Skip to content

uC Hobby
Microcontrollers Electronics Hobby

- About
- Discussions
- Giveaway Program
- Links
- Make Controller Kit
  - Parts
- Scrounging
  - FAQ
  - How-To: Use a Heat Gun
  - How-To: Work Surface
- Web Rings
- Log In

Search

Categories:

- Arduino
- Contest
- Development Tools
- Discovering
- Electronics Links
- Givaways
- Hacks
- Ideas
- Microcontroller
- Parts
- Projects
- Review
- Scrounging
- Scrounging Parts
- TGIMBOEJ
- uC Hobby Site
- Uncategorized
- Workshop Tips
- Workshop Tools

Arduino Interrupts

Often when working on microcontroller projects you need a background function to run at regular intervals. This is often done by setting up a hardware timer to generate an interrupt. The interrupt triggers an Interrupt Service Routine (ISR) to handle the periodic interrupt. In this article I describe setting up the 8-Bit Timer2 to generate interrupts on an Arduino ATMega168. I walk through steps required for setup and inside the ISR function.

If you are following the Arduino sound articles this one will be important to read as well. Future articles will make use of this timer feature to control frequency generation. In fact, I started to do this article as another installment on the sound project but decided it would be best to cover timer interrupts separately.

The Arduino default processor is an ATMega168 (datasheet link). This microcontroller has several I/O systems that every Arduino user is familiar with because the Arduino library makes using them easy. The Digital I/O, PWM, A/D Inputs, and Serial port for example. The ATMega168 also has three internal hardware timers. While the Arduino library does make use of some of the timer features it does not directly cover using a timer to generate periodic interrupts.

Interrupts?
As the name suggest, interrupts are signals that interrupt the normal flow of a program. Interrupts are usually used for hardware devices that require immediate attention when events occur. For example the serial port system or UART on the microcontroller must be serviced when a new character arrives. If it's not handled quickly the new character can be lost.

Interrupts Links

- Wikipedia has a good overview about interrupts.
- Interrupts and Exceptions
- Embedded.com – Introduction to Interrupts

When a new character arrives the UART system generates an interrupt. The microcontroller stops running the main code (your application) and jumps to an Interrupt Service Routine (ISR) for the specific interrupt. In this case, a received character interrupt. This ISR grabs the new character from the UART, places it into a buffer, then clears the interrupt and returns. When the ISR returns the microcontroller goes back to your main code and continues where it left off. This all happens in the background and your main application code is not directly effected.

If you have lots of interrupts firing or fast timer interrupts your main code will execute slower because the microcontroller is spreading it's processing time between your main code and all the ISR functions.

You may be thinking, why not just check for a new character every now and then instead of using this complicated interrupt process. Lets work an example to see how important these interrupt processes are. Say you have a serial port with a data rate of 9600 baud. This means each bit of a character is sent at a frequency of 9600Hz or about 10KHz. Each bit take about 100uS. About 10 bits are required to send one character so we receive one complete character every 1mS or so. If our UART is buffered, we have to pull out the last character before the next one finishes, this gives us 1mS to get the job done. If our UART is not buffered, we have to get the character out in about 1 bit time or 1uS. Lets use the buffered example for now.

We have to check for a received byte faster then once every millisecond to keep from loosing data. In the Arduino environment that means our Loop function has to get around to reading the UART status and possibly the data byte 1000 times a second. This is easily doable but it would greatly complicate the code you have to write. As long as your loop function never takes more then 1mS to complete you could get away with this. But consider that you might have more then one I/O device to service or that you need to operate at much higher baud rates. Surly you can see how ugly this can get very quick.

With an interrupt you don't have to keep checking to see if a character has arrived. The hardware will signal, with an interrupt, and the processor will quickly call the ISR to grab the character in time. Instead of spending a huge amount of your microprocessor time checking the status of the UART, you never have to check the status, you just setup the hardware interrupt and do the necessary work in the ISR. Your main code is not directly affected and requires no special considerations for the hardware device.

