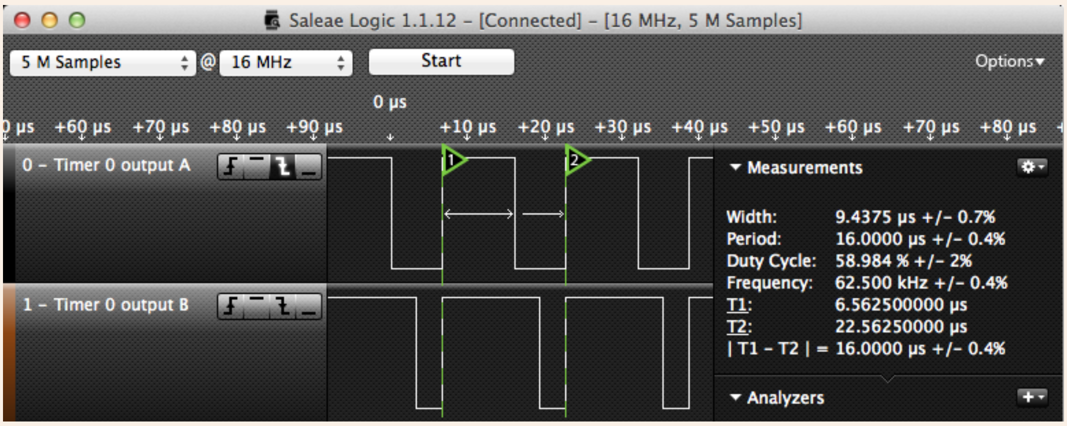
```
   OCR0A = 150;
   OCR0B = 200;
   Timer0::setMode (2, Timer0::PRESCALE_1, Timer0::TOGGLE_A_ON_COMPARE | Timer0::TOGGLE_B_ON_COMPARE);
```

Notice that the period of the timer seems to be just 62.5 ns (the processor clock cycle).

Timer 0, mode 3 (fast PWM mode, top at 255)

This mode counts up to the counter and toggles it when reached. In the example: 151/256 which is a duty cycle of 58.984%.

```
#include <TimerHelpers.h>

// Timer 0

// input     T0     pin  6  (D4)
// output    OC0A   pin 12  (D6)
// output    OC0B   pin 11  (D5)

const byte timer0Input = 4;
const byte timer0OutputA = 6;
const byte timer0OutputB = 5;

void setup() {
   pinMode (timer0OutputA, OUTPUT);
   pinMode (timer0OutputB, OUTPUT);
   TIMSK0 = 0;  // no interrupts
   Timer0::setMode (3, Timer0::PRESCALE_1, Timer0::CLEAR_A_ON_COMPARE | Timer0::CLEAR_B_ON_COMPARE);
   OCR0A = 150;
   OCR0B = 200;
}  // end of setup

void loop() {}
```



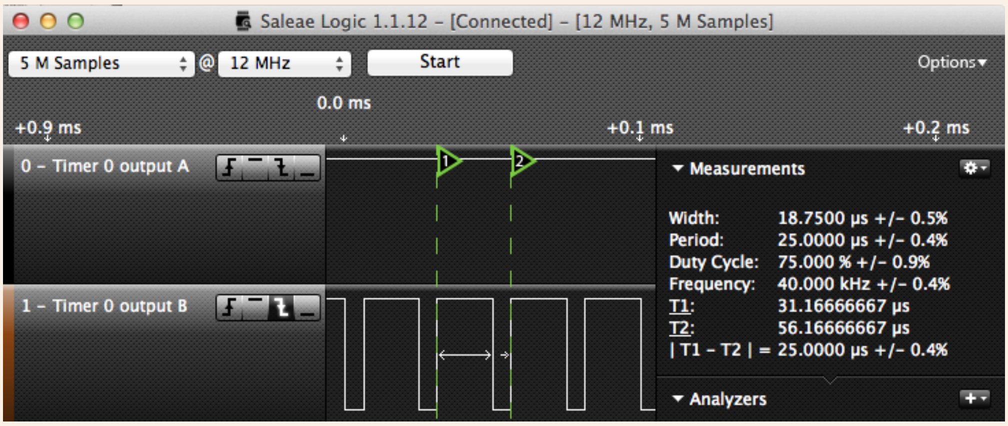
Notice that compared to the phase correct mode the frequency is doubled (it only counts up, not down again). Thus for a prescaler of 1, the period is going to be 256 * 62.5 ns, namely 16 µs. Also note how channel A and B are lined up, compared to how the pulses are centered in phase-correct mode.

Timer 0, mode 5 (PWM phase correct mode, top at OCR0A)

This mode counts up to the value in OCR0A and then down again. On the first cycle (counting up) and if you have CLEAR_A_ON_COMPARE set, then the output is initially high for OCR0B/OCR0A of the period (in the example: 150/255 which is a duty cycle of 75%), and then goes low. For the second cycle (counting down) it stays low, and flips back to high when the count is reached.

```
#include <TimerHelpers.h>

// Timer 0

// input     T0     pin  6  (D4)
// output    OC0A   pin 12  (D6)
// output    OC0B   pin 11  (D5)

const byte timer0Input = 4;
const byte timer0OutputA = 6;
```

```
const byte timer0OutputB = 5;

void setup() {
   pinMode (timer0OutputA, OUTPUT);
   pinMode (timer0OutputB, OUTPUT);
   TIMSK0 = 0;  // no interrupts
   Timer0::setMode (5, Timer0::PRESCALE_1, Timer0::CLEAR_A_ON_COMPARE | Timer0::CLEAR_B_ON_COMPARE);
   OCR0A = 200;    // number of counts for a cycle
   OCR0B = 150;    // duty cycle within OCR0A
}  // end of setup

void loop() {}
```



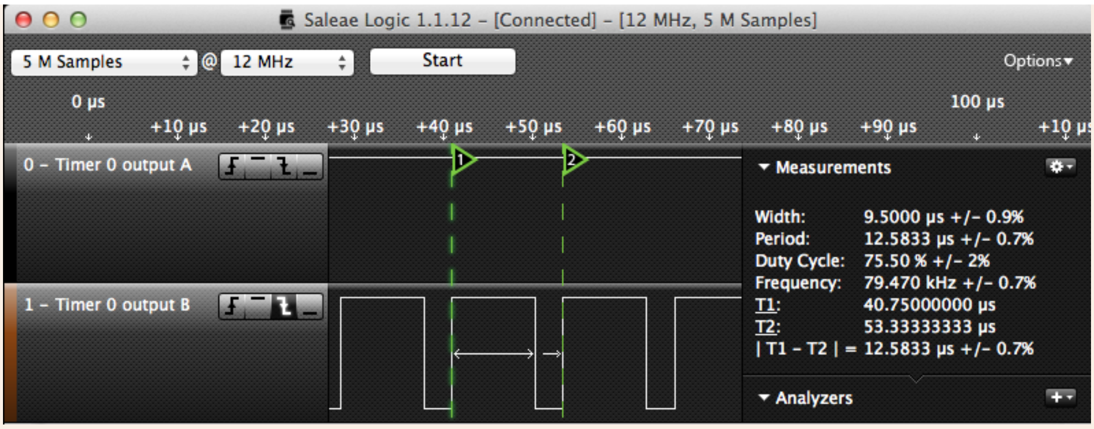
Note that OCR0A sets the frequency of the timer - since it counts up to that figure and back again. So you can use this to make a timer with a frequency other than simply 256 times the clock period. Of course the lower OCR0A is, the less resolution you have for the PWM duty cycle.

The overall frequency in this example was 40 kHz (period of 25 µs) because that is 200 * 62.5 ns * 2 (you multiply by two because it counts up and back down again).

Timer 0, mode 7 (fast PWM mode, top at OCR0A)

This mode counts up to OCR0B and toggles it when reached. It then counts (the rest of the way) up to OCR0A. This is the fast PWM version of mode 5.

```
#include <TimerHelpers.h>

// Timer 0

// input     T0     pin  6  (D4)
// output    OC0A   pin 12  (D6)
// output    OC0B   pin 11  (D5)

const byte timer0Input = 4;
const byte timer0OutputA = 6;
const byte timer0OutputB = 5;

void setup() {
   pinMode (timer0OutputA, OUTPUT);
   pinMode (timer0OutputB, OUTPUT);
   TIMSK0 = 0;  // no interrupts
   Timer0::setMode (7, Timer0::PRESCALE_1, Timer0::CLEAR_A_ON_COMPARE | Timer0::CLEAR_B_ON_COMPARE);
   OCR0A = 200;
   OCR0B = 150;
}  // end of setup

void loop() {}
```



Because it counts up to 200 the period is 200 * 62.5 ns (12.5 µs) and the duty cycle is now 75.5% (151/200).

Timer helpers library

The TimerHelpers.h file can be downloaded from:

http://gammon.com.au/Arduino/TimerHelpers.zip

More information

Some more useful information about PWM here, including some nice graphics that show how the phase-correct PWM works:

[What is PWM?]

Also this write-up by Ken Shirriff:

[Secrets of Arduino PWM]

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Willy** (4 posts) bio

Date Reply #4 on Wed 23 May 2012 10:07 AM (UTC)

Message Hello, Nick! Thanks for your job! I have tried your frequency counter sketch for Atmega328. I have modified slightly the code for my purposes for using LCD display and pre-scaler 1/10 with 74HC4017. It works fine up to 70MHz, however on high frequency's LCD shows 69.999 instead 70.000 MHz, but I don't need hight resolution, only 1KHz.

top

Posted by **Nick Gammon** Australia (22,250 posts) bio *Forum Administrator*

Date Reply #5 on Wed 23 May 2012 12:01 PM (UTC)

Message I meant to lock this thread as it is a tutorial. Would you mind posting it again in a new thread?

