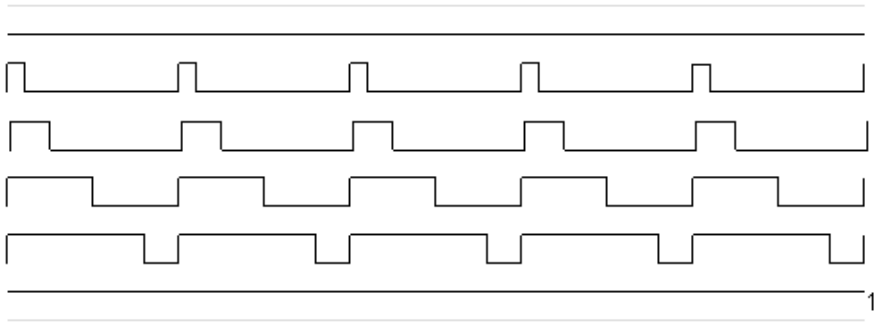
Ken Shirriff's blog

Chargers, microprocessors, Arduino, and whatever

Secrets of Arduino PWM

Pulse-width modulation (PWM) can be implemented on the Arduino in several ways. This article explains simple PWM techniques, as well as how to use the PWM registers directly for more control over the duty cycle and frequency. This article focuses on the Arduino Diecimila and Duemilanove models, which use the ATmega168 or ATmega328.

If you're unfamiliar with Pulse Width Modulation, see the tutorial. Briefly, a PWM signal is a digital square wave, where the frequency is constant, but that fraction of the time the signal is on (the duty cycle) can be varied between 0 and 100%.



PWM has several uses:

- Dimming an LED
- Providing an analog output; if the digital output is filtered, it will provide an analog voltage between 0% and 100% .
- Generating audio signals.
- Providing variable speed control for motors.
- Generating a modulated signal, for example to drive an infrared LED for a remote control.

Simple Pulse Width Modulation with `analogWrite`

The Arduino's programming language makes PWM easy to use; simply call `analogWrite(pin, dutyCycle)`, where `dutyCycle` is a value from 0 to 255, and `pin` is one of the PWM pins (3, 5, 6, 9, 10, or 11). The `analogWrite` function provides a simple interface to the hardware PWM, but doesn't provide any control over frequency. (Note that despite the function name, the output is a digital signal.)

Popular Posts

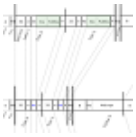

iPad charger teardown: inside Apple's charger and a risky phony


A dozen USB chargers in the lab: Apple is very good, but not quite the best

A Multi-Protocol Infrared Remote Library for the Arduino

Apple iPod charger teardown: quality in a tiny expensive package


Tiny, cheap, and dangerous: Inside a (fake) iPhone charger


Bitcoins the hard way: Using the raw Bitcoin protocol

Reverse-engineering the TL431: the most common chip you've never heard of


Teardown and exploration of Apple's Magsafe connector

Labels

Probably 99% of the readers can stop here, and just use `analogWrite`, but there are other options that provide more flexibility.

Bit-banging Pulse Width Modulation

You can "manually" implement PWM on any pin by repeatedly turning the pin on and off for the desired times. e.g.

```
void setup()
{
  pinMode(13, OUTPUT);
}

void loop()
{
  digitalWrite(13, HIGH);
  delayMicroseconds(100); //
Approximately 10% duty cycle @ 1KHz
  digitalWrite(13, LOW);
  delayMicroseconds(900);
}
```

This technique has the advantage that it can use any digital output pin. In addition, you have full control the duty cycle and frequency. One major disadvantage is that any interrupts will affect the timing, which can cause considerable jitter unless you disable interrupts. A second disadvantage is you can't leave the output running while the processor does something else. Finally, it's difficult to determine the appropriate constants for a particular duty cycle and frequency unless you either carefully count cycles, or tweak the values while watching an oscilloscope.

Using the ATmega PWM registers directly

The ATmega168P/328P chip has three PWM timers, controlling 6 PWM outputs. By manipulating the chip's timer registers directly, you can obtain more control than the `analogWrite` function provides.

The AVR ATmega328P datasheet provides a detailed description of the PWM timers, but the datasheet can be difficult to understand, due to the many different control and output modes of the timers. The following attempts to clarify the use of the timers.