Timer Interrupt

For this article I will focus on using the hardware timer 2 for a periodic interrupt. The original idea was to use this timer to generate the note frequencies for the Arduino sound projects. To bit-bang out a tone or frequency we need to toggle an I/O pin at a consistent frequency. We did this in parts 2 and 3 of the sound articles using delay loops. This was easy but means our processor is tied up doing nothing but waiting for the correct time to toggle the pin. Using the timer interrupt we can do other things and let the ISR toggle the pin when the timer signals that it is time.

We only need to setup the timer to signal with an interrupt at the correct times. Instead of spinning in a useless loop waiting for the delay to time out, our main code can be doing other things like monitoring a motion sensor or doing motor control. What ever our project needs, we no longer need processor time for polling delays.

I will cover the ISR in general, just enough to deal with timer 2 interrupts. Study this link from the avr-libc user-manual for more information about using interrupts on AVR processors. You can also look at the timer sections in the ATMega168 data sheet. You don't need a complete understanding at this point but ultimately you may want to get up to speed on using interrupts as they are an important tool for microcontroller applications.

Timers on the Arduino

I contacted David Mellis of the Arduino development team and learned that the library makes use of all three timers on the ATMega168.

- **Timer0** (System timing, PWM 5 and 6)
  Used to keep track of the time the program has been running. The millis() function to return the number of milliseconds since the program started using a global incremented in the timer 0 ISR. Timer 0 is also used for PWM outputs on digital pins 5 and 6.
- **Timer1** (PWM 9 and 10)
  Used to drive PWM outputs for digital pins 9 and 10.
- **Timer2** (PWM 3 and 11)
  Used to drive PWM outputs for digital pins 3 and 11.

While all the timers are used only Timer0 has an assigned timer ISR. This means we can hijack Timer1 and/or Timer2 for our uses. The PWM function on some of the I/O pins will be affected as a result however. If you plan to use PWM you need to know what is affected. I chose to use timer 2 so PWM pins 3 and 11 will be affected.

My test code completely disabled the PWM outputs for digital pins controlled from timer 2. I suspect that the library PWM functions are expecting the counter to work over a specific range which I have overridden. The capture compare values loaded just don't make sense with my timer values for PWM.

David Mellis gave me this link to explore how the library codes uses the timers. It is well documented and I am sure it will serve me well in future efforts. In a future experiment I plan to control the PWM generation directly so that it can be done at a much higher frequency then normal. High frequency PWM may be great for an Audio DAC function.

Arduino related interrupt links

Here are some links related to interrupts on the Arduino. I reviewed several of these while playing with interrupts. I have to give credit for my knowledge to these authors.

- Handling external Interrupts with Arduino

- I will think before I code
- Interrupts vs polling
- More external interrupts?
- AVR Libc : Interrupts

The Code
The code covered in this article is available in a zip file here.

Setup Timer2
The code below shows a function I created to setup timer2. This function enables the timer2 overflow interrupt, sets the prescalar for the timer and calculates the timer load value given the desired timeout frequency. This code is based on code I found at the links given above. I do have experience reading AVR datasheets but it is much easier to reuse code if you can find it. I recommend that you take a look at the timer information in the data sheets if for no other reason then to gain an appreciation for how difficult it can be to figure this out.

```
#define TIMER_CLOCK_FREQ 2000000.0 //2MHz for /8 prescale from 16MHz

//Setup Timer2.
//Configures the ATMega168 8-Bit Timer2 to generate an interrupt
//at the specified frequency.
//Returns the timer load value which must be loaded into TCNT2
//inside your ISR routine.
//See the example usage below.
unsigned char SetupTimer2(float timeoutFrequency){
  unsigned char result; //The timer load value.

  //Calculate the timer load value
  result=(int)((257.0-(TIMER_CLOCK_FREQ/timeoutFrequency))+0.5);
  //The 257 really should be 256 but I get better results with 257.

  //Timer2 Settings: Timer Prescaler /8, mode 0
  //Timer clock = 16MHz/8 = 2Mhz or 0.5us
  //The /8 prescale gives us a good range to work with
  //so we just hard code this for now.
  TCCR2A = 0;
  TCCR2B = 0<<CS22 | 1<<CS21 | 0<<CS20;

  //Timer2 Overflow Interrupt Enable
  TIMSK2 = 1<<TOIE2;

  //load the timer for its first cycle
  TCNT2=result;

  return(result);
}
```

SetupTimer2 walk through
First is a define for the timer clock frequency. It shows that the clock frequency is set to 2MHz because we use a divide by 8 prescale from the 16MHz master clock. This is hard coded in the function. The define just makes the code look better and may be useful in some application. At the very least, it reminds me how I setup the timer.

