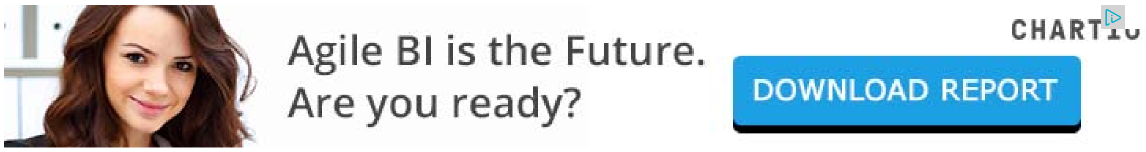
Home Resources Contact About

EngBlaze

Arduino, AVR, and hardware hacks, oh my.

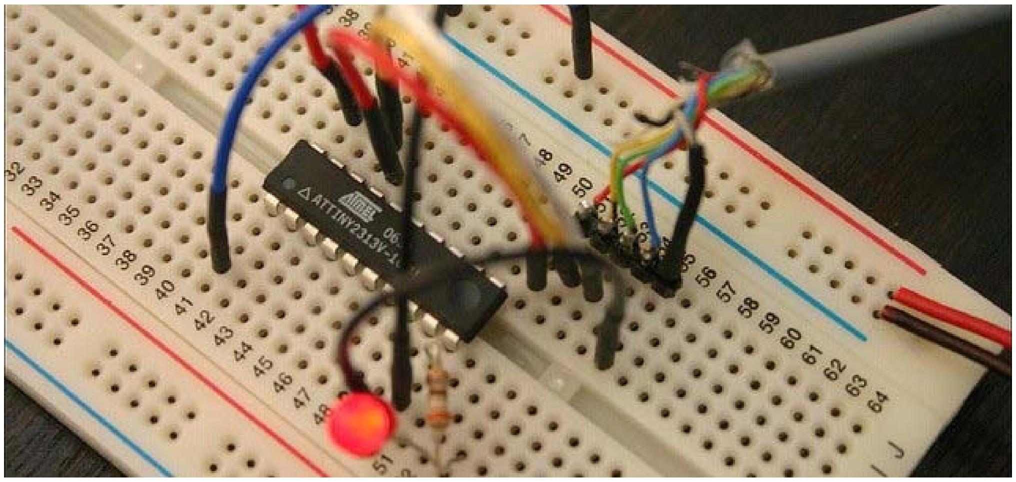
Arduino AVR Tutorials Books Programming Projects Prototyping



We interrupt this program to bring you a tutorial on… Arduino interrupts

TI's MSP430 16-Bit MCUs

Ultra-Low Power RISC Mixed-Signal Microprocessors for Portable Apps.



Ah yes… the wonderful and oft-misunderstood world of microcontroller interrupts. Are you looking to build a project that relies on very precise timing or needs to react quickly to an input? Then don't change that channel, my friend. In this tutorial we'll cover what interrupts are, what they do, and how to use them.

What is an interrupt?

On a very basic level, an interrupt is an signal that *interrupts* the current processor activity. It may be triggered by an external event (change in pin state) or an internal event (a timer or a software signal). Once triggered, an interrupt pauses the current activity and causes the program to execute a different function. This function is called an interrupt handler or an *interrupt service routine* (ISR). Once the function is completed, the program returns to what it was doing before the interrupt was triggered.

ISR? Signals? What is all of this nonsense for?

If you're new to the world of software development, you might wonder why all this complication is necessary just to respond to external events. After all, you can check the state of external pins at any time, or create your own timers, right?

You certainly can do all of these things in your main code, but interrupts give you a key advantage – they are *asynchronous*. An asynchronous event is something that occurs outside of the regular flow of your program – it can happen at any time, no matter what your code is crunching on at the moment. This means that rather than manually checking whether your desired event has happened, you can let your AVR do the checking for you.

Let's use a real-world example. Imagine you're sitting on your couch, enjoying a frosty brew and watching a movie after a long day. Life is good. There's only one problem: you're waiting for an incredibly important package to arrive, and you need it as soon as possible. If you were a normal AVR program or Arduino sketch, you'd have to repeatedly stop your movie, get up, and go check the mailbox every 5 minutes to make sure you knew when the package was there.

