LESSON 5 – Digital Inputs

Greetings YouTubers, it is Rick the Tech Enthusiast here with the next Elegoo Lesson number 5 – Digital Inputs.

In this lesson we're going to learn a little about digital inputs using push buttons to turn on and off a LED. Specifically, we're going to use one of the buttons to turn on the LED and the other to turn off the LED. So let's get to it.

Before we begin we'll the following items from your Elegoo kit:

- Arduino UNO R3 board
- 5mm red LED
- 220 ohm resistor (Red, Red, Black, Black)
- Two push button switches
- Seven male-to-male jumper wires
- And the Breadboard

One small confession, I purchased these handy breadboard wires. They're precut to fit the breadboard's hole spacing and I'll just substitute a few to tidy up the circuit. Regular jumpers would work just as well.

First let's talk about push buttons or push button switches. They are mechanical devices with springy metal contacts inside them. When you press the button, they make a mechanical connection to allow electricity to flow. They come in a variety of configurations like normally open, normally closed, momentary, latching, and even multiple individual connection called poles. Ones we'll be using are momentary normally open and are commonly called tactile switches. Note that these tactile switches have four pins, two set of pins for each side of the switch. Notice how two of the pins sets bend in the same direction, those are the pins that are electrically connected when you push the switch.

On page 60, you'll see the schematic. And here's my version. Notice how the buttons are tied to ground. When they get pushed, they drive the Arduino pins 8 or 9 to ground, or LOW. The LED is tied with a protective resistor to pin 5.

On page 61 (ignore the page 3) you can see the breadboard wiring diagram. And here's my version. Notice that we'll connect the GND pin from the Arduino to the ground bus on the breadboard. All the other devices will simply connect to the ground bus for easy wiring. Also the tactile switches work best when spanning the center gap.

(Switch to Physical Layout)

So here it is all assembled. See how the tactile switches spans the center breadboard gap? And we'll confirm that we are connected to pins 5, 8, and 9 on the Arduino board.

(Switch to the Code)

On page 62, the tutorial recommends that we load the Lesson 5 Digital Inputs sketch. Like before, you go to the file menu, select open, then go to the location where you extracted the Elegoo zip files. Under your language…in my case English, code, Lesson 5 Digital Inputs, Digital_Inputs, and Select Digital_inputs.ino and click the Open button.

The code consists of a few global variables, assigning pins 5, 9 and 8. In the void setup we setup the pinModes for each pin, and here we use a special parameter INPUT_PULLUP. This parameter turns on the internal Arduino pull-up resistors for those pins. In effect it makes them HIGH when not externally driven by something. (Show illustration) You see internally, the Arduino can enable the pull-up resistors which are connect to the plus 5 volt supply. While the buttons are open, the voltage potential at the pins become 5 volts, or HIGH. Some folks might also use a digitalWrite(8, High) in void setup which will also turn on the internal pull-up resistors. But the parameter INPUT_PULLUP saves a line of code.

The void loop consists of two if statements to check the button state. If you press one of the buttons, you change the state or drive it LOW. This operates the LED. So let's upload the sketch and check it out.

(Switch to the Physical Layout)

Here we have the circuit again, and this time if we press the left button it turns on the LED. If we press it again, nothing happens. If we press the right button, the LED turns off. If we press it again, nothing happens. Works pretty well. So how we improve this?

Well, what if you only wanted to use a single button to turn off and on the LED? We could cleverly keep track of the current and previous states as a variable and switch the LED accordingly. However, this will probably result in miss operating, turning on or off when you didn't intend to do so. Why? Well, do you recall that I mentioned that switches are mechanical? It's those internal springy contacts that sometimes bounce, which to the circuit looks like pressing the button several times before it settles to a HIGH state. Obviously, this could cause issues, making your program work incorrectly. So how do we fix this?

(Switch to illustration)

With discrete components you would add a capacitor and a resistor to the circuit to slowly discharge or charge the capacitor after pressing the button. This smooths out the bouncing effects seen by the circuit. But with the Arduino, we can program a simple function to detect the bounce and simply delay accepting the change in value until things it settles out.

Here's the schematic I came up with. I've removed one of the tactile switches and move the other to digital pin 2. I'll go into detail on why I moved that later. The LED and resistor remain the same.

Here's the breadboard wiring diagram. Other than the removal of one of the tactile switches, you can see I've just swung the other switch to pin 2.

So looking at the code, you can see that I moved the button pin to number 2. I even renamed it for simplicity. I've added several new Boolean values to keep track of the button and LED state. A Boolean is a data type that is either ON or OFF, true or false, LOW or HIGH. And I included a counter to track button pressed just for fun.

The void setup is like before with the addition of the Serial.begin communication function and a call to new serialOut function.

The main void loop checks the state of the button using a new debounce function. Followed by single if statement that checks lastButton and currentButton states. What I'm looking for is the rising edge of the button signal. Specifically, when the voltage goes from LOW to HIGH.

(switch to the button curve illustration)

If you think about it, if the lastButton is HIGH and the currentButton is LOW, we've found the falling edge of the button press. But once the lastButton is LOW and the currentButton is HIGH we've found the rising edge. The 5-millisecond delay allows the program to skip the bounces until the signal settles on either HIGH or LOW.

If it meets these criteria, we increment the counter, switch the LED using a cool shorthand method of assigning the variable to it's opposite value. In other words, the variable ledOn is set to equal not ledOn's value. Being that ledOn is a Boolean, it is either true or false. I then perform a digitalWrite to the LED pin and calls the serialOut function to send a message to the Serial Monitor.

Lastly, I set the lastButton Boolean variable to equal the currentButton value. Then it loops.