The function takes one argument, the desired timeout frequency, and returns the value needed to re-load the timer in the ISR. The function does not limit the requested frequency but you should not try going too high. I discuss this issue later in the article.

Next the timer reload value is calculated. This is a very easy calculation but needs to be done with floating point math. Luckily we only need to do this once as floating math is expensive in terms of processor time. We assume that the timer will be set to run at 2MHz for each count. The reload value is the number of counts we want at 2MHz between interrupts. You may notice that I used 257 instead of 256 for the number of counts in the equation above. I know that the correct value should be 256 but I get better results with 257. I will explain why later in this article.

The next chunk of cryptic code sets the timer into mode 0 and selects the /8 prescaler. Mode 0 is a basic timer mode and the /8 prescale is how we get the counter to count at 2MHz or 0.5uS per count.

Next the overflow interrupt is enabled. After this code executes the microcontroller will call the ISR every time the counter rolls over from 0xFF to 0×00. This will happen after the counter has stepped from our load value over FF and back to 00.

Lastly we load the count value into the timer and return this load value so that the ISR can use it later.

I have run the timer as high 50KHz. This is very fast and any work done in the ISR will significantly hurt the performance of your main application code. I recommend that you don't try frequencies above 50KHz unless you are doing almost nothing in the ISR.

Microcontroller Interrupt load
To give you an idea of the effect, consider that at 50Khz the timer ISR should be triggered every 20uS. The processor running at 16MHz can execute about one machine instruction every 63nS or about 320 machine instructions for every interrupt cycle (20uS). Also consider that each line of C code can take many machine instructions to execute. Every instruction used in the ISR subtracts from the time available for any other code to execute. If our ISR used about 150 machine cycles then we would eat up 1/2 of our available processor time. The main code would slow to about 1/2 the time it would have otherwise taken while the interrupt is active. 150 machine instructions is not very much C code so you have to be careful.

If you take too long in the ISR, your main code will slow to a crawl, if you take longer then the timer cycle time, you will either effectively never execute the main code and eventually suffer a crash of the system stack.

Measuring the Interrupt load
Because I did want to have a very fast timer ISR, I needed a way to measure how much load I was placing on the available resources. I devised a trick that works to estimate the load and lets me output a measurement on the serial port. As I work on the timer ISR I can keep track of the interrupt load.

The timer was not placed in a mode where it reloads automatically. This means the ISR must reload the timer for the next timeout interval. It would be more accurate to have the timer auto reload but using this mode, we can measure the time we spend in the ISR and correct the timer load value accordingly. The key is that we get reasonable accuracy with this correction but we also get a number that shows us how much time we are spending in the ISR.

The trick is that the timer keeps ticking even though it's overflowed and interrupted. At the end of our ISR we can capture the current

count in the timer. This value represents the time it took for us to get to that point in the code. It's the sum of the time it took to get into the interrupt routine and to execute the code in the ISR. There would be some error as the time for the instructions to reload the timer are not accounted for but we could correct for that empirically. In fact, it's why I used 257 in the load value math instead of 256. I discovered empirically that this gave me a better result. The extra count compensates for the timer reload instructions.

Timer2 ISR
The ISR for the Timer2 overflow interrupt is shown below.

```
#define TOGGLE_IO 9 //Arduino pin to toggle in timer ISR

//Timer2 overflow interrupt vector handler
ISR(TIMER2_OVF_vect) {
  //Toggle the IO pin to the other state.
  digitalWrite(TOGGLE_IO,!digitalRead(TOGGLE_IO));

  //Capture the current timer value. This is how much error we
  //have due to interrupt latency and the work in this function
  latency=TCNT2;

  //Reload the timer and correct for latency.
  TCNT2=latency+timerLoadValue;
}
```

Timer2 ISR walk through
The function is short and its primary job is to toggle an I/O pin. After it does the toggle, it captures the current timer count and uses it to correct for latency when the counter is reloaded. The latency value is a global which the main application can monitor for the load measurements mentioned above. It is the number of counts at 2MHz that it took for this ISR to do its function.