Plus, I doubt you got 70 MHz, after all the processor is only 16 MHz.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,250 posts) bio *Forum Administrator*

Date Reply #6 on Mon 24 Sep 2012 04:45 AM (UTC)

Amended on Sat 04 Jul 2015 04:45 AM (UTC) by Nick Gammon

Message

Modulating 38 kHz signal

This question seems to come up a few times on the Arduino forum: How to modulate a 38 kHz signal?

The code below uses Timer 1 to generate a 38 kHz pulse using fast PWM mode (mode 15). It then modulates the duty cycle from 0% to 100% based on a figure read from a potentiometer connected to A0.

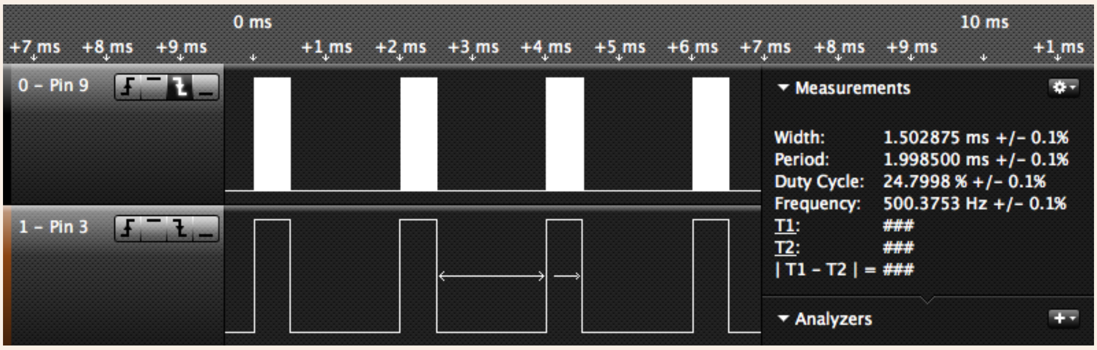
```
// Example of modulating a 38 kHz frequency duty cycle by reading a potentiometer
// Author: Nick Gammon
// Date: 24 September 2012

const byte POTENTIOMETER = A0;
const byte LED = 10;  // Timer 1 "B" output: OC1B

// Clock frequency divided by 38 kHz frequency desired
const long timer1_OCR1A_Setting = F_CPU / 38000L;

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

  // set up Timer 1 - gives us 38.005 kHz
  // Fast PWM top at OCR1A
  TCCR1A = bit (WGM10) | bit (WGM11) | bit (COM1B1); // fast PWM, clear OC1B on compare
  TCCR1B = bit (WGM12) | bit (WGM13) | bit (CS10);   // fast PWM, no prescaler
  OCR1A =  timer1_OCR1A_Setting - 1;                 // zero relative
  }  // end of setup

void loop()
  {
  // alter Timer 1 duty cycle in accordance with pot reading
  OCR1B = (((long) (analogRead (POTENTIOMETER) + 1) * timer1_OCR1A_Setting) / 1024L) - 1;

  // do other stuff here
  }
```

Modulating a 38 kHz carrier with a 500 Hz signal

Similar to the above, the sketch below generates a 38 kHz signal and then turns that carrier on and off with a 500 Hz signal (generated by Timer 2) with a variable duty cycle controlled by a potentiometer. The 500 Hz duty cycle is output on pin 3 which causes a pin change interrupt which is used to turn pin 9 on and off.

```
// Example of modulating a 38 kHz carrier frequency at 500 Hz with a variable duty cycle
// Author: Nick Gammon
// Date: 24 September 2012


const byte POTENTIOMETER = A0;
const byte LED = 9;  // Timer 1 "A" output: OC1A

// Clock frequency divided by 500 Hz frequency desired (allowing for prescaler of 128)
const long timer2_OCR2A_Setting = F_CPU / 500L / 128L;

ISR (PCINT2_vect)
   {

   // if pin 3 now high, turn on toggling of OC1A on compare
   if (PIND & bit (3))
     {
     TCCR1A |= bit (COM1A0) ;  // Toggle OC1A on Compare Match
     }
   else
     {
     TCCR1A &= ~bit (COM1A0) ;  // DO NOT Toggle OC1A on Compare Match
     digitalWrite (LED, LOW);  // ensure off
     }  // end of if

   }  // end of PCINT2_vect


void setup() {
  pinMode (LED, OUTPUT);
  pinMode (3, OUTPUT);  // OC2B

  // set up Timer 1 - gives us 38.095 kHz
  TCCR1A = 0;
  TCCR1B = bit(WGM12) | bit (CS10);   // CTC, No prescaler
  OCR1A =  (F_CPU / 38000L / 2) - 1;  // zero relative

  // Timer 2 - gives us our 1 ms counting interval
  // 16 MHz clock (62.5 ns per tick) - prescaled by 128
  //  counter increments every 8 µs.
```

```
  // So we count 250 of them, giving exactly 2000 µs (2 ms period = 500 Hz frequency)
  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1);   // Fast PWM mode
  TCCR2B = bit (WGM22) | bit (CS20) | bit (CS22) ;  // prescaler of 128
  OCR2A  = timer2_OCR2A_Setting - 1;                // count up to 250  (zero relative!!!!)

  // pin change interrupt
  PCMSK2 |= bit (PCINT19);  // want pin 3
  PCIFR  |= bit (PCIF2);    // clear any outstanding interrupts
  PCICR  |= bit (PCIE2);    // enable pin change interrupts for D0 to D7

}  // end of setup

void loop()
  {
  // alter Timer 2 duty cycle in accordance with pot reading
  OCR2B = (((long) (analogRead (POTENTIOMETER) + 1) * timer2_OCR2A_Setting) / 1024L) - 1;

  // other stuff here
  }  // end of loop
```



Simple timer output

The code below outputs on pin 3 a square wave using fast PWM mode of the desired frequency (constant "frequency").

This particular case uses a prescaler of 8 (hence the divide by 8 in the calculation of OCR2A).

You could change the duty cycle in the calculation of OCR2B (divide by 3, for example).

```
const byte LED = 3;  // Timer 2 "B" output: OC2B

const long frequency = 50000L;  // Hz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2B on compare
  TCCR2B = bit (WGM22) | bit (CS21);         // fast PWM, prescaler of 8
  OCR2A =  ((F_CPU / 8) / frequency) - 1;    // zero relative
  OCR2B = ((OCR2A + 1) / 2) - 1;             // 50% duty cycle
  }  // end of setup

void loop()
  {
  // do other stuff here
  }
```

With a 16 MHz clock, and with the prescaler of 8 you can generate frequencies in the range 7813 Hz to 1 MHz.

With no prescaler the sketch looks like this:

```
const byte LED = 3;  // Timer 2 "B" output: OC2B

const long frequency = 800000L;

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS20);         // fast PWM, no prescaler
```

```
  OCR2A =  (F_CPU / frequency) - 1;          // zero relative
  OCR2B = ((OCR2A + 1) / 2) - 1;             // 50% duty cycle
  }  // end of setup

void loop()
  {
  // do other stuff here
  }
```

With a 16 MHz clock, this can generate frequencies in the range 62.5 kHz to 4 MHz (8 MHz ought to work but has artifacts on the measured output).

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,250 posts) bio *Forum Administrator*

Date Reply #7 on Sat 24 Nov 2012 01:38 AM (UTC)

Amended on Sat 04 Jul 2015 04:49 AM (UTC) by Nick Gammon

Message

Frequencies and periods for various counter values

Below is a table of values for OCR2A (n) which shows the period and frequency you get for each possible value and prescaler, assuming a 16 MHz clock rate. The same figures apply to Timers 0 and 1 with the same prescalers. Note that the value in the first column is already zero-relative. This is what you plug into OCR2A to get this frequency. Thus the value of 255, for example, actually counts to 256.