The ATmega328P has three timers known as Timer 0, Timer 1, and Timer 2. Each timer has two output compare registers that control the PWM width for the timer's two outputs: when the timer reaches the compare register value, the corresponding output is toggled. The two outputs for each timer will normally have the same frequency, but can have different duty cycles (depending on

6502 8085 apple arc arduino bitcoin c# calculator css electronics f# fractals genome haskell html5 ipv6 ir java javascript math oscilloscope photo power supply random reverse-engineering sheevaplug snark spanish teardown theory Z-80

Power supply posts

- iPhone charger teardown
- A dozen USB chargers
- Magsafe hacking
- Inside a fake iPhone charger
- Power supply history

the respective output compare register).

Each of the timers has a prescaler that generates the timer clock by dividing the system clock by a prescale factor such as 1, 8, 64, 256, or 1024. The Arduino has a system clock of 16MHz and the timer clock frequency will be the system clock frequency divided by the prescale factor. Note that Timer 2 has a different set of prescale values from the other timers.

The timers are complicated by several different modes. The main PWM modes are "Fast PWM" and "Phase-correct PWM", which will be described below. The timer can either run from 0 to 255, or from 0 to a fixed value. (The 16-bit Timer 1 has additional modes to supports timer values up to 16 bits.) Each output can also be inverted.

The timers can also generate interrupts on overflow and/or match against either output compare register, but that's beyond the scope of this article.

Timer Registers

Several registers are used to control each timer. The Timer/Counter Control Registers TCCRnA and TCCRnB hold the main control bits for the timer. (Note that TCCRnA and TCCRnB do not correspond to the outputs A and B.) These registers hold several groups of bits:

- Waveform Generation Mode bits (WGM): these control the overall mode of the timer. (These bits are split between TCCRnA and TCCRnB.)
- Clock Select bits (CS): these control the clock prescaler
- Compare Match Output A Mode bits (COMnA): these enable/disable/invert output A
- Compare Match Output B Mode bits (COMnB): these enable/disable/invert output B

The Output Compare Registers OCRnA and OCRnB set the levels at which outputs A and B will be affected. When the timer value matches the register value, the corresponding output will be modified as specified by the mode.

The bits are slightly different for each timer, so consult the datasheet for details. Timer 1 is a 16-bit timer and has additional modes. Timer 2 has different prescaler values.

Fast PWM

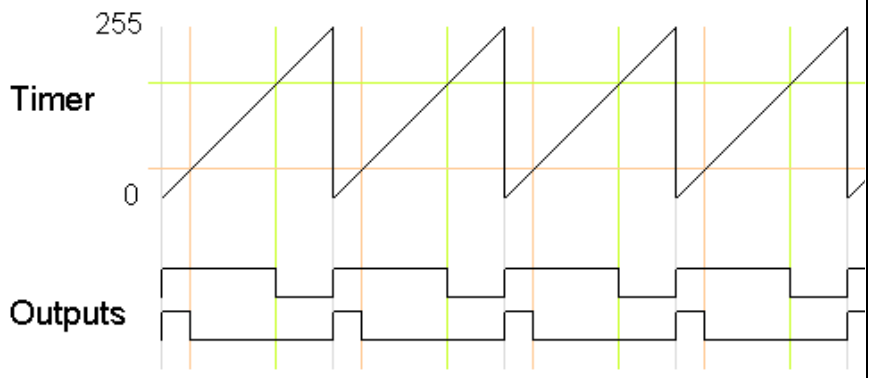
In the simplest PWM mode, the timer repeatedly counts from 0 to 255. The output turns on when the timer is at 0, and turns off when the timer matches the output compare register. The higher the value in the output compare register, the higher the duty cycle. This mode is known as **Fast PWM Mode**.
The following diagram shows the outputs for two particular values

Email:
kens@arcfn.com

Blog Archive

► 2014 (8)
► 2013 (24)
► 2012 (10)
► 2011 (11)
► 2010 (22)
▼ 2009 (22)
► December (2)
► November (5)
► September (1)
► August (3)
▼ July (1)
Secrets of Arduino PWM
► June (3)
► April (1)
► March (3)
► February (2)
► January (1)
► 2008 (27)

of OCRnA and OCRnB. Note that both outputs have the same frequncy, matching the frequency of a complete timer cycle.