Remember that the ISR needs to be short as it is called every 20uS when we run the timer for 50KHz. You can do more in the ISR but you need to find a balance between the interrupt interval and the amount of work done in the ISR. The latency value will help with this as described below.

My testing shows an average latency of about 20 ticks which works out to about a 45% processor load. This means that because of the ISR, the main code will execute about 45% slower on average. Not a big deal but notice that the only work done in the ISR is to toggle an I/O pin. This cost us almost half of our processor time! The cost is due to the time it takes to service interrupts, execute the digitalWrite/digitalRead functions and complete the timer reload process.

The ISR would be much faster if we directly accessed the I/O pin with port registers. But using the more generic library functions were easy to use and if I really did only need to toggle the I/O pin, the 45% cost would be OK with me.

Main Code, Setup()
The setup function is called by the Arduino system code once at program start. It initializes the I/O and the timer. It also outputs to the serial port showing that the code is running.

```
void setup(void) {
  //Set the pin we want the ISR to toggle for output.
  pinMode(TOGGLE_IO,OUTPUT);
```

```
  //Start up the serial port
  Serial.begin(9600);

  //Signal the program start
  Serial.println("Timer2 Test");

  //Start the timer and get the timer reload value.
  timerLoadValue=SetupTimer2(44100);

  //Output the timer reload value
  Serial.print("Timer2 Load:");
  Serial.println(timerLoadValue,HEX);
}
```

Main Code, Setup() Walk Through
Setup starts by setting the toggle pin to output so we can toggle it in the ISR. Then it activates the serial port and prints some text to show we are alive.

Next the SetupTimer2 function is called with the frequency set at 44100Hz, a common sound sampling frequency. The return value is stored into a global named timerLoadValue for future use by the ISR.

Lastly Setup prints out the timerLoadValue so we can confirm that it is within reason.

At this point, the timer is running and our ISR function is being called at the specified rate. If you hook up an o-scope you will see the pin toggling, generating a frequency that is 1/2 the timer interval. Its 1/2 because we set the pin low on one ISR pass then high on the other.

Main Code, Loop()
The loop function is called over and over as the program runs. Each time loop returns it is called again. The code here looks complex but really all we are doing is averaging the latency value from the timer2 ISR and outputting the measurements after we get 100 samples.

Notice that the loop function needs to do nothing related to the toggling of the I/O line. This is all handled by the ISR leaving the loop function to do other things without much regard to the processes occurring in the ISR. Just like with the serial port, you don't have to worry about loading the next character into the UART when it is ready. You just do your thing and the serial port is handled in the background. This is one of the great things about interrupt driven programs. The functions occur on an event basis and are decoupled from your application code.

```
void loop(void) {
  //Accumulate ISR latency every 10ms.
  delay(10);

  //Accumulate the current latency value from the ISR and increment
  //the sample counter
  latencySum+=latency;
  sampleCount++;
```

```
 //Once we have 100 samples, calculate and output the measurements
 if(sampleCount>99) {
  float latencyAverage;
  float loadPercent;

  //Calculate the average latency
  latencyAverage=latencySum/100.0;

  //zero the accumulator values
  sampleCount=0;
  latencySum=0;

  //Calculate the Percentage processor load estimate
  loadPercent=latencyAverage/(float)timerLoadValue;
  loadPercent*=100; //Scale up from ratio to percentage;

  //Output the average Latency
  Serial.print("Latency Average:");
  Serial.print((int)latencyAverage);
  Serial.print(".");
  latencyAverage-=(int)latencyAverage;
  Serial.print((int)(latencyAverage*100));

  //Output the load percentage estimate
  Serial.print(" Load:");
  Serial.print((int)loadPercent);
  Serial.println("%");
 }
}
```

Main Code, Loop() Walk Through

The loop function starts by delaying 10mS. (Note that we could not use a delay like this if we needed to toggle the I/O pin at a high rate without an interrupt). The 10mS delay just controls how fast we take latency samples and output measurements. Since we take 100 measurements and each one is spaced by the 10mS delay, we output a result every 1 second.