The value in OCR2B controls the duty cycle. It must be less than the counter number. For example, if you count to 3 (which is really a count of 4 because the 3 is zero-relative) then for a 50% duty cycle you would want to put 1 (really a count of 2) into OCR2B.

```
  n  -- Prescale 1   --  -- Prescale 8   --  -- Prescale 64  --  -- Prescale 256 --  -- Prescale 1024--
     Freq (Hz) Per (µs)  Freq (Hz) Per (µs)  Freq (Hz) Per (µs)  Freq (Hz) Per (µs)  Freq (Hz) Per (µs)
  1: 8,000,000*   0.125  1,000,000    1.000    125,000    8.000     31,250   32.000      7,813  128.000
  2: 5,333,333    0.188    666,667    1.500     83,333   12.000     20,833   48.000      5,208  192.000
  3: 4,000,000    0.250    500,000    2.000     62,500   16.000     15,625   64.000      3,906  256.000
  4: 3,200,000    0.313    400,000    2.500     50,000   20.000     12,500   80.000      3,125  320.000
  5: 2,666,667    0.375    333,333    3.000     41,667   24.000     10,417   96.000      2,604  384.000
  6: 2,285,714    0.438    285,714    3.500     35,714   28.000      8,929  112.000      2,232  448.000
  7: 2,000,000    0.500    250,000    4.000     31,250   32.000      7,813  128.000      1,953  512.000
  8: 1,777,778    0.563    222,222    4.500     27,778   36.000      6,944  144.000      1,736  576.000
  9: 1,600,000    0.625    200,000    5.000     25,000   40.000      6,250  160.000      1,563  640.000
 10: 1,454,545    0.688    181,818    5.500     22,727   44.000      5,682  176.000      1,420  704.000
 11: 1,333,333    0.750    166,667    6.000     20,833   48.000      5,208  192.000      1,302  768.000
 12: 1,230,769    0.813    153,846    6.500     19,231   52.000      4,808  208.000      1,202  832.000
 13: 1,142,857    0.875    142,857    7.000     17,857   56.000      4,464  224.000      1,116  896.000
 14: 1,066,667    0.937    133,333    7.500     16,667   60.000      4,167  240.000      1,042  960.000
 15: 1,000,000    1.000    125,000    8.000     15,625   64.000      3,906  256.000        977 1024.000
 16:   941,176    1.063    117,647    8.500     14,706   68.000      3,676  272.000        919 1088.000
 17:   888,889    1.125    111,111    9.000     13,889   72.000      3,472  288.000        868 1152.000
 18:   842,105    1.187    105,263    9.500     13,158   76.000      3,289  304.000        822 1216.000
 19:   800,000    1.250    100,000   10.000     12,500   80.000      3,125  320.000        781 1280.000
 20:   761,905    1.313     95,238   10.500     11,905   84.000      2,976  336.000        744 1344.000
 21:   727,273    1.375     90,909   11.000     11,364   88.000      2,841  352.000        710 1408.000
 22:   695,652    1.438     86,957   11.500     10,870   92.000      2,717  368.000        679 1472.000
 23:   666,667    1.500     83,333   12.000     10,417   96.000      2,604  384.000        651 1536.000
 24:   640,000    1.562     80,000   12.500     10,000  100.000      2,500  400.000        625 1600.000
 25:   615,385    1.625     76,923   13.000      9,615  104.000      2,404  416.000        601 1664.000
 26:   592,593    1.688     74,074   13.500      9,259  108.000      2,315  432.000        579 1728.000
 27:   571,429    1.750     71,429   14.000      8,929  112.000      2,232  448.000        558 1792.000
 28:   551,724    1.813     68,966   14.500      8,621  116.000      2,155  464.000        539 1856.000
 29:   533,333    1.875     66,667   15.000      8,333  120.000      2,083  480.000        521 1920.000
 30:   516,129    1.938     64,516   15.500      8,065  124.000      2,016  496.000        504 1984.000
 31:   500,000    2.000     62,500   16.000      7,813  128.000      1,953  512.000        488 2048.000
 32:   484,848    2.062     60,606   16.500      7,576  132.000      1,894  528.000        473 2112.000
 33:   470,588    2.125     58,824   17.000      7,353  136.000      1,838  544.000        460 2176.000
 34:   457,143    2.188     57,143   17.500      7,143  140.000      1,786  560.000        446 2240.000
 35:   444,444    2.250     55,556   18.000      6,944  144.000      1,736  576.000        434 2304.000
 36:   432,432    2.313     54,054   18.500      6,757  148.000      1,689  592.000        422 2368.000
 37:   421,053    2.375     52,632   19.000      6,579  152.000      1,645  608.000        411 2432.000
 38:   410,256    2.438     51,282   19.500      6,410  156.000      1,603  624.000        401 2496.000
 39:   400,000    2.500     50,000   20.000      6,250  160.000      1,563  640.000        391 2560.000
 40:   390,244    2.563     48,780   20.500      6,098  164.000      1,524  656.000        381 2624.000
 41:   380,952    2.625     47,619   21.000      5,952  168.000      1,488  672.000        372 2688.000
 42:   372,093    2.687     46,512   21.500      5,814  172.000      1,453  688.000        363 2752.000
 43:   363,636    2.750     45,455   22.000      5,682  176.000      1,420  704.000        355 2816.000
 44:   355,556    2.812     44,444   22.500      5,556  180.000      1,389  720.000        347 2880.000
 45:   347,826    2.875     43,478   23.000      5,435  184.000      1,359  736.000        340 2944.000
 46:   340,426    2.938     42,553   23.500      5,319  188.000      1,330  752.000        332 3008.000
 47:   333,333    3.000     41,667   24.000      5,208  192.000      1,302  768.000        326 3072.000
 48:   326,531    3.063     40,816   24.500      5,102  196.000      1,276  784.000        319 3136.000
```

49:	320,000	3.125	40,000	25.000	5,000	200.000	1,250	800.000	313	3200.000
50:	313,725	3.188	39,216	25.500	4,902	204.000	1,225	816.000	306	3264.000
51:	307,692	3.250	38,462	26.000	4,808	208.000	1,202	832.000	300	3328.000
52:	301,887	3.313	37,736	26.500	4,717	212.000	1,179	848.000	295	3392.000
53:	296,296	3.375	37,037	27.000	4,630	216.000	1,157	864.000	289	3456.000
54:	290,909	3.437	36,364	27.500	4,545	220.000	1,136	880.000	284	3520.000
55:	285,714	3.500	35,714	28.000	4,464	224.000	1,116	896.000	279	3584.000
56:	280,702	3.562	35,088	28.500	4,386	228.000	1,096	912.000	274	3648.000
57:	275,862	3.625	34,483	29.000	4,310	232.000	1,078	928.000	269	3712.000
58:	271,186	3.688	33,898	29.500	4,237	236.000	1,059	944.000	265	3776.000
59:	266,667	3.750	33,333	30.000	4,167	240.000	1,042	960.000	260	3840.000
60:	262,295	3.813	32,787	30.500	4,098	244.000	1,025	976.000	256	3904.000
61:	258,065	3.875	32,258	31.000	4,032	248.000	1,008	992.000	252	3968.000
62:	253,968	3.938	31,746	31.500	3,968	252.000	992	1008.000	248	4032.000
63:	250,000	4.000	31,250	32.000	3,906	256.000	977	1024.000	244	4096.000
64:	246,154	4.063	30,769	32.500	3,846	260.000	962	1040.000	240	4160.000
65:	242,424	4.125	30,303	33.000	3,788	264.000	947	1056.000	237	4224.000
66:	238,806	4.188	29,851	33.500	3,731	268.000	933	1072.000	233	4288.000
67:	235,294	4.250	29,412	34.000	3,676	272.000	919	1088.000	230	4352.000
68:	231,884	4.313	28,986	34.500	3,623	276.000	906	1104.000	226	4416.000
69:	228,571	4.375	28,571	35.000	3,571	280.000	893	1120.000	223	4480.000
70:	225,352	4.437	28,169	35.500	3,521	284.000	880	1136.000	220	4544.000
71:	222,222	4.500	27,778	36.000	3,472	288.000	868	1152.000	217	4608.000
72:	219,178	4.563	27,397	36.500	3,425	292.000	856	1168.000	214	4672.000
73:	216,216	4.625	27,027	37.000	3,378	296.000	845	1184.000	211	4736.000
74:	213,333	4.688	26,667	37.500	3,333	300.000	833	1200.000	208	4800.000
75:	210,526	4.750	26,316	38.000	3,289	304.000	822	1216.000	206	4864.000
76:	207,792	4.813	25,974	38.500	3,247	308.000	812	1232.000	203	4928.000
77:	205,128	4.875	25,641	39.000	3,205	312.000	801	1248.000	200	4992.000
78:	202,532	4.938	25,316	39.500	3,165	316.000	791	1264.000	198	5056.000
79:	200,000	5.000	25,000	40.000	3,125	320.000	781	1280.000	195	5120.000
80:	197,531	5.062	24,691	40.500	3,086	324.000	772	1296.000	193	5184.000
81:	195,122	5.125	24,390	41.000	3,049	328.000	762	1312.000	191	5248.000
82:	192,771	5.188	24,096	41.500	3,012	332.000	753	1328.000	188	5312.000
83:	190,476	5.250	23,810	42.000	2,976	336.000	744	1344.000	186	5376.000
84:	188,235	5.312	23,529	42.500	2,941	340.000	735	1360.000	184	5440.000
85:	186,047	5.375	23,256	43.000	2,907	344.000	727	1376.000	182	5504.000
86:	183,908	5.438	22,989	43.500	2,874	348.000	718	1392.000	180	5568.000
87:	181,818	5.500	22,727	44.000	2,841	352.000	710	1408.000	178	5632.000
88:	179,775	5.563	22,472	44.500	2,809	356.000	702	1424.000	176	5696.000
89:	177,778	5.625	22,222	45.000	2,778	360.000	694	1440.000	174	5760.000
90:	175,824	5.687	21,978	45.500	2,747	364.000	687	1456.000	172	5824.000
91:	173,913	5.750	21,739	46.000	2,717	368.000	679	1472.000	170	5888.000
92:	172,043	5.813	21,505	46.500	2,688	372.000	672	1488.000	168	5952.000
93:	170,213	5.875	21,277	47.000	2,660	376.000	665	1504.000	166	6016.000
94:	168,421	5.937	21,053	47.500	2,632	380.000	658	1520.000	164	6080.000
95:	166,667	6.000	20,833	48.000	2,604	384.000	651	1536.000	163	6144.000
96:	164,948	6.063	20,619	48.500	2,577	388.000	644	1552.000	161	6208.000
97:	163,265	6.125	20,408	49.000	2,551	392.000	638	1568.000	159	6272.000
98:	161,616	6.188	20,202	49.500	2,525	396.000	631	1584.000	158	6336.000
99:	160,000	6.250	20,000	50.000	2,500	400.000	625	1600.000	156	6400.000
100:	158,416	6.313	19,802	50.500	2,475	404.000	619	1616.000	155	6464.000
101:	156,863	6.375	19,608	51.000	2,451	408.000	613	1632.000	153	6528.000
102:	155,340	6.438	19,417	51.500	2,427	412.000	607	1648.000	152	6592.000
103:	153,846	6.500	19,231	52.000	2,404	416.000	601	1664.000	150	6656.000
104:	152,381	6.562	19,048	52.500	2,381	420.000	595	1680.000	149	6720.000
105:	150,943	6.625	18,868	53.000	2,358	424.000	590	1696.000	147	6784.000
106:	149,533	6.688	18,692	53.500	2,336	428.000	584	1712.000	146	6848.000
107:	148,148	6.750	18,519	54.000	2,315	432.000	579	1728.000	145	6912.000
108:	146,789	6.813	18,349	54.500	2,294	436.000	573	1744.000	143	6976.000
109:	145,455	6.875	18,182	55.000	2,273	440.000	568	1760.000	142	7040.000
110:	144,144	6.938	18,018	55.500	2,252	444.000	563	1776.000	141	7104.000
111:	142,857	7.000	17,857	56.000	2,232	448.000	558	1792.000	140	7168.000
112:	141,593	7.063	17,699	56.500	2,212	452.000	553	1808.000	138	7232.000
113:	140,351	7.125	17,544	57.000	2,193	456.000	548	1824.000	137	7296.000
114:	139,130	7.187	17,391	57.500	2,174	460.000	543	1840.000	136	7360.000
115:	137,931	7.250	17,241	58.000	2,155	464.000	539	1856.000	135	7424.000
116:	136,752	7.313	17,094	58.500	2,137	468.000	534	1872.000	134	7488.000
117:	135,593	7.375	16,949	59.000	2,119	472.000	530	1888.000	132	7552.000
118:	134,454	7.437	16,807	59.500	2,101	476.000	525	1904.000	131	7616.000
119:	133,333	7.500	16,667	60.000	2,083	480.000	521	1920.000	130	7680.000
120:	132,231	7.563	16,529	60.500	2,066	484.000	517	1936.000	129	7744.000
121:	131,148	7.625	16,393	61.000	2,049	488.000	512	1952.000	128	7808.000
122:	130,081	7.687	16,260	61.500	2,033	492.000	508	1968.000	127	7872.000
123:	129,032	7.750	16,129	62.000	2,016	496.000	504	1984.000	126	7936.000
124:	128,000	7.813	16,000	62.500	2,000	500.000	500	2000.000	125	8000.000
125:	126,984	7.875	15,873	63.000	1,984	504.000	496	2016.000	124	8064.000
126:	125,984	7.938	15,748	63.500	1,969	508.000	492	2032.000	123	8128.000
127:	125,000	8.000	15,625	64.000	1,953	512.000	488	2048.000	122	8192.000
128:	124,031	8.063	15,504	64.500	1,938	516.000	484	2064.000	121	8256.000
129:	123,077	8.125	15,385	65.000	1,923	520.000	481	2080.000	120	8320.000
130:	122,137	8.188	15,267	65.500	1,908	524.000	477	2096.000	119	8384.000
131:	121,212	8.250	15,152	66.000	1,894	528.000	473	2112.000	118	8448.000
132:	120,301	8.313	15,038	66.500	1,880	532.000	470	2128.000	117	8512.000
133:	119,403	8.375	14,925	67.000	1,866	536.000	466	2144.000	117	8576.000
134:	118,519	8.438	14,815	67.500	1,852	540.000	463	2160.000	116	8640.000
135:	117,647	8.500	14,706	68.000	1,838	544.000	460	2176.000	115	8704.000
136:	116,788	8.563	14,599	68.500	1,825	548.000	456	2192.000	114	8768.000
137:	115,942	8.625	14,493	69.000	1,812	552.000	453	2208.000	113	8832.000
138:	115,108	8.688	14,388	69.500	1,799	556.000	450	2224.000	112	8896.000
139:	114,286	8.750	14,286	70.000	1,786	560.000	446	2240.000	112	8960.000
140:	113,475	8.812	14,184	70.500	1,773	564.000	443	2256.000	111	9024.000
141:	112,676	8.875	14,085	71.000	1,761	568.000	440	2272.000	110	9088.000
142:	111,888	8.938	13,986	71.500	1,748	572.000	437	2288.000	109	9152.000
143:	111,111	9.000	13,889	72.000	1,736	576.000	434	2304.000	109	9216.000
144:	110,345	9.063	13,793	72.500	1,724	580.000	431	2320.000	108	9280.000
145:	109,589	9.125	13,699	73.000	1,712	584.000	428	2336.000	107	9344.000
146:	108,844	9.188	13,605	73.500	1,701	588.000	425	2352.000	106	9408.000
147:	108,108	9.250	13,514	74.000	1,689	592.000	422	2368.000	106	9472.000
148:	107,383	9.313	13,423	74.500	1,678	596.000	419	2384.000	105	9536.000
149:	106,667	9.375	13,333	75.000	1,667	600.000	417	2400.000	104	9600.000

