



External Interrupts on MPIDE part 1: attachInterrupt()

by **JayWeeks** on October 17, 2015

Table of Contents

External Interrupts on MPIDE part 1: attachInterrupt()	1
Intro: External Interrupts on MPIDE part 1: attachInterrupt()	2
Step 1: What You'll Need	2
Step 2: What Interrupts Aren't: Polling	3
File Downloads	3
Step 3: What even is "Interrupt"?	3
File Downloads	4
Step 4: Try Out The Code For Yourself!	4
Step 5: Debouncing	5
Step 6: AttachInterrupt()	5
Step 7: Interrupt Service Routine	6
Step 8: Volatile Variables	6
Step 9: Void loop()	6
Step 10: Where to Find Your Interrupts!	7
Step 11: Where To Go From Here?	7
Related Instructables	7
Advertisements	8
Comments	8



Author: JayWeeks

I build robots out of boxes! I love teaching what I've learned and seeing people add their own ideas to what they've learned. Nothing excites me more than seeing a student really take an idea and run with it!

Intro: External Interrupts on MPIDE part 1: attachInterrupt()

Hey! So it's been a while since I made the [Metal Wheels for Cheap Robots](#) tutorial, but that's because I was untangling interrupts for you guys. I found a couple of ways to do it, but I'm going to start with the simplest method first, so I can move on to the next project! I'm really excited you guys!