Next the latency value from the ISR is accumulated in the latencySum global. We just grab the current latency value and add it to what we already have. We also increment a counter that keeps track of how many samples we have accumulated.

Now we check to see if 100 samples have been accumulated. If not, we skip the rest of the code and return. If we have 100 samples, then we get an average by dividing the accumulated latency by the sample count and store the result in latencyAverage. After we do this we clear the accumulator and the sample count so it can start again.

Now that we have a good latency measurement we can calculate the processor load estimate. We know the timer will timeout every time it counts from our reload value to 0xFF then back to 0×00, this is the overflow rate. The load percentage should be the latency/ticks in the ISR. This value is calculated and output as a measurement so we can see the effects of our ISR code.

The measurements are output and the loop function returns to be called again and wait 10mS. Again I point out that the main code has nothing to do with the job of toggling the I/O pin. It is free to use delay calls for its timing and is only affected in terms of how fast it will execute. The time it gets is the left over time from all the background ISR processing, for this timer ISR and the others that are always active, Timer0 and serial for example.

In the example code, the timer is loaded with D4 hex or 212. That means it will interrupt every time it ticks 44 times. We know that while the processor is executing the ISR code the timer ticks about 20 times so there are only about 24 more ticks before it's back in the ISR again. That 24 ticks worth of time is the left over time our main code gets to execute. So for a total time of 44 ticks between interrupts we spend 20 ticks in the ISR leaving about 24 ticks worth for the application. This works out as about 45% of the processor used up in the ISR.

We also know that about one tick is used up in all the ISR processing code that is not measured in the latency value. This is because I needed to add one to the timeout interval to correct for this in the code described above. We really only get about 23 out of the 44 ticks for the application. I ran the same code with the pin toggle commented out, the latency averaged at about 3 ticks or a 6% load. The ISR itself, without serving any useful purpose, uses about 6% of the available resource. The digital read and write functions eat up the rest of the time.

Summary
If you have read this far you should have a basic understanding of interrupts, why they are very useful and important, and how they affect the main code. You also have some example code to build on for your projects. Be careful how much work you do in a fast timer ISR. If you run the timer at a lower rate, then of course your problem is reduced. I used extreme cases here to give you an idea of what to watch for.

Comments Please:
I would like to hear your thoughts on interrupts in general and using the timers with the Arduino.

- Did I miss something important?
- Was this article useful to you?
- Should I do more articles like this?
- Are there some other good links to share?
- Have you used interrupts on the Arduino?
- Have any links for projects that make use of interrupts?

Posted in Arduino, Discovering, Electronics Links, Microcontroller, Projects.

By admin
November 24, 2007
27 comments

27 Responses

Stay in touch with the conversation, subscribe to the RSS feed for comments on this post.

1. dfowler said
   on November 25, 2007

Follower found a bunch of typos and a few grammer errors in my post. I have fixed this and thank him greatly for the thorough review.

2. fatlimey said
on November 26, 2007
Good to see an Arduino article where the Arduino is used as a *computing device*, not just as a simplified programming environment. I'm a low-level software guy who is learning electronics and it's nice to see our side represented for once. Keep up the good work!

3. dfowler said
on November 26, 2007
Hey fatlimey,

I am also a low-level hardware/software guy. I think the Arduino is a great start for hobbyist but am very intereted in taking the reins of the AVR using the nice interface to GCC that the Arduino IDE provides.

4. tat said
on November 26, 2007
very cool article. love the ones on sound as well and can't wait to see where it's going. thought the code below could be interesting to all. it basically reads analog0 in free running mode (and left adjusted so i only read the 8 MSBs) and reproduce on an 8-bit R2R on pins d0-7. it works but the output is VERY noisy (i'm assuming it comes from the R2R). with dv set at 205, it runs at about 40khz.
for info, i send live radio into analog0. cheers.

```
#include
byte v, dv;

ISR(TIMER2_OVF_vect) {
  TCNT2 = dv;
  PORTD = v;
}

void setup()
{
  UCSRB = 0; // turns off rx/tx
  DDRD = 0xff; // digital 0-7 as output
  dv = 205;

  ADMUX = B11100000; // Vref = 2.56v, result left adjusted, input = analog0
  ADCSRA |= (1 < < ADFR); // free running mode
  ADCSRA |= (1 << ADSC); // start conversions

  cli();
```

```
  TIMSK = B00000000; // all interupts OFF
  TCCR2 = B00000010; // normal mode, prescaler = 8
  TIMSK = B01000000; // enable Timer2 overflow interupt
  sei();
}

void loop()
{
  v = ADCH; // since R2R is only 8 bits, we only read the 8 MSB bits instead of the 10 bits..
}
```

5. gonium said
   on November 27, 2007
   Hey,

   someone actually uses my findings… Nice!