150:	105,960	9.437	13,245	75.500	1,656	604.000	414	2416.000	103	9664.000
151:	105,263	9.500	13,158	76.000	1,645	608.000	411	2432.000	103	9728.000
152:	104,575	9.563	13,072	76.500	1,634	612.000	408	2448.000	102	9792.000
153:	103,896	9.625	12,987	77.000	1,623	616.000	406	2464.000	101	9856.000
154:	103,226	9.688	12,903	77.500	1,613	620.000	403	2480.000	101	9920.000
155:	102,564	9.750	12,821	78.000	1,603	624.000	401	2496.000	100	9984.000
156:	101,911	9.813	12,739	78.500	1,592	628.000	398	2512.000	100	10048.000
157:	101,266	9.875	12,658	79.000	1,582	632.000	396	2528.000	99	10112.000
158:	100,629	9.938	12,579	79.500	1,572	636.000	393	2544.000	98	10176.000
159:	100,000	10.000	12,500	80.000	1,563	640.000	391	2560.000	98	10240.000
160:	99,379	10.062	12,422	80.500	1,553	644.000	388	2576.000	97	10304.000
161:	98,765	10.125	12,346	81.000	1,543	648.000	386	2592.000	96	10368.000
162:	98,160	10.188	12,270	81.500	1,534	652.000	383	2608.000	96	10432.000
163:	97,561	10.250	12,195	82.000	1,524	656.000	381	2624.000	95	10496.000
164:	96,970	10.313	12,121	82.500	1,515	660.000	379	2640.000	95	10560.000
165:	96,386	10.375	12,048	83.000	1,506	664.000	377	2656.000	94	10624.000
166:	95,808	10.438	11,976	83.500	1,497	668.000	374	2672.000	94	10688.000
167:	95,238	10.500	11,905	84.000	1,488	672.000	372	2688.000	93	10752.000
168:	94,675	10.563	11,834	84.500	1,479	676.000	370	2704.000	92	10816.000
169:	94,118	10.625	11,765	85.000	1,471	680.000	368	2720.000	92	10880.000
170:	93,567	10.687	11,696	85.500	1,462	684.000	365	2736.000	91	10944.000
171:	93,023	10.750	11,628	86.000	1,453	688.000	363	2752.000	91	11008.000
172:	92,486	10.813	11,561	86.500	1,445	692.000	361	2768.000	90	11072.000
173:	91,954	10.875	11,494	87.000	1,437	696.000	359	2784.000	90	11136.000
174:	91,429	10.938	11,429	87.500	1,429	700.000	357	2800.000	89	11200.000
175:	90,909	11.000	11,364	88.000	1,420	704.000	355	2816.000	89	11264.000
176:	90,395	11.063	11,299	88.500	1,412	708.000	353	2832.000	88	11328.000
177:	89,888	11.125	11,236	89.000	1,404	712.000	351	2848.000	88	11392.000
178:	89,385	11.188	11,173	89.500	1,397	716.000	349	2864.000	87	11456.000
179:	88,889	11.250	11,111	90.000	1,389	720.000	347	2880.000	87	11520.000
180:	88,398	11.312	11,050	90.500	1,381	724.000	345	2896.000	86	11584.000
181:	87,912	11.375	10,989	91.000	1,374	728.000	343	2912.000	86	11648.000
182:	87,432	11.438	10,929	91.500	1,366	732.000	342	2928.000	85	11712.000
183:	86,957	11.500	10,870	92.000	1,359	736.000	340	2944.000	85	11776.000
184:	86,486	11.563	10,811	92.500	1,351	740.000	338	2960.000	84	11840.000
185:	86,022	11.625	10,753	93.000	1,344	744.000	336	2976.000	84	11904.000
186:	85,561	11.688	10,695	93.500	1,337	748.000	334	2992.000	84	11968.000
187:	85,106	11.750	10,638	94.000	1,330	752.000	332	3008.000	83	12032.000
188:	84,656	11.813	10,582	94.500	1,323	756.000	331	3024.000	83	12096.000
189:	84,211	11.875	10,526	95.000	1,316	760.000	329	3040.000	82	12160.000
190:	83,770	11.937	10,471	95.500	1,309	764.000	327	3056.000	82	12224.000
191:	83,333	12.000	10,417	96.000	1,302	768.000	326	3072.000	81	12288.000
192:	82,902	12.063	10,363	96.500	1,295	772.000	324	3088.000	81	12352.000
193:	82,474	12.125	10,309	97.000	1,289	776.000	322	3104.000	81	12416.000
194:	82,051	12.188	10,256	97.500	1,282	780.000	321	3120.000	80	12480.000
195:	81,633	12.250	10,204	98.000	1,276	784.000	319	3136.000	80	12544.000
196:	81,218	12.313	10,152	98.500	1,269	788.000	317	3152.000	79	12608.000
197:	80,808	12.375	10,101	99.000	1,263	792.000	316	3168.000	79	12672.000
198:	80,402	12.437	10,050	99.500	1,256	796.000	314	3184.000	79	12736.000
199:	80,000	12.500	10,000	100.000	1,250	800.000	313	3200.000	78	12800.000
200:	79,602	12.562	9,950	100.500	1,244	804.000	311	3216.000	78	12864.000
201:	79,208	12.625	9,901	101.000	1,238	808.000	309	3232.000	77	12928.000
202:	78,818	12.688	9,852	101.500	1,232	812.000	308	3248.000	77	12992.000
203:	78,431	12.750	9,804	102.000	1,225	816.000	306	3264.000	77	13056.000
204:	78,049	12.813	9,756	102.500	1,220	820.000	305	3280.000	76	13120.000
205:	77,670	12.875	9,709	103.000	1,214	824.000	303	3296.000	76	13184.000
206:	77,295	12.938	9,662	103.500	1,208	828.000	302	3312.000	75	13248.000
207:	76,923	13.000	9,615	104.000	1,202	832.000	300	3328.000	75	13312.000
208:	76,555	13.062	9,569	104.500	1,196	836.000	299	3344.000	75	13376.000
209:	76,190	13.125	9,524	105.000	1,190	840.000	298	3360.000	74	13440.000
210:	75,829	13.187	9,479	105.500	1,185	844.000	296	3376.000	74	13504.000
211:	75,472	13.250	9,434	106.000	1,179	848.000	295	3392.000	74	13568.000
212:	75,117	13.313	9,390	106.500	1,174	852.000	293	3408.000	73	13632.000
213:	74,766	13.375	9,346	107.000	1,168	856.000	292	3424.000	73	13696.000
214:	74,419	13.438	9,302	107.500	1,163	860.000	291	3440.000	73	13760.000
215:	74,074	13.500	9,259	108.000	1,157	864.000	289	3456.000	72	13824.000
216:	73,733	13.563	9,217	108.500	1,152	868.000	288	3472.000	72	13888.000
217:	73,394	13.625	9,174	109.000	1,147	872.000	287	3488.000	72	13952.000
218:	73,059	13.687	9,132	109.500	1,142	876.000	285	3504.000	71	14016.000
219:	72,727	13.750	9,091	110.000	1,136	880.000	284	3520.000	71	14080.000
220:	72,398	13.812	9,050	110.500	1,131	884.000	283	3536.000	71	14144.000
221:	72,072	13.875	9,009	111.000	1,126	888.000	282	3552.000	70	14208.000
222:	71,749	13.938	8,969	111.500	1,121	892.000	280	3568.000	70	14272.000
223:	71,429	14.000	8,929	112.000	1,116	896.000	279	3584.000	70	14336.000
224:	71,111	14.063	8,889	112.500	1,111	900.000	278	3600.000	69	14400.000
225:	70,796	14.125	8,850	113.000	1,106	904.000	277	3616.000	69	14464.000
226:	70,485	14.188	8,811	113.500	1,101	908.000	275	3632.000	69	14528.000
227:	70,175	14.250	8,772	114.000	1,096	912.000	274	3648.000	69	14592.000
228:	69,869	14.312	8,734	114.500	1,092	916.000	273	3664.000	68	14656.000
229:	69,565	14.375	8,696	115.000	1,087	920.000	272	3680.000	68	14720.000
230:	69,264	14.437	8,658	115.500	1,082	924.000	271	3696.000	68	14784.000
231:	68,966	14.500	8,621	116.000	1,078	928.000	269	3712.000	67	14848.000
232:	68,670	14.563	8,584	116.500	1,073	932.000	268	3728.000	67	14912.000
233:	68,376	14.625	8,547	117.000	1,068	936.000	267	3744.000	67	14976.000
234:	68,085	14.688	8,511	117.500	1,064	940.000	266	3760.000	66	15040.000
235:	67,797	14.750	8,475	118.000	1,059	944.000	265	3776.000	66	15104.000
236:	67,511	14.813	8,439	118.500	1,055	948.000	264	3792.000	66	15168.000
237:	67,227	14.875	8,403	119.000	1,050	952.000	263	3808.000	66	15232.000
238:	66,946	14.937	8,368	119.500	1,046	956.000	262	3824.000	65	15296.000
239:	66,667	15.000	8,333	120.000	1,042	960.000	260	3840.000	65	15360.000
240:	66,390	15.062	8,299	120.500	1,037	964.000	259	3856.000	65	15424.000
241:	66,116	15.125	8,264	121.000	1,033	968.000	258	3872.000	65	15488.000
242:	65,844	15.188	8,230	121.500	1,029	972.000	257	3888.000	64	15552.000
243:	65,574	15.250	8,197	122.000	1,025	976.000	256	3904.000	64	15616.000
244:	65,306	15.313	8,163	122.500	1,020	980.000	255	3920.000	64	15680.000
245:	65,041	15.375	8,130	123.000	1,016	984.000	254	3936.000	64	15744.000
246:	64,777	15.438	8,097	123.500	1,012	988.000	253	3952.000	63	15808.000
247:	64,516	15.500	8,065	124.000	1,008	992.000	252	3968.000	63	15872.000
248:	64,257	15.562	8,032	124.500	1,004	996.000	251	3984.000	63	15936.000
249:	64,000	15.625	8,000	125.000	1,000	1000.000	250	4000.000	63	16000.000
250:	63,745	15.687	7,968	125.500	996	1004.000	249	4016.000	62	16064.000

```
251:    63,492   15.750      7,937  126.000        992 1008.000        248 4032.000         62 16128.000
252:    63,241   15.812      7,905  126.500        988 1012.000        247 4048.000         62 16192.000
253:    62,992   15.875      7,874  127.000        984 1016.000        246 4064.000         62 16256.000
254:    62,745   15.937      7,843  127.500        980 1020.000        245 4080.000         61 16320.000
255:    62,500   16.000      7,813  128.000        977 1024.000        244 4096.000         61 16384.000
```