Instead, imagine if the package was sent Fedex or UPS with delivery confirmation. Now, the delivery man will go to your front door and ring the doorbell as soon as he arrives. That's your interrupt trigger. Once you get the trigger, you can pause your movie and go deal with the package. That's your interrupt service routine. As soon as you're done, you can pick up the film where you left off, with no extra time wasted. That's the power of interrupts.

The AVR chips used in most Arduinos are not capable of parallel processing, i.e. they can't do multiple things at once. Using asynchronous processing via interrupts enables us to maximize the efficiency of our code, so we don't waste any precious clock cycles on polling loops or waiting for things to occur. Interrupts are also good for applications that require precise timing, because we know we'll catch our event the moment it occurs, and won't accidentally miss anything.

Types of Interrupts

There are two main types of interrupts:

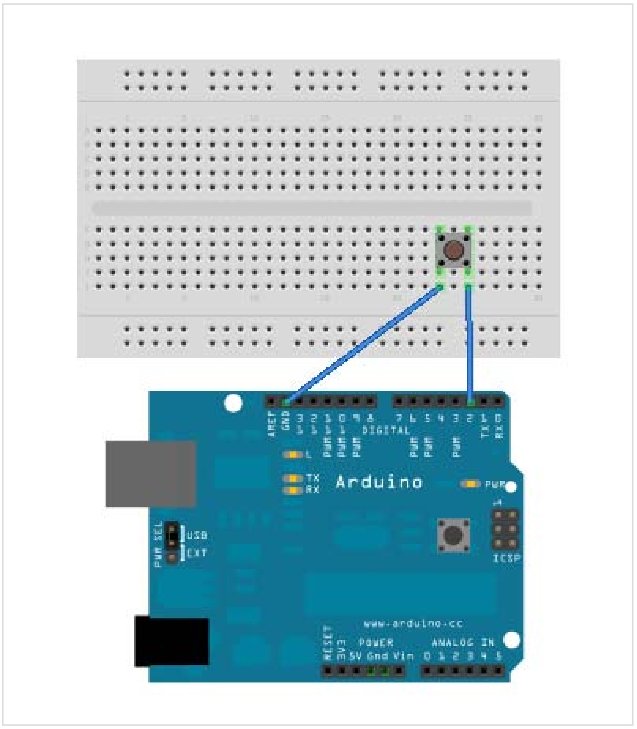
- Hardware interrupts, which occur in response to an external event, such as an input pin going high or low
- Software interrupts, which occur in response to an instruction sent in software

8-bit AVR processors, like the ones that power most Arduino boards, aren't capable of software interrupts (though you can emulate them by changing a pin's state in software). We'll focus on hardware for now. Most tutorials out there talk about handling external interrupts like pin changes. If you're using an Arduino, that's the only type of interrupt the Arduino "language" supports, using the attachInterrupt() function. That's fine, but there's a whole different category of hardware interrupts that rely on the AVR's built in timers, which can be incredibly useful. We'll cover external interrupts in this tutorial, then go over timer interrupts in a followup tutorial, since there's enough information to warrant splitting things up.

How to get an interrupt to do what we want – Interrupt Service Routines

Every AVR processor has a list of interrupt sources, or *vectors*, which include the type of events that can trigger an interrupt. When interrupts are enabled and one of these events occur, the code will jump to a specific location in program memory – the interrupt vector. By writing an ISR and then placing a link to it at the interrupt vector's memory location, we can tell our code to do something specific when an interrupt is triggered.

Let's implement this using a simple example: detecting when a pushbutton has been pressed, and performing an action based on that press. In this tutorial, we'll use a standard Arduino board to make things consistent when we refer to pins and our program setup. You can use these techniques with a plain AVR too, you just need to check the datasheet to make sure you know which pins you need. Here's our example circuit:



Implementing an interrupt in a program

In order to successfully use an interrupt, we'll need to do three things:

1. Set the AVR's Global Enable Interrupts bit in Status Register
2. Set the interrupt enable bit for our specific interrupt vector (each vector has it's own on/off switch)
3. Write an ISR and attach it to our target interrupt vector

Starting with the first step, we'll include the interrupt library from avr-libc, then use an avr-libc function to set our global interrupt enable bit. Next, we need to enable the interrupt we want. Most 8-bit AVR's like the ATMega328 have 2 hardware interrupts, INT0 and INT1. If you're using a standard Arduino board, these are tied to digital pins 2 and 3, respectively. Let's enable INT0 so we can detect an input change on pin 2 from a button or switch.

```
#include <avr/interrupt.h>
                              //
void setup(void)
{
    sei();                    // Enable global interrupts
    EIMSK |= (1 << INT0);     // Enable external interrupt INT0
```