The following code fragment sets up fast PWM on pins 3 and 11 (Timer 2). To summarize the register settings, setting the waveform generation mode bits WGM to 011 selects fast PWM. Setting the COM2A bits and COM2B bits to 10 provides non-inverted PWM for outputs A and B. Setting the CS bits to 100 sets the prescaler to divide the clock by 64. (Since the bits are different for the different timers, consult the datasheet for the right values.) The output compare registers are arbitrarily set to 180 and 50 to control the PWM duty cycle of outputs A and B. (Of course, you can modify the registers directly instead of using `pinMode`, but you do need to set the pins to output.)

```
  pinMode(3, OUTPUT);
  pinMode(11, OUTPUT);
  TCCR2A = _BV(COM2A1) | _BV(COM2B1) |
_BV(WGM21) | _BV(WGM20);
  TCCR2B = _BV(CS22);
  OCR2A = 180;
  OCR2B = 50;
```

On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / 256 = 976.5625Hz
- Output A duty cycle: (180+1) / 256 = 70.7%
- Output B frequency: 16 MHz / 64 / 256 = 976.5625Hz
- Output B duty cycle: (50+1) / 256 = 19.9%

The output frequency is the 16MHz system clock frequency, divided by the prescaler value (64), divided by the 256 cycles it takes for the timer to wrap around. Note that fast PWM holds the output high one cycle longer than the compare register value.

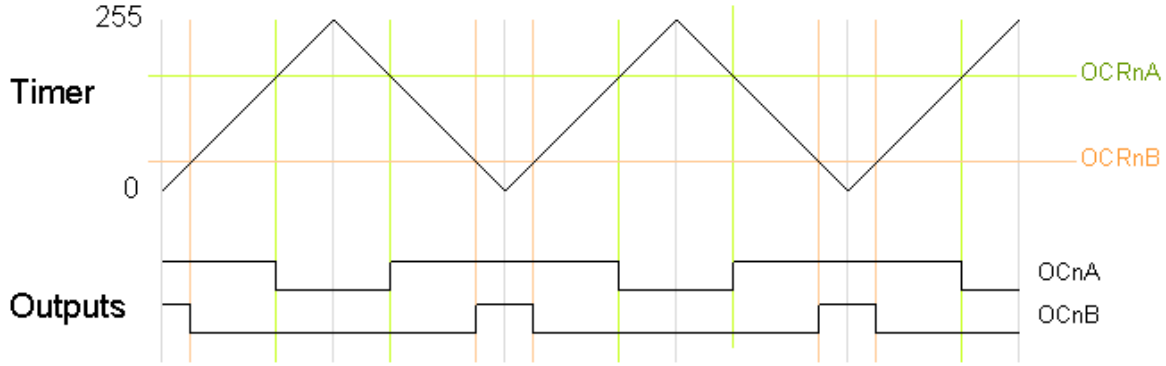
Phase-Correct PWM

The second PWM mode is called **phase-correct PWM**. In this mode, the timer counts from 0 to 255 and then back down to 0. The output turns off as the timer hits the output compare register value on the way up, and turns back on as the timer hits the output

Quick links

- Arduino IR library
- 6502 reverse-engineering

compare register value on the way down. The result is a more symmetrical output. The output frequency will be approximately half of the value for fast PWM mode, because the timer runs both up and down.



The following code fragment sets up phase-correct PWM on pins 3 and 11 (Timer 2). The waveform generation mode bits WGM are set to to 001 for phase-correct PWM. The other bits are the same as for fast PWM.

```
  pinMode(3, OUTPUT);
  pinMode(11, OUTPUT);
  TCCR2A = _BV(COM2A1) | _BV(COM2B1) |
_BV(WGM20);
  TCCR2B = _BV(CS22);
  OCR2A = 180;
  OCR2B = 50;
```

On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / 255 / 2 = 490.196Hz
- Output A duty cycle: 180 / 255 = 70.6%
- Output B frequency: 16 MHz / 64 / 255 / 2 = 490.196Hz
- Output B duty cycle: 50 / 255 = 19.6%