* = may not work reliably, testing shows.

Example code which uses a prescaler of one (no prescaler):

```
const byte OUTPUT_PIN = 3;  // Timer 2 "B" output: OC2B

const byte n = 224;  // for example, 71.111 kHz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS20);         // fast PWM, no prescaler
  OCR2A =  n;                                // from table
  OCR2B = ((n + 1) / 2) - 1;                 // 50% duty cycle
  }  // end of setup

void loop() { }
```

Example code which uses a prescaler of 8:

```
const byte OUTPUT_PIN = 3;  // Timer 2 "B" output: OC2B

const byte n = 224;  // for example, 8.89 kHz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS21);         // fast PWM, prescaler of 8
  OCR2A =  n;                                // from table
  OCR2B = ((n + 1) / 2) - 1;                 // 50% duty cycle
  }  // end of setup

void loop() { }
```

Example code which uses a prescaler of 64:

```
const byte OUTPUT_PIN = 3;  // Timer 2 "B" output: OC2B

const byte n = 224;  // for example, 1.111 kHz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS22);         // fast PWM, prescaler of 64
  OCR2A =  n;                                // from table
  OCR2B = ((n + 1) / 2) - 1;                 // 50% duty cycle
  }  // end of setup

void loop() { }
```

Example code which uses a prescaler of 256:

```
const byte OUTPUT_PIN = 3;  // Timer 2 "B" output: OC2B

const byte n = 224;  // for example, 278 Hz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS21) | bit (CS22);         // fast PWM, prescaler of 256
  OCR2A =  n;                                // from table
  OCR2B = ((n + 1) / 2) - 1;                 // 50% duty cycle
  }  // end of setup

void loop() { }
```

Example code which uses a prescaler of 1024:

```
const byte OUTPUT_PIN = 3;  // Timer 2 "B" output: OC2B

const byte n = 224;  // for example, 69 Hz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS20) | bit (CS21) | bit (CS22);         // fast PWM, prescaler of 1024
  OCR2A =  n;                                // from table
  OCR2B = ((n + 1) / 2) - 1;                 // 50% duty cycle
  }  // end of setup

void loop() { }
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,250 posts) bio *Forum Administrator*

Date Reply #8 on Sat 24 Nov 2012 01:46 AM (UTC)

Amended on Wed 04 Sep 2013 04:28 AM (UTC) by Nick Gammon

Message

Additional frequencies and periods for Timer 2

Timer 2 also offers the prescalers of 32 and 128. On timers 0 and 1 these "clock source" settings are used for an external clock (rising or falling edge).