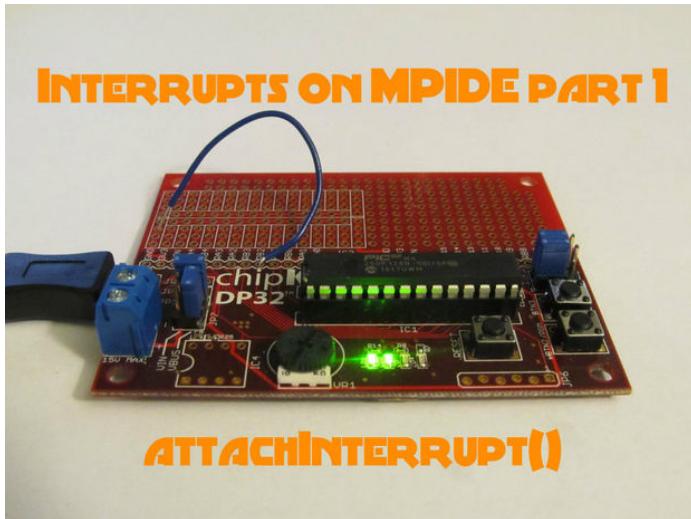
Let's get started!

~~~~~

For more Instructables on building cheap robots, please check out the [For Cheap Robots collection](#)!

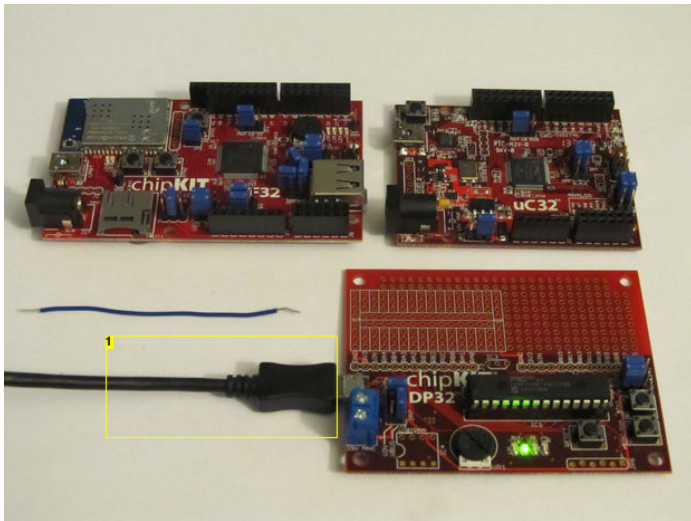
For more things that I've done, you can check out my [profile page](#)!

For more info from Digilent on the Digilent Makerspace, check out the [Digilent blog](#)!



## Step 1: What You'll Need

For this tutorial, you're not going to need much. Just one of the chipKIT boards from Digilent, like the DP32, uC32, or WF32, an appropriate programming cable, and a bit of wire. Everything else will be provided right on the board!



### Image Notes

1. It's worth pointing out that the DP32 uses a micro USB cable, while the WF32 and uC32 use a mini USB. If what I just said makes no sense to you, google "micro USB vs mini USB".

## Step 2: What Interrupts Aren't: Polling

You are in a classroom and the professor is going through her lecture about interrupts. She wants to make sure everybody understands the lecture, so after every sentence, she stops and asks each of you if you have a question... individually.

"Greg, do you have a question? No? Okay."

"Sarah, do you have a question? No? Okay."

Mark, do you have a question? No? Okay."

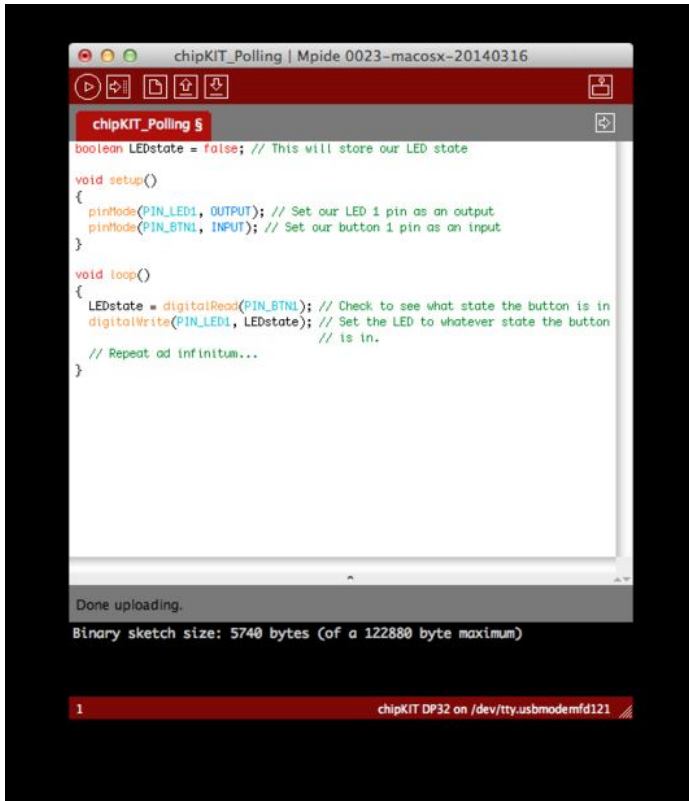
etc.

This is called polling, and most of the programs that you've made use polling. Polling is when you check your inputs over and over again to see if they've changed.

Check out the chipKIT\_Polling example code. In this code, we ask the on-board button over and over again, setting our LED to whatever state the button is at currently.

This works fine for most code, but if you've got a lot of inputs to check (as in our classroom example) it can take a very long time and slow down your code. It also runs into trouble when you have time sensitive applications, and need to execute special code immediately.

Now let's see what an interrupt looks like.



```
chipKIT_Polling | Mpile 0023-macosx-20140316

chipKIT_Polling $
boolean LEDstate = false; // This will store our LED state

void setup()
{
  pinMode(PIN_LED1, OUTPUT); // Set our LED 1 pin as an output
  pinMode(PIN_BTN1, INPUT); // Set our button 1 pin as an input
}