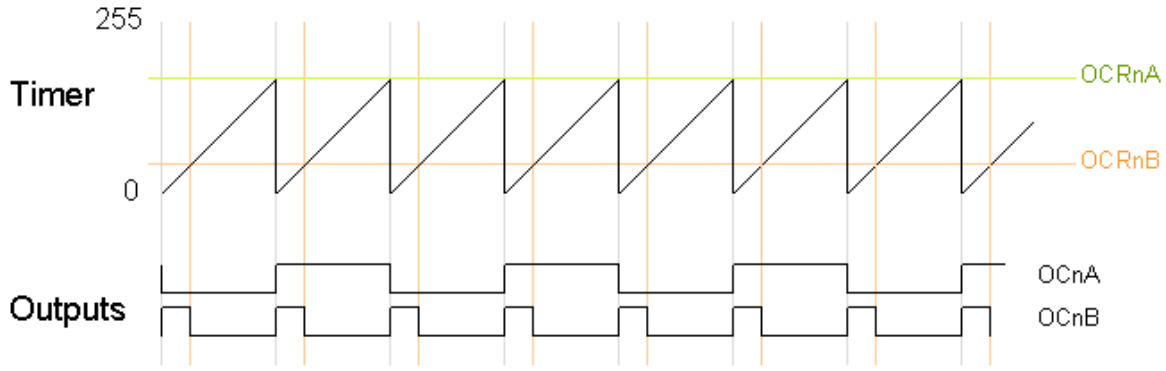
Phase-correct PWM divides the frequency by two compared to fast PWM, because the timer goes both up and down. Somewhat surprisingly, the frequency is divided by 255 instead of 256, and the duty cycle calculations do not add one as for fast PWM. See the explanation below under "Off-by-one".

Varying the timer top limit: fast PWM

Both fast PWM and phase correct PWM have an additional mode that gives control over the output frequency. In this mode, the timer counts from 0 to OCRA (the value of output compare register A), rather than from 0 to 255. This gives much more control over the output frequency than the previous modes. (For even more frequency control, use the 16-bit Timer 1.)

Note that in this mode, only output B can be used for PWM; OCRA cannot be used both as the top value and the PWM compare value. However, there is a special-case mode "Toggle OCnA on Compare Match" that will toggle output A at the end of each cycle, generating a fixed 50% duty cycle and half frequency in this case. The examples will use this mode.

In the following diagram, the timer resets when it matches OCRnA, yielding a faster output frequency for OCnB than in the previous diagrams. Note how OCnA toggles once for each timer reset.



The following code fragment sets up fast PWM on pins 3 and 11 (Timer 2), using OCR2A as the top value for the timer. The waveform generation mode bits WGM are set to to 111 for fast PWM with OCRA controlling the top limit. The OCR2A top limit is arbitrarily set to 180, and the OCR2B compare register is arbitrarily set to 50. OCR2A's mode is set to "Toggle on Compare Match" by setting the COM2A bits to 01.

```
  pinMode(3, OUTPUT);
  pinMode(11, OUTPUT);
  TCCR2A = _BV(COM2A0) | _BV(COM2B1) |
_BV(WGM21) | _BV(WGM20);
  TCCR2B = _BV(WGM22) | _BV(CS22);
  OCR2A = 180;
  OCR2B = 50;
```

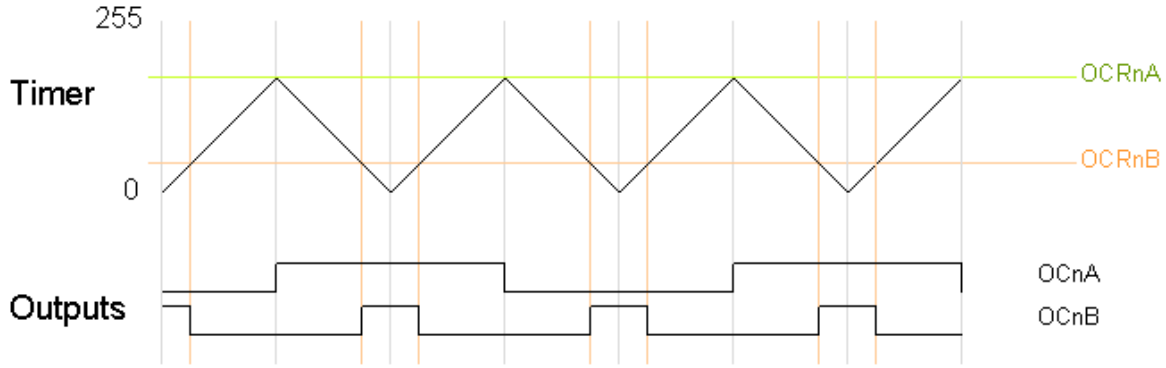
On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / (180+1) / 2 = 690.6Hz
- Output A duty cycle: 50%
- Output B frequency: 16 MHz / 64 / (180+1) = 1381.2Hz
- Output B duty cycle: (50+1) / (180+1) = 28.2%