   -Mathias

6. Mike said
   on November 29, 2007
   Code needs:

   #include

   at the top in windows. However,this does not seem work on my Ubuntu linux system?

   Mike

7. Mike said
   on November 29, 2007
   System will not take the “greater than” and “less than” characters

   So this should have said (putting in the greater and less than signs

   #include “less than” avr/interrupt.h “greater than”

   Hope thats clear

8. NiñoScript said
   on December 18, 2007
   \

maybe using “\\”

9. NiñoScript said
on December 18, 2007
< >

10. NiñoScript said
on December 18, 2007
Sorry for the tripost
i was trying to write the greater than and less than symbols
you have to use some html here…
write & then lt (for less than) or gt (for greater than) and finish with an ; (without spaces in between), then your code will look like the following:

#include <avr/interrupt.h>

btw, i don’t know why it doesn’t work in ubuntu, i guess it should… could you check the avr folder and see if you have the interrupt header?

11. Paul said
on January 1, 2008
Hi David,

Great article. It’s great to have a place to go to get this explained on a fine-grained level. Now to find the time to rerun your code!

I’m looking at where this is going with a lot of interest. Is the Arduino fast enough to really get some sound synthesis done? Seems like the answer is going to be a qualified yes.

Paul

12. dfowler said
on January 1, 2008
Thanks Paul.

I will be getting back to the audio article serries soon. Been tied up with the holidays. Yes, I think we can do some neat sound generation with the Arduino.

13. ben said
on January 12, 2008
Great Article! I was looking for exactly this, you have been immeasurably helpful!

I know you mentioned that the ISR could be much more efficient if you toggle the pin directly, and I wanted to add that just a quick optimization can yield some nice results:

for instance, in the ISR, instead of doing a digital read (which is processor expensive), keep track of the pin value in a variable like so:

digitalWrite(TOGGLE_IO,flipFlop);
flipFlop = flipFlop ^ 0×01;

This brought the latency down to around 14 ticks. I got it to run up to 90kHz easily after this quick change.

Cheers! and thanks again!

14. dfowler said
on January 12, 2008
Ben,

Yes. You can also get a big boost by replacing the digitalWrite with direct I/O.

David

15. John said
on January 13, 2008
Thanks much for the great tutorial!

It seems it may be what I need to communicate with a chip about 100 times / sec.

However, using the example, when I call SetupTimer2 with a an argument such as 100 the load goes way up (45%) and I don't think I'm getting 100Hz.

I put a piezo on pin 9 and also a sound card scope. The freq seems to be all over the place – even with higher values.

I can't figure out what I'm doing wrong, any suggestions appreciated.

John

16. John said
on January 15, 2008
Well, since I posted I learned more about the prescalers and more. Using the 1024 prescaler with a reset of 100 worked out about right. Thanks again.

17. dfowler said

on January 17, 2008
John,

Glad you found a solution.

18. JMG said
on January 22, 2008
Thanks so much for the fantastic tutorial! You do a great service for the hobbyist community, not only by posting this information, but by responding so diligently to comments!

One thing I'm not sure I understand, though, is the need to correct for the timer load and ISR latency. Wouldn't this need be obviated if the timer reset itself automatically? Do you simply do the manual measuring in specific example, because you want to measure what the load and latency are, or is there more to it than that?

19. dfowler said
on January 22, 2008
JMG,

You are correct, if the timer autoloaded then you probably would not need to correct for latency. There are some cases where the processor could have been delayed getting to the ISR due to interrupts being disabled. The correction method above would maintain the average timing where the autoload would tend to have longer times on average.