void loop()
{
  LEDstate = digitalRead(PIN_BTN1); // Check to see what state the button is in
  digitalWrite(PIN_LED1, LEDstate); // Set the LED to whatever state the button
                                   // is in.
  // Repeat ad infinitum...
}
```

Done uploading.

Binary sketch size: 5740 bytes (of a 122880 byte maximum)

1 chipKIT DP32 on /dev/tty.usbmodemfd121

## File Downloads



chipKIT\_Polling.pde (518 bytes)

[NOTE: When saving, if you see .tmp as the file ext, rename it to 'chipKIT\_Polling.pde']

## Step 3: What even is "Interrupt"?

So, our previous classroom example wasn't very realistic. In a real classroom, your professor wouldn't stop and ask every person if they have questions. Here's a more realistic example:

You're in a classroom and the professor is lecturing on interrupts. You're not sure you understand something, so you raise your hand. The professor sees your raised hand, stops her lecture, and answers your question. Once your question has been answered, the professor picks up again where she left off and continues her lecture.

This is, in a very real sense, how interrupts work.

Check out the chipKIT\_Interrupts\_1 example code. In this code, we don't keep asking our input what its status is. Instead, we set up the interrupt on one of our pins and wait for that to be triggered. When the interrupt is triggered, our microcontroller stops executing the code in "void loop()", jumps to the code for that interrupt (called an Interrupt Service Routine or ISR) and executes it. Once the ISR is done executing, we jump back to where we left off in the main code.



## File Downloads



chipKIT\_Interrupts\_1.pde (1 KB)

[NOTE: When saving, if you see .tmp as the file ext, rename it to 'chipKIT\_Interrupts\_1.pde']

### Step 4: Try Out The Code For Yourself!

Upload the interrupts example to your board.

To use this code, you'll need to connect a wire to your interrupt pin.

On the DP32, interrupt 2 is on pin 13, also labeled RB2 on some boards. On the uC32 and WF32, interrupt 2 is on pin 7.

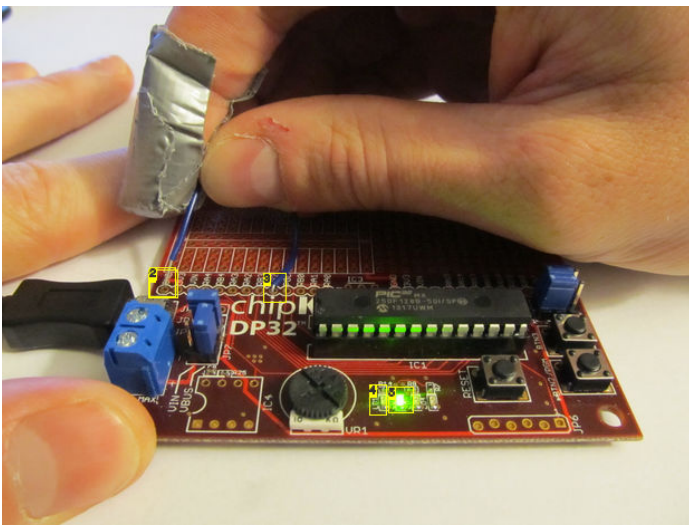
Use the wire to tap on the ground pin.

When you do this, on any of these boards, you'll see LED 1 change whenever you tap ground.

(On the DP32, you'll also see LED 2 turn on and off, but this is because pin 13 on the DP32 is also connected to LED 2, so this is useful to see what the voltage on pin 13 is at.)

When the voltage on your interrupt pin goes low, that'll trigger your interrupt to change the state of the LED. That way, instead of changing to match whatever the state of your input is (like with our polling example), your interrupt will act like a switch, toggling the LED states.

This can be a lot to absorb all in one go, so next we'll go through each part of the code and dissect exactly what it's doing and why. That way you'll be able to understand and use it for your own applications!



#### Image Notes

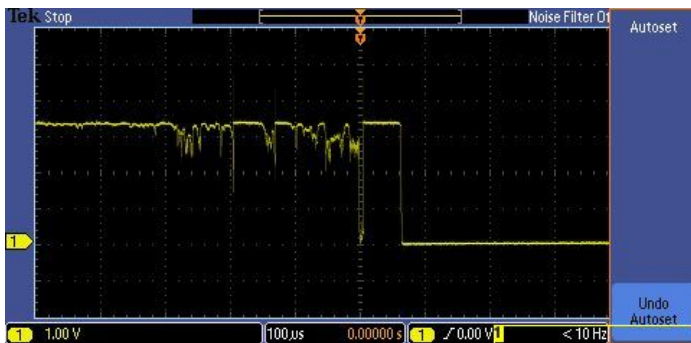
1. Ground
2. Ground
3. RB2/pin 13
4. LED 1
5. LED2

### Step 5: Debouncing

Now that you've played with the code a little, you may notice your interrupts behaving a little erratically. Maybe sometimes the LED doesn't seem to switch at all, or maybe you see it turn off, only to turn back on again. Maybe it just sits and flickers.

This is called "bouncing" and it's caused because the physical world isn't as nice and discreet as the electronics world sometimes wants it to be. Basically what is happening, is your wire isn't making a complete connection right away, so the voltage "bounces" between high and low before finally settling. "Debouncing" is the practice of accounting for and correcting for this erratic physical behavior.

There are ~~several~~reallygood tutorials on the Digilent Learn site all about how to debounce buttons when you're using them normally in your main loop() function, but interrupts are a little different. It's too much to get into in this tutorial, so look for a tutorial on debouncing for external interrupts very soon! (I'll update this tutorial with a link, so don't you worry.)



#### Image Notes

1. This image comes from one of the Digilent Learn modules linked in this step.

### Step 6: AttachInterrupt()

This is the main function behind this interrupt code. Setting up interrupts and using them correctly can be very complex, but this little function does most of that for you. Let's pull apart it's inputs so you'll understand how to use it.

The [chipKIT](#) page documenting attachInterrupt(), labels it's inputs as "interrupt", "function", and "mode".

"Interrupt" refers to the number of the external interrupt you want to use. The DP32, uC32, and WF32 have four external interrupts that you can choose from, numbered 1 through 4 on the DP32, and 0 through 3 on the uC32 and WF32. Later I'll show you how to find what interrupts are connected to what pins on your board, but for now you just need to know their numbers.

"Function" refers to the name of the function that you want to be your ISR. When your interrupt is triggered, your microcontroller will execute the code in the ISR for that interrupt, and then return to the main loop where it left off. In our example, we called our ISR "LEDchange".

Finally, "mode" refers to what is also known as "edge polarity". Basically, you're telling the interrupt to trigger either when the voltage goes from high voltage (usually 3.3 volts) to low voltage (usually ground), or from low voltage to high voltage. Currently, our interrupt is set to "FALLING", otherwise known as a falling edge. This means it will trigger when the voltage falls from high to low. If you wanted it to trigger when the voltage went from low to high, you would use "RISING". You can also use "CHANGE" if you just want the interrupt to trigger whenever there is a change in voltage, either going from high to low, or low to high.

Edge polarity can be a little confusing, so try changing this code and seeing how your interrupt's behavior changes.

<http://www.instructables.com/id/External-Interrupts-on-MPIDE-part-1-attachInterrupt/>