Note that in this example, the timer goes from 0 to 180, which takes 181 clock cycles, so the output frequency is divided by 181. Output A has half the frequency of Output B because the Toggle on Compare Match mode toggles Output A once each complete timer cycle.

Varying the timer top limit: phase-correct PWM

Similarly, the timer can be configured in phase-correct PWM mode to reset when it reaches OCRnA.



The following code fragment sets up phase-correct PWM on pins 3 and 11 (Timer 2), using OCR2A as the top value for the timer. The waveform generation mode bits WGM are set to to 101 for phase-correct PWM with OCRA controlling the top limit. The OCR2A top limit is arbitrarily set to 180, and the OCR2B compare register is arbitrarily set to 50. OCR2A's mode is set to "Toggle on Compare Match" by setting the COM2A bits to 01.

```
  pinMode(3, OUTPUT);
  pinMode(11, OUTPUT);
  TCCR2A = _BV(COM2A0) | _BV(COM2B1) |
_BV(WGM20);
  TCCR2B = _BV(WGM22) | _BV(CS22);
  OCR2A = 180;
  OCR2B = 50;
```

On the Arduino Duemilanove, these values yield:

- Output A frequency: 16 MHz / 64 / 180 / 2 / 2 = 347.2Hz
- Output A duty cycle: 50%
- Output B frequency: 16 MHz / 64 / 180 / 2 = 694.4Hz
- Output B duty cycle: 50 / 180 = 27.8%

Note that in this example, the timer goes from 0 to 180 and back to 0, which takes 360 clock cycles. Thus, everything is divided by 180 or 360, unlike the fast PWM case, which divided everything by 181; see below for details.

Off-by-one

You may have noticed that fast PWM and phase-correct PWM seem to be off-by-one with respect to each other, dividing by 256 versus 255 and adding one in various places. The documentation is a bit opaque here, so I'll explain in a bit of detail.

Suppose the timer is set to fast PWM mode and is set to count up to an OCRnA value of 3. The timer will take on the values

012301230123... Note that there are 4 clock cycles in each timer cycle. Thus, the frequency will be divided by 4, not 3. The duty cycle will be a multiple of 25%, since the output can be high for 0, 1, 2, 3, or 4 cycles out of the four. Likewise, if the timer counts up to 255, there will be 256 clock cycles in each timer cycle, and the duty cycle will be a multiple of 1/256. To summarize, fast PWM divides by N+1 where N is the maximum timer value (either OCRnA or 255).

Now consider phase-correct PWM mode with the timer counting up to an OCRnA value of 3. The timer values will be 012321012321... There are 6 clock cycles in each timer cycle (012321). Thus the frequency will be divided by 6. The duty cycle will be a multiple of 33%, since the output can be high for 0, 2, 4, or 6 of the 6 cycles. Likewise, if the timer counts up to 255 and back down, there will be 510 clock cycles in each timer cycle, and the duty cycle will be a multiple of 1/255. To summarize, phase-correct PWM divides by 2N, where N is the maximum timer value.

The second important timing difference is that fast PWM holds the output high for one cycle longer than the output compare register value. The motivation for this is that for fast PWM counting to 255, the duty cycle can be from 0 to 256 cycles, but the output compare register can only hold a value from 0 to 255. What happens to the missing value? The fast PWM mode keeps the output high for N+1 cycles when the output compare register is set to N so an output compare register value of 255 is 100% duty cycle, but an output compare register value of 0 is **not** 0% duty cycle but 1/256 duty cycle. This is unlike phase-correct PWM, where a register value of 255 is 100% duty cycle and a value of 0 is a 0% duty cycle.