Next, we need to set the sensing method of our interrupt. For a pin change interrupt, we have four options: rising edge, falling edge, any logical state change, or a low level on the pin. The default for INT0 and INT1 is to trigger on the pin being low level; you can use the datasheet to read more about how to set each method. Let's use falling edge for this example, just to show how it's done.

```
EICRA |= (1 << ISC01);    // Trigger INT0 on falling edge
```

Finally, we'll define an ISR that performs our desired task. Every ISR is defined as:

```
ISR({vector}_vect)
{
    // Perform task here
}
```

where {vector} is the name of our chosen interrupt vector. Again, the names of these vectors are defined in the processor datasheet; here, the one we want is EXT_INT0. For this example, we'll use the ISR to toggle the built-in LED on pin 13.

With our LED code added, here's how it looks all together:

```
#include <avr/interrupt.h>
                              //
void setup(void)
{
    pinMode(2, INPUT);
    pinMode(13, OUTPUT);
    digitalWrite(2, HIGH);    // Enable pullup resistor
    sei();                    // Enable global interrupts
    EIMSK |= (1 << INT0);     // Enable external interrupt INT0
    EICRA |= (1 << ISC01);    // Trigger INT0 on falling edge
}
                              //
void loop(void)
{
                              //
}
                              //
// Interrupt Service Routine attached to INT0 vector
ISR(EXT_INT0_vect)
{
    digitalWrite(13, !digitalRead(13));    // Toggle LED on pin 13
}
```

Double click anywhere in this code to select it so you can copy and paste into your own sketches.

Note that we can use the program's main loop() to do anything we want in the meantime, and it won't affect the functionality of our LED toggle.

Arduino interrupt functionality

There's an alternative way to implement INT0 and INT1 using the Arduino programming "language". I put "language" in quotes because although that's how the Arduino website refers to it, it's more like a wrapper around avr-libc to make certain things easier. But that's an article for another day.

For our example, we could use the attachInterrupt() function to enable the same interrupt in setup():

```
setup(void)
{
    attachInterrupt(0, pin2ISR, FALLING);
}
```

In this case, we define our own custom ISR (pin2ISR) and pass it as an argument. pin2ISR() would simply take the place of ISR(EXT_INT0_vect). This is often a bit simpler than using the raw avr-libc functions, and it does abstract away any difference in chip models. However, the AVR methods described earlier can be used with any interrupt vector, while attachInterrupt() only works with external interrupts.

Putting it all together

There are a few final things to keep in mind when implementing interrupts.

First, keep in mind that whatever you use your ISR for, it's good practice to keep it short, because while the ISR is executing, it's holding up the rest of your program. If you have lots of interrupts, this can slow things to a crawl. Also, by "the rest of your program", we do mean *everything*: Arduino functions like millis() won't increment, and delay() won't work within an ISR.

Next, if you want to modify any variables within your ISR, you'll need to make them global variables and mark them as volatile to ensure that your ISR has proper access to them. For example, instead of globally declaring

```
int myInterruptVar;
```

you would declare

```
volatile int myInterruptVar;
```

Finally, interrupts are normally globally disabled inside of any ISR (this is why delay() and millis() don't work). This means that if more interrupt events occur before your ISR has completed, your program won't catch them. This is another reason to keep ISRs short.

Wrapping up

That's about it for the basics. If you're ready to keep going, check out our followup article on timer interrupts. It builds on the concepts discussed here and helps you use the AVR's built-in timers to do useful things. In the meantime, implement this example and dive into the Atmel datasheets to discover other interrupt vectors. They sky's the limit!

If you enjoyed this tutorial, make sure to sign up for our email updates so you can get access to all of our hardware hacking tips and tricks. Happy hacking!

Like 54 Tweet 20 51

This entry was posted in **Arduino**, **AVR**, **Tutorials** and tagged **arduino**, **AVR**, **avr-libc**, **code**, **interrupts**, **tutorial** by **EngBlaze**.

13 THOUGHTS ON "WE INTERRUPT THIS PROGRAM TO BRING YOU A TUTORIAL ON… ARDUINO INTERRUPTS"

Js
on **October 31, 2011 at 1:46 pm** said:

Your AVR examples won't work because you have "#include" in them. You need to include the correct headers. Good article.