```
attachInterrupt(2, LEDchange, FALLING); // Attach our ISR to interrupt 2
// and set it to trigger on falling
// voltage
```

## Step 7: Interrupt Service Routine

Like I mentioned in the previous step, we set our Interrupt Service Routine or ISR to our LEDchange() function. This function can be anything you want, but interrupts are very finicky things, so there are a couple of guidelines you'll want to follow.

First and foremost, ISRs must execute quickly. Let's go back to our classroom example. When the professor called on you for your question, what if you started having a long conversation with the professor? It would totally disrupt the class, and the professor wouldn't be able to get back to her lecture to finish it before class ended. Instead, you try to make your question very short, and your professor tries to give you a concise answer. The same is true for interrupts. They interrupt the flow of the normal program, so you'll want to get back to that as soon as possible.

DO NOT EVER USE DELAY() IN AN ISR. In general, the delay() function is considered sloppy programming. Basically, it freezes your microcontroller for a set amount of time. During that time, no other code can execute (except for other interrupts), and that's functionally wasted time. If you use delay() in an ISR, that ISR will take forever, which violates our first rule. It merits its own warning, however, because it is one of the fundamental tools that new programmers are introduced to during their very first blink program, and can easily trip you up when getting into more advanced subjects like interrupts.

This last rule is more of a personal preference than anything else. I have heard that sometimes other people have issues when trying to manipulate outputs in their ISR, like turning LEDs on and off. Generally I think of these as slow processes, so if at all possible I try to avoid manipulating outputs inside my ISRs. That's why we use the LEDstate variable to set the state of our LED, instead of setting it directly. However, interrupts are commonly used when communicating with other boards and such, so it's not always practical or even possible to keep from manipulating outputs, so this isn't a hard and fast rule. Use your best judgement!