The real reason that I use the latency trick above is to measure the load my ISR is putting on the uC. Once you have the few instructions required to do the load process it really is not a big problem and the benefit is that you can always measure latency and/or the ISR load. This is especially useful when you are developing the ISR code with a fast timing cycle.

20. Steve Hobley said
on January 24, 2008
Hi,

I've been trying to set the prescaler to /32 but the timer no longer fires when I set the LSB to 0,1,1 – when I set them to 1,0,0 (/64) it works OK.

// Set to /64

TCCR2B |= (1

21. xSmurf said
on January 27, 2008
Is it possible to change the duty cycle at a specific frequency to use this for fast PWM? I'm looking into building a switching led driver and need to generate a 33% duty cycle at 13Khz using the internal osc. at 8Mhz.

22. dfowler said
on January 27, 2008
xSmurf,

Yes, it is described in the article on PWM sound generation at the link below.
http://www.uchobby.com/index.php/2008/01/01/pwm-sound-generation/

The source code is on the Arduino playground and runs the PWM at about 60Khz. It should be easy to change this to meet your needs.

23. dfowler said
on January 27, 2008
Steve,

Are you still having problems? Maybe I can look at your code to see what's wrong. You can send it to dfowler@uchobby.com It maybe a few days before I can look at it.

David

24. achterwerk said
on February 28, 2008
Thanks for pointing me in the right direction.
From your code I worked towards a squarewave oscillator running on the compare interrupt of timer1. The difference is that I used the compare mode of the Timer instead of resetting the counter by the program. It is running stable with a sample rate of 44,1 kHz with the rimer running at 16 MHz driving 6 IO Pins connected to an R2R-DAC. According to my tuner the frequency of the oscillator is quite exact and stable over a wide range. But still some coding to do towards a small synthesizer.

Thanks a lot.

Hecke

25. Dane said
on November 14, 2008
How do I use a interrupt on an arduino for converting a pitch into a programmed colored LED. I am using several different colors.

26. Jin said
on January 1, 2009
I don't think your load calculation in the code is correct (it is different from the explanation you give in the text). You are treating timerLoadValue as if it were the total length of time available between interrupts, but shouldn't it be 256-timerLoadValue? If latency is 20 and timerLoadValue is 200, your could would report 10% load, when it should be 36%.

27. Jin said
on January 1, 2009
I've also instrumented the loop to sample max latency as well as the average. Doing the exact same instructions every loop, I see a max latency of 22-27, with the average between 6 and 9. Why such variability? It's doing nothing but reading the PIND register. Maybe the main loop's Serial prints are interfering with the register interrupts and that's being recorded? If latency is > 256-timerLoadValue, won't the next interrupt happen at a somewhat random interval?

Subscribe

Site RSS feed

Tag Cloud

Adaptors Arduino Audio AVR Breadboard Contest Development IPhone LED Links Logic Making Microcontroller Mods Nixie Tubes Parts PCB PIC Project Prototype Review Robot Scrounging Tips Tools Tutorial TV

Ads by Google

Cheapest Arduinos Online
Buy the Arduino for $27 And the Mega for $49!
www.liquidware.com

Freescale Analog Chips
Leading Manufacturers of Analog Processor & Microcontrollers
www.Freescale.com/Analog

LOGO Programmable Relay
Siemens LOGO! logic relay Built-in: Timers, Counters, Clock
www.sea.siemens.com/logo

ARM Single Board Computer
$29.95 module, 21 IOs, 7 A/Ds and easy to program in BASIC or C
www.coridiumcorp.com

Electronic Timers
Wireless electronic timing for equestrian events and other sports!
farmtek.net

Blogroll

- BricoGeek
- Circuit-Projects
- DIY Live
- DIY4Fun
- EEBeat
- Electronics Lab
- Electroniq
- Elektronik
- Embedds
- Geeksinside
- GrinanBarrett
- Hack A Day

- Hacked Gadgets
- justDIY
- Make:Blog
- N5EBW Crazy Man
- Open Circuits
- Pinotronics
- Scienceprog
- Solder in the Veins
- Spark Fun
- Youritronics

This work is licensed under a Creative Commons Attribution 2.5 License.
Attribution with link to www.uCHobby.com. Contact info@uCHobby.com for site information.