> Anonymous
> on **October 31, 2011 at 2:30 pm** said:
>
> Good call. We're using Syntax Highlighter for the code blocks but
> Wordpress still likes to mess with things that look like HTML tags if we're not careful. Annoyingly it's also stripping line breaks in the code too… have to look into that. For now I added blank comments to preserve the formatting. Thanks for the tip and kind words!

Dennis Mabrey
on **November 8, 2011 at 3:28 pm** said:

Two things:

You might also want to mention the PCInt and PCChangeInt libraries mentioned on the Arduino site here:
http://www.arduino.cc/playground/Main/PinChangeInt with samples here:
http://arduino.cc/playground/Main/PinChangeIntExample which gives you flexibility to use any pin for an interrupt.
Second, a common example of where you use interrupts is using quadrature encoders on motors.   As the motor spins two signals come in from the encoders which can be captured with interrupts to determine direction, speed, and amount of rotation of your motors.

Sample code can be found here: http://www.arduino.cc/playground/Main/RotaryEncoders

atif khair
on **November 19, 2011 at 12:51 am** said:

Good article that make some thing more effective well something going wrong when i am going to put headers.

Jaycon

J. Bryant Hill
on **December 2, 2011 at 2:50 am** said:

thx much. stoked my interest

Rick
on **February 13, 2012 at 6:43 pm** said:

If you are using the ATmega328P (i.e. Arduino UNO) you must change the line:

ISR(EXT_INT0_vect)
to
ISR(INT0_vect)
Otherwise the code compiles OK, but the ISR will never get called.

Rick

> Joe
> on **August 11, 2012 at 1:10 pm** said:
>
> Thanks. Made the change to ISR(INT0_vect)…… and got it to work on my OSEPP Uno R3 Plus. Learned a lot trying to get it to work though.

GreyGnome
on **November 28, 2012 at 10:09 am** said:

Technically you are correct that the ATmega328p does not have software interrupts, however if you set a pin as an output and then change its state, you can get the equivalent of a software interrupt. That means of course that it's no longer a general purpose pin, but the designers of the chip made it this way for just this purpose: to emulate a software interrupt.

> **EngBlaze**
> on **December 6, 2012 at 7:19 pm** said:
>
> Fair enough, that's a good clarification. We'll update the tutorial, thanks for the tip.

GreyGnome
on **November 28, 2012 at 10:21 am** said:

Under “Types of Interrupts”, you say, “Most tutorials out there talk about handling external interrupts like pin state changes. If you’re using an Arduino, that’s the only type of interrupt the Arduino “language” supports, using the attachInterrupt() function.”

This is not true. You have basically declared that the Arduino only supports External Interrupts; a beginner is liable to think that there are only 2 interrupt pins available on the Arduino (as I did at one time). As already mentioned here, there is the PinChangeInt library which can enable interrupts on any GPIO pin. But even if one gets pedantic and declares that “That is not an Arduino language feature,” which is true, the mechanism is still available for programmers to configure interrupts on whatever port they want; there are examples on the Internet that will show you how to enable a Pin Change Interrupt and they use basic C coding techniques without a library, which is essentially just the Arduino language.

Disclaimer: Though I maintain the PinChangeInt library, I believe that having more than 2 interrupts available on the ATmega328p is incredibly useful.

> **EngBlaze**
> on **December 6, 2012 at 7:16 pm** said:
>
> The statement you quote is absolutely true. If you are staying within the confines of the Arduino core library, you only have access to the ATMega’s two external interrupt pins. That’s not pedantic, it’s a fact. There are examples on the internet to do anything and everything that the Arduino library exposes using pure C. The point of the platform is to abstract at least some complexity away from beginning programmers.
>
> However, since the tutorial describes how to enable said interrupts using avr-libc, it is reasonable to suggest that it should do the same for pin change interrupts as well. PinChangeInt also seems like a viable solution. I’ll see if we can provide an update with information on both resources.

fixedProblem
on **August 4, 2014 at 9:26 am** said:

Thanks for the tutorial. I spent a while trying get the LED to light up until I realized you need to add

pinMode(13, OUTPUT);

to setup(). Hope this helps some others.

> **EngBlaze**
> on **October 30, 2014 at 8:33 am** said:
>
> Hi fixed,
>
> You’re right, that’s been clarified in the code. Thanks for the feedback!