```
  n  -- Prescale 32 --  -- Prescale 128 --
     Freq (Hz) Per (uS)  Freq (Hz) Per (uS)
  1:   250,000    4.000     62,500   16.000
  2:   166,667    6.000     41,667   24.000
  3:   125,000    8.000     31,250   32.000
  4:   100,000   10.000     25,000   40.000
  5:    83,333   12.000     20,833   48.000
  6:    71,429   14.000     17,857   56.000
  7:    62,500   16.000     15,625   64.000
  8:    55,556   18.000     13,889   72.000
  9:    50,000   20.000     12,500   80.000
 10:    45,455   22.000     11,364   88.000
 11:    41,667   24.000     10,417   96.000
 12:    38,462   26.000      9,615  104.000
 13:    35,714   28.000      8,929  112.000
 14:    33,333   30.000      8,333  120.000
 15:    31,250   32.000      7,813  128.000
 16:    29,412   34.000      7,353  136.000
 17:    27,778   36.000      6,944  144.000
 18:    26,316   38.000      6,579  152.000
 19:    25,000   40.000      6,250  160.000
 20:    23,810   42.000      5,952  168.000
```

```
 21:   22,727  44.000      5,682  176.000
 22:   21,739  46.000      5,435  184.000
 23:   20,833  48.000      5,208  192.000
 24:   20,000  50.000      5,000  200.000
 25:   19,231  52.000      4,808  208.000
 26:   18,519  54.000      4,630  216.000
 27:   17,857  56.000      4,464  224.000
 28:   17,241  58.000      4,310  232.000
 29:   16,667  60.000      4,167  240.000
 30:   16,129  62.000      4,032  248.000
 31:   15,625  64.000      3,906  256.000
 32:   15,152  66.000      3,788  264.000
 33:   14,706  68.000      3,676  272.000
 34:   14,286  70.000      3,571  280.000
 35:   13,889  72.000      3,472  288.000
 36:   13,514  74.000      3,378  296.000
 37:   13,158  76.000      3,289  304.000
 38:   12,821  78.000      3,205  312.000
 39:   12,500  80.000      3,125  320.000
 40:   12,195  82.000      3,049  328.000
 41:   11,905  84.000      2,976  336.000
 42:   11,628  86.000      2,907  344.000
 43:   11,364  88.000      2,841  352.000
 44:   11,111  90.000      2,778  360.000
 45:   10,870  92.000      2,717  368.000
 46:   10,638  94.000      2,660  376.000
 47:   10,417  96.000      2,604  384.000
 48:   10,204  98.000      2,551  392.000
 49:   10,000 100.000      2,500  400.000
 50:    9,804 102.000      2,451  408.000
 51:    9,615 104.000      2,404  416.000
 52:    9,434 106.000      2,358  424.000
 53:    9,259 108.000      2,315  432.000
 54:    9,091 110.000      2,273  440.000
 55:    8,929 112.000      2,232  448.000
 56:    8,772 114.000      2,193  456.000
 57:    8,621 116.000      2,155  464.000
 58:    8,475 118.000      2,119  472.000
 59:    8,333 120.000      2,083  480.000
 60:    8,197 122.000      2,049  488.000
 61:    8,065 124.000      2,016  496.000
 62:    7,937 126.000      1,984  504.000
 63:    7,813 128.000      1,953  512.000
 64:    7,692 130.000      1,923  520.000
 65:    7,576 132.000      1,894  528.000
 66:    7,463 134.000      1,866  536.000
 67:    7,353 136.000      1,838  544.000
 68:    7,246 138.000      1,812  552.000
 69:    7,143 140.000      1,786  560.000
 70:    7,042 142.000      1,761  568.000
 71:    6,944 144.000      1,736  576.000
 72:    6,849 146.000      1,712  584.000
 73:    6,757 148.000      1,689  592.000
 74:    6,667 150.000      1,667  600.000
 75:    6,579 152.000      1,645  608.000
 76:    6,494 154.000      1,623  616.000
 77:    6,410 156.000      1,603  624.000
 78:    6,329 158.000      1,582  632.000
 79:    6,250 160.000      1,563  640.000
 80:    6,173 162.000      1,543  648.000
 81:    6,098 164.000      1,524  656.000
 82:    6,024 166.000      1,506  664.000
 83:    5,952 168.000      1,488  672.000
 84:    5,882 170.000      1,471  680.000
 85:    5,814 172.000      1,453  688.000
 86:    5,747 174.000      1,437  696.000
 87:    5,682 176.000      1,420  704.000
 88:    5,618 178.000      1,404  712.000
 89:    5,556 180.000      1,389  720.000
 90:    5,495 182.000      1,374  728.000
 91:    5,435 184.000      1,359  736.000
 92:    5,376 186.000      1,344  744.000
 93:    5,319 188.000      1,330  752.000
 94:    5,263 190.000      1,316  760.000
 95:    5,208 192.000      1,302  768.000
 96:    5,155 194.000      1,289  776.000
 97:    5,102 196.000      1,276  784.000
 98:    5,051 198.000      1,263  792.000
 99:    5,000 200.000      1,250  800.000
100:    4,950 202.000      1,238  808.000
101:    4,902 204.000      1,225  816.000
102:    4,854 206.000      1,214  824.000
103:    4,808 208.000      1,202  832.000
104:    4,762 210.000      1,190  840.000
105:    4,717 212.000      1,179  848.000
106:    4,673 214.000      1,168  856.000
107:    4,630 216.000      1,157  864.000
108:    4,587 218.000      1,147  872.000
109:    4,545 220.000      1,136  880.000
110:    4,505 222.000      1,126  888.000
111:    4,464 224.000      1,116  896.000
112:    4,425 226.000      1,106  904.000
113:    4,386 228.000      1,096  912.000
114:    4,348 230.000      1,087  920.000
115:    4,310 232.000      1,078  928.000
116:    4,274 234.000      1,068  936.000
117:    4,237 236.000      1,059  944.000
118:    4,202 238.000      1,050  952.000
119:    4,167 240.000      1,042  960.000
120:    4,132 242.000      1,033  968.000
121:    4,098 244.000      1,025  976.000
```

```
122:    4,065  246.000     1,016  984.000
123:    4,032  248.000     1,008  992.000
124:    4,000  250.000     1,000 1000.000
125:    3,968  252.000       992 1008.000
126:    3,937  254.000       984 1016.000
127:    3,906  256.000       977 1024.000
128:    3,876  258.000       969 1032.000
129:    3,846  260.000       962 1040.000
130:    3,817  262.000       954 1048.000
131:    3,788  264.000       947 1056.000
132:    3,759  266.000       940 1064.000
133:    3,731  268.000       933 1072.000
134:    3,704  270.000       926 1080.000
135:    3,676  272.000       919 1088.000
136:    3,650  274.000       912 1096.000
137:    3,623  276.000       906 1104.000
138:    3,597  278.000       899 1112.000
139:    3,571  280.000       893 1120.000
140:    3,546  282.000       887 1128.000
141:    3,521  284.000       880 1136.000
142:    3,497  286.000       874 1144.000
143:    3,472  288.000       868 1152.000
144:    3,448  290.000       862 1160.000
145:    3,425  292.000       856 1168.000
146:    3,401  294.000       850 1176.000
147:    3,378  296.000       845 1184.000
148:    3,356  298.000       839 1192.000
149:    3,333  300.000       833 1200.000
150:    3,311  302.000       828 1208.000
151:    3,289  304.000       822 1216.000
152:    3,268  306.000       817 1224.000
153:    3,247  308.000       812 1232.000
154:    3,226  310.000       806 1240.000
155:    3,205  312.000       801 1248.000
156:    3,185  314.000       796 1256.000
157:    3,165  316.000       791 1264.000
158:    3,145  318.000       786 1272.000
159:    3,125  320.000       781 1280.000
160:    3,106  322.000       776 1288.000
161:    3,086  324.000       772 1296.000
162:    3,067  326.000       767 1304.000
163:    3,049  328.000       762 1312.000
164:    3,030  330.000       758 1320.000
165:    3,012  332.000       753 1328.000
166:    2,994  334.000       749 1336.000
167:    2,976  336.000       744 1344.000
168:    2,959  338.000       740 1352.000
169:    2,941  340.000       735 1360.000
170:    2,924  342.000       731 1368.000
171:    2,907  344.000       727 1376.000
172:    2,890  346.000       723 1384.000
173:    2,874  348.000       718 1392.000
174:    2,857  350.000       714 1400.000
175:    2,841  352.000       710 1408.000
176:    2,825  354.000       706 1416.000
177:    2,809  356.000       702 1424.000
178:    2,793  358.000       698 1432.000
179:    2,778  360.000       694 1440.000
180:    2,762  362.000       691 1448.000
181:    2,747  364.000       687 1456.000
182:    2,732  366.000       683 1464.000
183:    2,717  368.000       679 1472.000
184:    2,703  370.000       676 1480.000
185:    2,688  372.000       672 1488.000
186:    2,674  374.000       668 1496.000
187:    2,660  376.000       665 1504.000
188:    2,646  378.000       661 1512.000
189:    2,632  380.000       658 1520.000
190:    2,618  382.000       654 1528.000
191:    2,604  384.000       651 1536.000
192:    2,591  386.000       648 1544.000
193:    2,577  388.000       644 1552.000
194:    2,564  390.000       641 1560.000
195:    2,551  392.000       638 1568.000
196:    2,538  394.000       635 1576.000
197:    2,525  396.000       631 1584.000
198:    2,513  398.000       628 1592.000
199:    2,500  400.000       625 1600.000
200:    2,488  402.000       622 1608.000
201:    2,475  404.000       619 1616.000
202:    2,463  406.000       616 1624.000
203:    2,451  408.000       613 1632.000
204:    2,439  410.000       610 1640.000
205:    2,427  412.000       607 1648.000
206:    2,415  414.000       604 1656.000
207:    2,404  416.000       601 1664.000
208:    2,392  418.000       598 1672.000
209:    2,381  420.000       595 1680.000
210:    2,370  422.000       592 1688.000
211:    2,358  424.000       590 1696.000
212:    2,347  426.000       587 1704.000
213:    2,336  428.000       584 1712.000
214:    2,326  430.000       581 1720.000
215:    2,315  432.000       579 1728.000
216:    2,304  434.000       576 1736.000
217:    2,294  436.000       573 1744.000
218:    2,283  438.000       571 1752.000
219:    2,273  440.000       568 1760.000
220:    2,262  442.000       566 1768.000
221:    2,252  444.000       563 1776.000
222:    2,242  446.000       561 1784.000
```