Timers and the Arduino

The Arduino supports PWM on a subset of its output pins. It may not be immediately obvious which timer controls which output, but the following table will clarify the situation. It gives for each timer output the output pin on the Arduino (i.e. the silkscreened label on the board), the pin on the ATmega chip, and the name and bit of the output port. For instance Timer 0 output OC0A is connected to the Arduino output pin 6; it uses chip pin 12 which is also known as PD6.

Timer output	Arduino output	Chip pin	Pin name
OC0A	6	12	PD6
OC0B	5	11	PD5
OC1A	9	15	PB1
OC1B	10	16	PB2
OC2A	11	17	PB3
OC2B	3	5	PD3

The Arduino performs some initialization of the timers. The Arduino initializes the prescaler on all three timers to divide the

clock by 64. Timer 0 is initialized to Fast PWM, while Timer 1 and Timer 2 is initialized to Phase Correct PWM. See the Arduino source file wiring.c for details.

The Arduino uses Timer 0 internally for the `millis()` and `delay()` functions, so be warned that changing the frequency of this timer will cause those functions to be erroneous. Using the PWM outputs is safe if you don't change the frequency, though.

The `analogWrite(pin, duty_cycle)` function sets the appropriate pin to PWM and sets the appropriate output compare register to `duty_cycle` (with the special case for duty cycle of 0 on Timer 0). The `digitalWrite()` function turns off PWM output if called on a timer pin. The relevant code is wiring_analog.c and wiring_digital.c.

If you use `analogWrite(5, 0)` you get a duty cycle of 0%, even though pin 5's timer (Timer 0) is using fast PWM. How can this be, when a fast PWM value of 0 yields a duty cycle of 1/256 as explained above? The answer is that `analogWrite` "cheats"; it has special-case code to explicitly turn off the pin when called on Timer 0 with a duty cycle of 0. As a consequency, the duty cycle of 1/256 is unavailable when you use `analogWrite` on Timer0, and there is a jump in the actual duty cycle between values of 0 and 1.

Some other Arduino models use dfferent AVR processors with similar timers. The Arduino Mega uses the ATmega1280 (datasheet), which has four 16-bit timers with 3 outputs each and two 8-bit timers with 2 outputs each. Only 14 of the PWM outputs are supported by the Arduino Wiring library, however. Some older Arduino models use the ATmega8 (datasheet), which has three timers but only 3 PWM outputs: Timer 0 has no PWM, Timer 1 is 16 bits and has two PWM outputs, and Timer 2 is 8 bits and has one PWM output.

Troubleshooting

It can be tricky to get the PWM outputs to work. Some tips:

- You need to both enable the pin for output and enable the PWM mode on the pin in order to get any output. I.e. you need to do `pinMode()` and set the COM bits.
- The different timers use the control bits and prescaler differently; check the documentation for the appropriate timer.
- Some combinations of bits that you might expect to work are reserved, which means if you try to use them, they won't work. For example, toggle mode doesn't work with fast PWM to 255, or with output B.
- Make sure the bits are set the way you think. Bit

operations can be tricky, so print out the register values and make sure they are what you expect.

- Make sure you're using the right output pins. See the table above.
- You'll probably want a decoupling capacitor to avoid spikes on the output.

An oscilloscope is very handy for debugging PWM if you have access to one. If you don't have one, I recommend using your sound card and a program such as xoscope.

Conclusion

I hope this article helps explain the PWM modes of the Arduino. I found the documentation of the different modes somewhat opaque, and the off-by-one issues unexplained. Please let me know if you encounter any errors.

Labels: arduino

31 comments:

Anonymous said...

Thank you for the article!

As a beginning Arduino user, this is very helpful!

November 1, 2009 at 10:46 AM

Didier said...

Excellent paper. Many Thanks

January 10, 2010 at 5:57 AM

Jason said...

exactly what I needed! thanks!

January 23, 2010 at 1:38 PM

Tom said...