```
// This is our Interrupt Service Routine
void LEDchange()
{
    LEDstate = !LEDstate; // If the interrupt is triggered, switch the LED state
                          // If it's high, it goes low. If it's low, it goes high.
}
```

## Step 8: Volatile Variables

I mentioned before that because I try not to manipulate outputs directly inside my interrupts, we use the LEDstate variable to manipulate our LED indirectly. This calls for a very special variable called a "volatile" variable.

MPIDE has code optimizers that help your microcontroller function more quickly and more smoothly. One of the things these optimizers will do is take variables that don't change and make them into constants, thus saving space in memory. Unfortunately, because our LEDstate variable isn't used in the main loop, and the function that does use it (LEDchange()) isn't called in the main loop, these optimizers will see LEDstate as a variable that doesn't change, and turn it into a constant, which causes problems when our interrupt tries to change it.

Now, we know that LEDstate isn't a constant, but our code doesn't. Thankfully, we can tell our optimizers that LEDstate isn't a constant, and we do that by making it into a "volatile" variable.

You might be thinking, then, that you should make all your variables volatile, to ensure your optimizers won't accidentally make them constants when it shouldn't, but this would be a bad idea.

Because of how volatile variables are stored in memory, they're a little slower to respond than normal variables. That means that if you call your variables frequently, they might slow your code down significantly. Thankfully, because most of our interrupt applications are called so rarely, this doesn't end up being an issue, but it does mean you only want to make the variables used in interrupts volatile.

```
volatile boolean LEDstate = false; // This will store our LED state.
// This way we don't waste cycles in our
// interrupt changing the LED.
```

## Step 9: Void loop()

This part seemed obvious, but I figured it's worth covering anyway, just to make sure that everything is perfectly clear.

The first thing that you might notice about our loop code is that all it does is set our LED pin to whatever LEDstate is. It never checks for any inputs or even changes LEDstate. If you didn't know that there was an ISR that would change LEDstate whenever the interrupt was triggered, you might think that this was useless code! In fact, this is the very reason why interrupts are useful!

Consider that this code could be anything! You no longer need to worry about checking your input states, because your ISR will handle that for you! Now you're free to run your code, knowing that at any given time LEDstate will always reflect your input correctly.

But interrupts can also complicate things as well! Because interrupts can occur at any time and without warning, that means that LEDstate can change at any time! If you have code that takes several steps to execute, you have to remember that LEDstate could change in between any given step, and account for that.