```
223:     2,232  448.000        558 1792.000
224:     2,222  450.000        556 1800.000
225:     2,212  452.000        553 1808.000
226:     2,203  454.000        551 1816.000
227:     2,193  456.000        548 1824.000
228:     2,183  458.000        546 1832.000
229:     2,174  460.000        543 1840.000
230:     2,165  462.000        541 1848.000
231:     2,155  464.000        539 1856.000
232:     2,146  466.000        536 1864.000
233:     2,137  468.000        534 1872.000
234:     2,128  470.000        532 1880.000
235:     2,119  472.000        530 1888.000
236:     2,110  474.000        527 1896.000
237:     2,101  476.000        525 1904.000
238:     2,092  478.000        523 1912.000
239:     2,083  480.000        521 1920.000
240:     2,075  482.000        519 1928.000
241:     2,066  484.000        517 1936.000
242:     2,058  486.000        514 1944.000
243:     2,049  488.000        512 1952.000
244:     2,041  490.000        510 1960.000
245:     2,033  492.000        508 1968.000
246:     2,024  494.000        506 1976.000
247:     2,016  496.000        504 1984.000
248:     2,008  498.000        502 1992.000
249:     2,000  500.000        500 2000.000
250:     1,992  502.000        498 2008.000
251:     1,984  504.000        496 2016.000
252:     1,976  506.000        494 2024.000
253:     1,969  508.000        492 2032.000
254:     1,961  510.000        490 2040.000
255:     1,953  512.000        488 2048.000
```

Example code which uses a prescaler of 32:

```
const byte OUTPUT_PIN = 3;  // Timer 2 "B" output: OC2B

const byte n = 224;  // for example, 2.222 kHz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS20) | bit (CS21);         // fast PWM, prescaler of 32
  OCR2A =  n;                                 // from table
  OCR2B = ((n + 1) / 2) - 1;                  // 50% duty cycle
  }  // end of setup

void loop() { }
```

Example code which uses a prescaler of 128:

```
const byte OUTPUT_PIN = 3;  // Timer 2 "B" output: OC2B

const byte n = 224;  // for example, 556 Hz

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

  TCCR2A = bit (WGM20) | bit (WGM21) | bit (COM2B1); // fast PWM, clear OC2A on compare
  TCCR2B = bit (WGM22) | bit (CS20) | bit (CS22);         // fast PWM, prescaler of 128
  OCR2A =  n;                                 // from table
  OCR2B = ((n + 1) / 2) - 1;                  // 50% duty cycle
  }  // end of setup

void loop() { }
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,250 posts) bio Forum Administrator

Date Reply #9 on Thu 29 Nov 2012 03:48 AM (UTC)

Amended on Wed 04 Sep 2013 04:29 AM (UTC) by Nick Gammon

Message

Attiny85 example

The code below was developed in answer to a question on the Arduino forum. It shows how you can have 3 x PWM outputs on an Attiny85 chip:

```
// For Attiny85
// Author: Nick Gammon
// Date: 29 November 2012

void setup()
 {
  pinMode (0, OUTPUT);  // chip pin 5  // OC0A - Timer 0 "A"
  pinMode (1, OUTPUT);  // chip pin 6  // OC0B - Timer 0 "B"
  pinMode (4, OUTPUT);  // chip pin 3  // OC1B - Timer 1 "B"

  // Timer 0, A side
  TCCR0A = bit (WGM00) | bit (WGM01) | bit (COM0A1); // fast PWM, clear OC0A on compare
  TCCR0B = bit (CS00);           // fast PWM, top at 0xFF, no prescaler
  OCR0A = 127;                   // duty cycle (50%)

  // Timer 0, B side
  TCCR0A |= bit (COM0B1);        // clear OC0B on compare
  OCR0B = 63;                    // duty cycle (25%)

  // Timer 1
  TCCR1 = bit (CS10);           // no prescaler
  GTCCR = bit (COM1B1) | bit (PWM1B);  //  clear OC1B on compare
  OCR1B = 31;                   // duty cycle (25%)
  OCR1C = 127;                  // frequency
  }  // end of setup

void loop() { }
```

Timer 0 is set as fast PWM, counting up to 0xFF (255). Timer 1 is also PWM, counting up to 127 (OCR1C). This means you can adjust the frequency of Timer 1.

Logic analyzer output on a processor running at 8 MHz:

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,250 posts) bio Forum Administrator

Date Reply #10 on Fri 01 Feb 2013 05:32 AM (UTC)

Amended on Wed 04 Sep 2013 04:29 AM (UTC) by Nick Gammon

Message

Asynchronous timer example

You can clock timer 2 "asynchronously". What this means is, that if you use the internal oscillator for the processor (ie. run at 8 MHz) then the clock input pins (XTAL1 and XTAL1, which are pins 9 and 10 on the chip) can have a different crystal on them, as a clock input. For example, a 32.768 kHz "clock" crystal.

The advantage of a 32.768 crystal is that, by dividing it by 32768, you get exactly one second per timer firing (less or more by playing with the timer counter interval).

This lets you sleep for an accurate interval (more accurate than the watchdog timer), and indeed, a long interval, like 8 seconds.

Whilst asleep, the processor uses very little power. In the example sketch I measured 1.46 µA current consumption, when the output was LOW, if running from 5V power supply, and 1.1 µA if running from 3.3V power supply.

```
#include <avr/sleep.h>
#include <avr/power.h>

const byte tick = 3;

// interrupt on Timer 2 compare "A" completion - does nothing
EMPTY_INTERRUPT (TIMER2_COMPA_vect);

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

  // clock input to timer 2 from XTAL1/XTAL2
  ASSR = bit (AS2);

  // set up timer 2 to count up to 32 * 1024  (32768)
  TCCR2A = bit (WGM21);                             // CTC
  TCCR2B = bit (CS20) | bit (CS21) | bit (CS22);    // Prescaler of 1024
  OCR2A =  31;              // count to 32 (zero-relative)

  // enable timer interrupts
  TIMSK2 |= bit (OCIE2A);

  // disable ADC
  ADCSRA = 0;

  // turn off everything we can
  power_adc_disable ();
  power_spi_disable();
  power_twi_disable();
  power_timer0_disable();
  power_timer1_disable();
  power_usart0_disable();

  // full power-down doesn't respond to Timer 2
  set_sleep_mode (SLEEP_MODE_PWR_SAVE);

  // get ready ...
  sleep_enable();

  }  // end of setup

void loop()
  {

  // turn off brown-out enable in software
  MCUCR = bit (BODS) | bit (BODSE);
  MCUCR = bit (BODS);

  // sleep, finally!
  sleep_cpu ();

  // we awoke! pulse the clock hand
  digitalWrite (tick, ! digitalRead (tick));

  }  // end of loop
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,250 posts) bio *Forum Administrator*

Date Reply #11 on Mon 01 Apr 2013 04:33 AM (UTC)

Message

Tone library for timers

The small library below simplifies generating tones with the hardware timers. Unlike the "Tone" library that comes with the Arduino this one directly uses the outputting capability of the hardware timers, and thus does not use interrupts.

http://www.gammon.com.au/Arduino/TonePlayer.zip

Example of use on the Uno:

```
#include <TonePlayer.h>

TonePlayer tone1 (TCCR1A, TCCR1B, OCR1AH, OCR1AL, TCNT1H, TCNT1L);  // pin D9 (Uno), D11 (Mega)

void setup()
  {
  pinMode (9, OUTPUT);  // output pin is fixed (OC1A)

  tone1.tone (220);  // 220 Hz
  delay (500);
  tone1.noTone ();

  tone1.tone (440);
  delay (500);
  tone1.noTone ();

  tone1.tone (880);
  delay (500);
  tone1.noTone ();
  }

void loop() { }
```

The library is written for the 16-bit timers, and supports Timer 1 on the Atmega328, and Timers 1, 3, 4, 5 on the Atmega2560. Comments inside the library show how to set up the other timers.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,250 posts) bio *Forum Administrator*

Date Reply #12 on Sat 31 Aug 2013 04:33 AM (UTC)

Amended on Sat 04 Jul 2015 04:43 AM (UTC) by Nick Gammon

Message

Timing an interval using the input capture unit

The code below is a modified version of the sketch above (in reply #1) that calculates a frequency by inverting the period.

In other words, it can be used to time the interval between two consecutive pulses.

This one uses the "input capture" input to capture the value in Timer 1 at the moment an event occurs. Since the hardware captures the timer value in a separate register, this eliminates the delay caused by entering the interrupt routine (itself around 2 µs).