I am so glad I found this. Just what I needed.
One question though-likely I am not understanding some subtlety.
It seems from the datasheet that the 64 pre-scale corresponds to 110. I am wondering if I am missing something, as you mention 100 as the 64 prescale divider.
Thank you!

February 7, 2010 at 9:56 AM

Anonymous said...

Excellent article, I've been going through some example code but this made it all come together.

Thanks,
Mike

February 27, 2010 at 9:42 AM

Ken Shirriff said...

Tom, you asked about how to divide by 64 with the prescaler. For timer0 and timer1, the clock select bits are set to 011 (CS02,CS01,CS00 or CS12,CS11,CS10). But for timer2, the clock select bits are set to 100 (CS20,CS21,CS20). Confusingly, Timer 2 uses different clock select bit values from Timers 0 and 1.

March 16, 2010 at 11:22 PM

monty said...

Ken,

I am just learning about Arduino and I have a question about your article on “Secrets of Arduino PWM”.

Your example says
pinMode(3, OUTPUT);
pinMode(11, OUTPUT);
TCCR2A = _BV(COM2A1) | _BV(COM2B1) |
_BV(WGM20);
TCCR2B = _BV(CS22);
OCR2A = 180;
OCR2B = 50;
What I am confused about is previously you stated that that these examples set the clock divisor to 64. Is there a bit inversion in the writes to an AVR register?
Your line TCCR2B = _BV(CS22); would set the TCCR2B register to 00000100b.

Is there something I’m missing here?

Thanks in advance for your response.

Monty

April 21, 2010 at 9:15 AM

Steve Gough said...

Very helpful, thanks--we're trying to use Arduinos to power a small pump and measure flow for an open source river modeling system, and I was hung on this topic until reading this--see our stuff at lrrd.blogspot.com.

May 22, 2010 at 6:06 PM

Matej said...

that is wonderful.. but whan i was looking at the library i saw in the comments that this will work only for 36-40khz modulation? Whay is that and how can i change it so it will work on 56khz?

May 23, 2010 at 1:32 AM

Cassiano Rabelo said...

Excellent tutorial Ken. Thanks a lot for sharing your knowledge.

Would you mind elaborating a bit more on how someone could use the sound card and a program such as xoscope to help debug this sort of thing? Do you mean by plugin a speaker to the arduino pin, getting it close to the computer mic and using such a software to graph the wave?

Thanks once again.

September 23, 2010 at 5:55 PM

Colin Adamson said...

Most excellent thank you -- saved me days trying to make sense of that Atmega328 datasheet
I was using the default 490 Mhz arduino analogWrite PWM and was getting horribly low torque with some small DC motors on low duty cycles. Changed it to 30Hz and now its way better, still runs smooth.

December 16, 2010 at 11:44 PM

Ken Shirriff said...

Colin, thanks for your comment. I'm glad the article was helpful.

Cassiano, with a sound card oscilloscope, you connect the signal directly to the sound card through a resistor to reduce the current. There are details at http://www.ehow.com/how_2278973_use-sound-card-oscilloscope.html

December 17, 2010 at 10:27 PM

Thorsten said...

Thanks for explaining this matter so extensively! I was looking for a way to generate 1 MHz on one of the Arduino-pins. Your post helped me a great deal to accomplish that.

The reason I am writing this comment is the following: It took me almost 6 hours till I found out (mainly in sheer desperation) that the order of setting the timer control registers TCCR2* and the output compare registers OCR2* seems to matter! If you assign an OCR before setting the corresponding TCCR the timer simply doesn't start counting.

February 13, 2011 at 11:47 AM

Anonymous said...

but how can i get it to make a 38khz freq with 50% duty cicle?

March 8, 2011 at 2:14 PM

Quarkninja said...

Very informative! Excellent Work! Thanks!

May 1, 2011 at 12:37 AM

coopermaa said...

Very helpful, thanks for your share.

As for debugging PWM, I think proteus ISIS is a good tool for that purpose. Proteus has a virtual osilloscope, see my post(in chinese, sorry): http://coopermaa2nd.blogspot.com/2011/05/proteus-pwm.html

May 15, 2011 at 2:56 AM

Sami Mughal said...