```
volatile boolean LEDstate = false; // This will store our LED state.
// This way we don't waste cycles in our
// interrupt changing the LED.
```



## Step 10: Where to Find Your Interrupts!

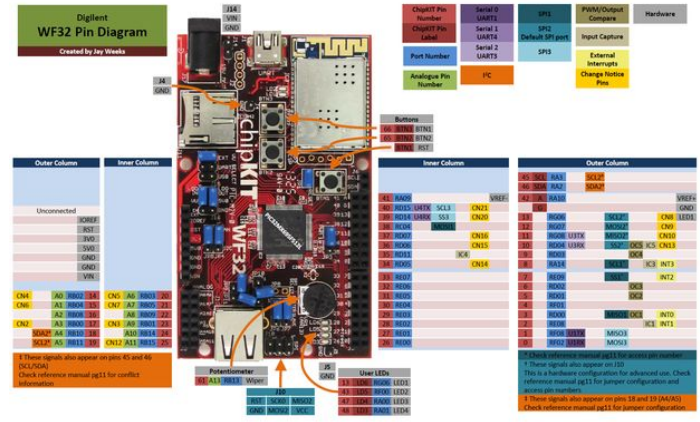
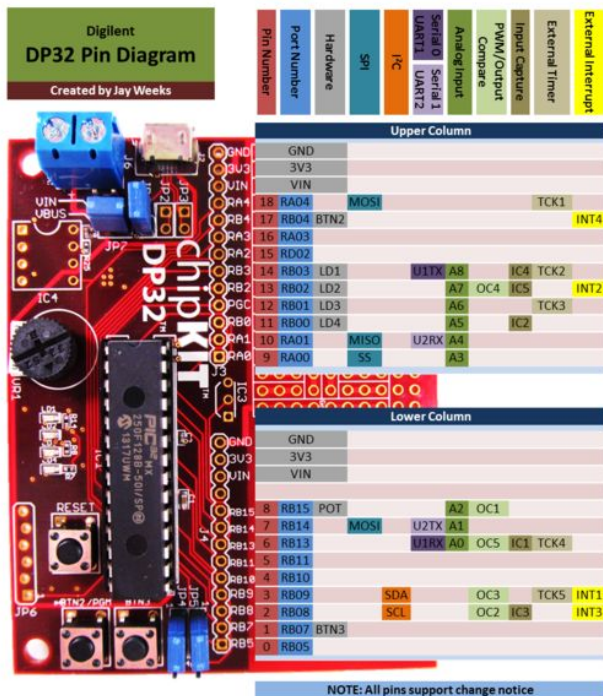
I mentioned earlier that the DP32, uC32, and WF32 all have four interrupts to choose from, but so far all I've shown you is interrupt 2. If you want to use a different pin for your interrupt, or if you want to have more than one interrupt, you'll have to know which pins each interrupt is associated with.

For the DP32 and the WF32, I've compiled these super-handly pinout diagrams that will not only show you where all the interrupts on these boards are, but everything else from LEDs to PWM to I2C!

I have to point out that the diagram for the WF32 is somewhat old, and I've since become aware of some inaccuracies in it. I do want to fix these sometime, but for now it'll still serve as a pretty solid reference for most applications.

"But Jay!" I can hear you mumble groggily from lack of sleep (I don't blame you, I've had a rough year too), "what about the uC32?"

Well, as it turns out the interrupt layout for the uC32 is the same as the WF32, but I'm also glad to point out that my pinout diagrams are not the only references for these boards. (Wouldn't that be terrifying?) I now direct you to the Digilent reference manuals for the DP32, WF32, and of course the uC32. In each of these, if you do a search for "external interrupts", you'll find the section that gives the interrupt numbers, as well as which pins the interrupts are attached to.



## Step 11: Where To Go From Here?

You may have noticed that this tutorial is auspiciously labeled "part 1", hinting at more parts to come! I'm looking forward to delving a little deeper into both external interrupts and timer interrupts with you, so keep an eye out for those tutorials in the near future!

I've also been hinting that these tutorials are working towards something neat associated with my Metal Wheels for Cheap Robots tutorial, and that's still very much in the works. Wheels, interrupts... I wonder where that's going? Some of you probably have already guessed what I'm moving towards. I'm super excited!

## Related Instructables



**Getting Started with the DP32 from Digilent** by JayWeeks



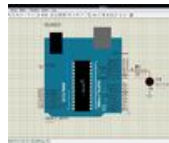
**Create External Interrupt in arduino** by Abdullah\_Al\_Mam



**Using the MPIDE Board-Defs** by JayWeeks



**Getting Started With the WF32!** by JayWeeks



**Create Internal Interrupt In Arduino** by Abdullah\_Al\_Mam



**Lab test bench-oscilloscope/wav** by ionescualexandru

## Comments