```
// Frequency timer using input capture unit
// Author: Nick Gammon
// Date: 31 August 2013

// Input: Pin D8

volatile boolean first;
volatile boolean triggered;
volatile unsigned long overflowCount;
volatile unsigned long startTime;
volatile unsigned long finishTime;

// timer overflows (every 65536 counts)
ISR (TIMER1_OVF_vect)
{
  overflowCount++;
}  // end of TIMER1_OVF_vect

ISR (TIMER1_CAPT_vect)
  {
  // grab counter value before it changes any more
  unsigned int timer1CounterValue;
  timer1CounterValue = ICR1;  // see datasheet, page 117 (accessing 16-bit registers)
  unsigned long overflowCopy = overflowCount;

  // if just missed an overflow
  if ((TIFR1 & bit (TOV1)) && timer1CounterValue < 0x7FFF)
    overflowCopy++;

  // wait until we noticed last one
  if (triggered)
    return;

  if (first)
    {
    startTime = (overflowCopy << 16) + timer1CounterValue;
```

```
    first = false;
    return;
    }

  finishTime = (overflowCopy << 16) + timer1CounterValue;
  triggered = true;
  TIMSK1 = 0;    // no more interrupts for now
  }  // end of TIMER1_CAPT_vect

void prepareForInterrupts ()
  {
  noInterrupts ();  // protected code
  first = true;
  triggered = false;  // re-arm for next time
  // reset Timer 1
  TCCR1A = 0;
  TCCR1B = 0;

  TIFR1 = bit (ICF1) | bit (TOV1);  // clear flags so we don't get a bogus interrupt
  TCNT1 = 0;          // Counter to zero
  overflowCount = 0;  // Therefore no overflows yet

  // Timer 1 - counts clock pulses
  TIMSK1 = bit (TOIE1) | bit (ICIE1);   // interrupt on Timer 1 overflow and input capture
  // start Timer 1, no prescaler
  TCCR1B =  bit (CS10) | bit (ICES1);  // plus Input Capture Edge Select (rising on D8)
  interrupts ();
  }  // end of prepareForInterrupts


void setup ()
  {
  Serial.begin(115200);
  Serial.println("Frequency Counter");
  // set up for interrupts
  prepareForInterrupts ();
  } // end of setup

void loop ()
  {
  // wait till we have a reading
  if (!triggered)
    return;

  // period is elapsed time
  unsigned long elapsedTime = finishTime - startTime;
  // frequency is inverse of period, adjusted for clock period
  float freq = F_CPU / float (elapsedTime);  // each tick is 62.5 ns at 16 MHz

  Serial.print ("Took: ");
  Serial.print (elapsedTime);
  Serial.print (" counts. ");

  Serial.print ("Frequency: ");
  Serial.print (freq);
  Serial.println (" Hz. ");

  // so we can read it
  delay (500);

  prepareForInterrupts ();
}   // end of loop
```

The above code successfully counted a 200 kHz input. Output:

```
Took: 80 counts. Frequency: 200000.00 Hz.
Took: 80 counts. Frequency: 200000.00 Hz.
Took: 80 counts. Frequency: 200000.00 Hz.
Took: 80 counts. Frequency: 200000.00 Hz.
Took: 80 counts. Frequency: 200000.00 Hz.
```

Put another way, it captured an interval of 5 µs (80 x 62.5 ns). Since it takes about 2.5 µs to enter and leave an ISR (interrupt service routine) then this sounds about right (you need two interrupts to capture the interval: the "start of interval" interrupt and the "end of interval" interrupt).

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,250 posts) bio *Forum Administrator*

Date Reply #13 on Tue 05 Nov 2013 06:41 AM (UTC)

Amended on Sat 04 Jul 2015 04:50 AM (UTC) by Nick Gammon

Message

Measuring a duty cycle using the input capture unit

The sketch below is a variation on the frequency counter above. However this one measures the width of the "on" portion of the duty cycle of a pulse. This could be useful for decoding information from PWM style controls.

```
// Duty cycle calculation using input capture unit
// Author: Nick Gammon
// Date: 5 November 2013

// Input: Pin D8

volatile boolean first;
volatile boolean triggered;
volatile unsigned long overflowCount;
volatile unsigned long startTime;
volatile unsigned long finishTime;

// timer overflows (every 65536 counts)
ISR (TIMER1_OVF_vect)
{
  overflowCount++;
}  // end of TIMER1_OVF_vect

ISR (TIMER1_CAPT_vect)
  {
  // grab counter value before it changes any more
  unsigned int timer1CounterValue;
  timer1CounterValue = ICR1;  // see datasheet, page 117 (accessing 16-bit registers)
  unsigned long overflowCopy = overflowCount;

  // if just missed an overflow
  if ((TIFR1 & bit (TOV1)) && timer1CounterValue < 0x7FFF)
    overflowCopy++;

  // wait until we noticed last one
  if (triggered)
    return;

  if (first)
    {
    startTime = (overflowCopy << 16) + timer1CounterValue;
    TIFR1 |= bit (ICF1);     // clear Timer/Counter1, Input Capture Flag
    TCCR1B =  bit (CS10);    // No prescaling, Input Capture Edge Select (falling on D8)
    first = false;
    return;
    }

  finishTime = (overflowCopy << 16) + timer1CounterValue;
  triggered = true;
  TIMSK1 = 0;    // no more interrupts for now
  }  // end of TIMER1_CAPT_vect

void prepareForInterrupts ()
  {
  noInterrupts ();  // protected code
  first = true;
  triggered = false;  // re-arm for next time
  // reset Timer 1
  TCCR1A = 0;
  TCCR1B = 0;

  TIFR1 = bit (ICF1) | bit (TOV1);  // clear flags so we don't get a bogus interrupt
  TCNT1 = 0;          // Counter to zero
  overflowCount = 0;  // Therefore no overflows yet

  // Timer 1 - counts clock pulses
  TIMSK1 = bit (TOIE1) | bit (ICIE1);   // interrupt on Timer 1 overflow and input capture
  // start Timer 1, no prescaler
  TCCR1B =  bit (CS10) | bit (ICES1);  // plus Input Capture Edge Select (rising on D8)
  interrupts ();
  }  // end of prepareForInterrupts


void setup ()
  {
  Serial.begin(115200);
  Serial.println("Duty cycle width calculator");
  // set up for interrupts
  prepareForInterrupts ();
  } // end of setup

void loop ()
  {
  // wait till we have a reading
  if (!triggered)
    return;

  // period is elapsed time
  unsigned long elapsedTime = finishTime - startTime;

  Serial.print ("Took: ");
  Serial.print (float (elapsedTime) * 62.5e-9 * 1e6);  // convert to microseconds
  Serial.println (" uS. ");
```

```
  // so we can read it
  delay (500);

  prepareForInterrupts ();
}   // end of loop
```

Testing shows that the displayed pulse width is within 100 ns of the actual width (two clock cycles basically) which isn't too bad.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,250 posts) bio *Forum Administrator*

Date Reply #14 on Tue 04 Nov 2014 03:12 AM (UTC)

Amended on Thu 14 Jan 2016 06:30 AM (UTC) by Nick Gammon

Message

Example strobe for car brake light

This example shows how we can use Timer 1 to strobe a brake light for a second, and then settle down into a constant "on" light, until the brake pedal is released.

```
const byte LED = 10;  // Timer 1 "B" output: OC1B
const byte PEDAL_PIN = 2;

const int ON_TIME     = 1000;    // This is the amount of time that the strobe will flash before going solid
const int STROBE_FREQ = 100;     // sets the delay between strobe pulses in milliseconds
const unsigned long countTo = (F_CPU / 1024) / (1000 / STROBE_FREQ);

volatile unsigned long whenPressed;
volatile bool pressed;

// ISR entered when brake pedal pressed
void brakePedal ()
  {
  bitSet (TCCR1A, COM1B1);   // clear OC1B on compare
  whenPressed = millis ();
  pressed = true;
  }  // end of brakePedal

void setup()
 {
  pinMode (LED, OUTPUT);
  pinMode (PEDAL_PIN, INPUT_PULLUP);

  // Fast PWM top at OCR1A
  TCCR1A = bit (WGM10) | bit (WGM11); // fast PWM
  TCCR1B = bit (WGM12) | bit (WGM13) | bit (CS12) | bit (CS10);   // fast PWM, prescaler of 1024
  OCR1A = countTo - 1;                  // zero relative
  OCR1B = (countTo / 4) - 1;            // 25% duty cycle

  attachInterrupt (0, brakePedal, FALLING);
  }  // end of setup

void loop()
  {
  // switch from strobing to steady after ON_TIME
  if (pressed && (millis () - whenPressed >= ON_TIME))
    {
    bitClear (TCCR1A, COM1B1);
    digitalWrite (LED, HIGH);  // turn light on fully
    }

  // if pedal up, make sure light is off
  if (digitalRead (PEDAL_PIN) == HIGH)
    {
    bitClear (TCCR1A, COM1B1);
    digitalWrite (LED, LOW);  // turn light off
    pressed = false;
    }

  // do other stuff here

  }  // end of loop
```

Example of flashing an LED with minimal code

This flashes D10 at a frequency of 2 Hz (off for one second, on for one second)

```
const int STROBE_FREQ = 2000;     // sets the period in milliseconds
const unsigned long countTo = ((float) F_CPU / 1024.0) / (1000.0 / STROBE_FREQ);

int main (void)
  {
  // D10 to output
  bitSet (DDRB, 2);

  // Fast PWM top at OCR1A
  TCCR1A = bit (WGM10) | bit (WGM11); // fast PWM
  TCCR1B = bit (WGM12) | bit (WGM13) | bit (CS12) | bit (CS10);   // fast PWM, prescaler of 1024
  OCR1A = countTo - 1;                 // zero relative
  OCR1B = (countTo / 2) - 1;           // 50% duty cycle
  bitSet (TCCR1A, COM1B1);   // clear OC1B on compare
  }  // end of main
```

Sketch size: 222 bytes on a Uno using IDE 1.0.6.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

The dates and times for posts above are shown in Universal Co-ordinated Time (UTC).

To show them in your local time you can join the forum, and then set the 'time correction' field in your profile to the number of hours difference between your location and UTC time.

188,524 views.

This is page 1, subject is 2 pages long: 1 2 NEXT >>

Postings by administrators only.

Refresh page

Go to topic: (Choose topic) Go Search the forum

top

Quick links: **MUSHclient**. MUSHclient **help**. Forum **shortcuts**. Posting **templates**. Lua **modules**. Lua **documentation**.

Home