I wrote a brief article on creating PWM using the Arduino UNO, with ability to control the frequency on my blog. Just leaving a link here as I found this article very useful in my research !

http://smacula.blogspot.com/2011/04/creating-variable-frequency-pwm-output.html

June 27, 2011 at 8:21 AM

Anonymous said...

Great summary and guide for using the ATmega328p timers! Thanks for taking the time to put this together. Very helpful.

July 24, 2011 at 2:05 PM

Michael said...

This may be a stupid question by a newbie, but what is the _BV function? Nowhere to be found in the Arduino reference.

August 14, 2011 at 11:58 PM

coopermaa said...

_BV is a macro defined in avr-libc:

#define _BV(x) (1 << x)

see
http://194.81.104.27/~brian/microprocessor/BVMacro.pdf

August 15, 2011 at 12:47 AM

Anonymous said...

Thank you very much, this is the best condensed reference I have seen, all clear and to the point!!!

January 19, 2012 at 1:21 AM

mitch deoudes said...

The version of this article posted at arduino.cc is missing all of the diagrams except the first one.

February 3, 2012 at 9:32 PM

Henry Best said...

I want to use a low frequency PWM, below 10Hz. The frequency isn't critical but would have to be in that area. Any ideas how to get down to that frequency? I want to use the Arduino Uno for other things whilst the PWM is being output.

February 6, 2012 at 10:02 PM

Anonymous said...

Hi, is it possible for my Arduino Duemilanove to go down to 70 Hz? I need to output a PWM from my board at that frequency. Basically I am creating a buffer that takes in a PWM signal and outputs a PWM that has the same pulse width and frequency as the input signal. Is it also possible to update the frequency in every execution? I can't seem to get it to work
properly.. Thanks in advance!

March 29, 2012 at 8:50 PM

Sami Mughal said...

Hi,

If you read my article:
http://www.smacula.co.uk/2011/04/creating-variable-frequency-pwm-output.html

You can actually go as low as 15Hz. I have not tried to go that low myself though, but don't see why it would not work.

March 29, 2012 at 11:50 PM

GratefulFrog said...

Hi,
Your article has been a great help, but now I am working on an Arduino Micro with an ATMEGA 32u4 processor.

All I want to do is get phase-correct PWM at the highest possible frequency on 3 pins.

Will these 2 lines do that?

TCCR1B = _BV(CS00); // change the PWM frequencey to 31.25kHz - pins 9 & 10

// timer 0B : pin 3 & 11
TCCR0B = _BV(CS00); // change the PWM frequencey to 31.25 kHz - pin 3 & 11

please let me know, if you can, by relying to my id at gmail.
Cheers,
Bob

March 8, 2013 at 9:36 AM

Сергей Орляк said...

Hi Ken,
Thank you for your library!
I modified it a bit to work with Hitachi air conditioning. But there is one momnet I can not understand. I had to increase the buffer to 600 (RAWBUF 600). But then I try to read I can read only 532 byte. While there is still 8 :(In what may be another reason of not getting all the data from the console?
Thank you! And excuse me for my english!

April 14, 2013 at 9:33 AM

Ken Shirriff said...

Сергей: another user of the irremote library found that rawlen type (uint8_t) is too small for more than 256 entries, so try replacing it by unsigned short int. Also, discussion of the irremote library is here.

April 15, 2013 at 8:42 AM

Gabriel said...

Fantastic Article!!! Thanks a ton.

As Mitch Deoudes, said, "The version of this article posted at arduino.cc is missing all of the diagrams except the first one." This is still true. If you could figure out how to get the diagrams added back into the article on arduino.cc (http://arduino.cc/en/Tutorial/SecretsOfArduinoPWM) that would be great!

Thanks!
Gabriel Staples
http://electricrcaircraftguy.blogspot.com/

December 21, 2013 at 9:29 PM

dafaddah said...

I'm working on 'sculpting' an output signal to meet certain shape and frequency parameters. This is the first time I'm beginning to see the light at the end of the tunnel trying to understand how to code PWM to accomplish this. Many thanks!!

February 11, 2014 at 4:22 AM

Gabriel said...

dafaddah, that sounds very interesting; I'd like to see what you can do to "sculpt" a signal shape, so would you mind sharing a link here when you are done, so others can see your work?

February 11, 2014 at 4:56 AM

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)