Below I've added some new functions. First there's the debounce function that returns a Boolean value. This handy little function performs the digitalRead command to the button pin. If the last value is different than the current, it rechecks it 5 milliseconds later to hopefully ride out the bouncing that occurs with most mechanical switches. It then returns the current value.

Now the serialOut function simply sends out Serial.print statements to the Serial Monitor. This is for diagnostic purposes only. We pass the ledON value to the local variable ledState, and pass the counter value to the local variable i. The Serial Monitor can now tell us if the LED is on and how many time the button has been pressed.

So, let's upload this sketch and check it out.

(Switch to physical circuit)

Here's the circuit, and the LED is OFF. Notice that now I can press the button once and it turns ON the LED. If I press it again, it turns OFF the LED. I can press it several times and it just works. Excellent!

(Switch to Serial Monitor)

I recorded the Serial Monitor as I was demonstrating the circuit. You can see how the Serial.print statements display the status of the LED and the button press count. Cool.

That was fun…. but, what if you wanted your circuit to turn ON initially, run your code for checking… say the temperature, and then go into a deep sleep and automatically wake up to repeat the check. In addition, what if you wanted to be able to press a button to wake the Arduino then have it perform its check and then go back to sleep.

Here's the schematic I came up with. Oh, that's right, I didn't change a thing. I mentioned before that I switched the button to pin 2 and that's because the Arduino UNO only has an interrupt on two pins, 2 and 3. So by using pin 2, we can enable an interrupt to wake the Arduino when it's in sleep mode.

(Switch to the code)

OK, looking at the code, I removed the lastbutton and currentbutton variables since I'll only be using the button to wake to the circuit.

Big changes take place in the void setup area.   Ok we start the Serial Monitor with a Serial.begin statement. Followed by the pinMode to set up the button pin, again with INPUT_PULLUP parameter to enable the pull up resistors. So far pretty standard. But here's where is gets interesting.

The manufacturer's data sheet suggest that you can save power if you set every unused pin to HIGH or LOW. So, we start by setting all the remaining pins to LOW using a simple for loop. The LED pin doesn't need to be added because the for loop takes care of it.

Next, we enable the interrupt pin using the attachInterrupt command. Arduino recommends that we also use the digitalPinToInterrupt function. This allows use to simply pass the buttonpin and let the compiler figure out what interrupt number to use. The second parameter is the function it will call after it wakes up. The third parameter is at what point of the button press do you want to trigger at. Here I chose RISING for the rising edge as you release the button.

Ok, now it gets a little more technical. Kevin Darrah does a great job of explaining how to enable the watchdog timer, disable the analog to digital converter, and enable power down deep sleep. He does this all without including any additional libraries. I encourage you to check out his video.   The links will be provided below.

Setting the watchdog timer is the first part of this code that involves setting a specific Arduino registers. I've included some additional comments to help the reader along in the code. But if you follow along carefully, you should be able to use or include this code in your project. Ok. We start by setting specific bits, in a specific order, to enable the watchdog timer.   First, we set the register WDTCSR to equal 24. This has the effect of setting the two bits, WDCE and WDE, to 1 while clearing the other bits. What happens is that the number 24 is 00011000 in binary. Note that the two one bits are the WDCE and WDE locations we need to set while clearing the others. Next, we set WDTCSR to equal 33. 33 is 00100001 in binary. This sets the prescalers bits, and clears WDE and WDCE bits. For the last part, we enable WDIE bit using a compound bitwise OR operator shift register trick. If you like for information about this trick. Kevin has an earlier video that goes over this technique. The result is that we've set the watchdog timer for every 8 seconds. Again, without including a library. The Arduino's data sheet has more setting options and details, and I encourage you to check it out.

If you're still with me, the next line disables the analog to digital converter (or ADC). A lot of power can be saved if this is disabled. This time we set the ADEN bit to zero inside the Analog to Digital Converter Control Status Register A using a compound bitwise AND operator and an inverted shift register trick where we move the register to bit 7 and invert it. Kevin also goes over this trick in a previous video. Oh, and you can re-enable the ADC if you need it when the circuit wakes up and then disable it before it goes back to sleep. I'll talk about that more later.

Finally, we enable the deep sleep mode by modifying the Sleep Mode Control Register. I've included some details in the code. First, we enable the power down sleep mode and then enable sleep mode.

So now that the setup is complete, we'll move on to the void loop. Here I just send a serial print line to the serial monitor, perform a digitalWrite to turn on the LED, and another serial print to show the LED ON. Here's where you would place whatever code you wanted to run while awake, I kinda show that with a small delay. We then turn off the LED, send another monitor message and call a function goToSleep.

The code jumps out of the void loop and down to a new function void goToSleep. For diagnostic purposes, I've added a serial print here to indicate we're "going to sleep". But in the spirit of saving power, we'll first disable the brownout detection circuit while sleeping. This needs to be done right before we make the Arduino go to sleep. And we do this by setting the BODS and BODSE bits at the same time. The number 3 is 11 in binary and then we us the shift register trick to set them both. Next, we need to clear BODSE and set again BODS bits with four clock cycles. And Kevin came up with this clever piece of code. The actual line of code that make the Arduino go to sleep is this line of code. It's an in-line assembler code, and Kevin has a video on that too.

You may have noticed that I have a small delay after the first serial print. That's because serial print function is asynchronous. Which means it will go to the next line of code before finishing the serial monitor output. Which means it will go to sleep before the serial print finishes. When the circuit wakes, it will continue to the next line after the sleep code. So, if you want to disable and enable analog to digital converter, would move that code before and after the disable brownout and sleep code.

Now once the it wakes, the program will return to what called it, in this case, at the end of the void loop. Here I have another serial print with "void loop ends" for diagnostic purposes. Which of course, all these serial prints, and the serial begin statements, can be commented out or remove in your final code. But it's fun for now.

Ok, below the void loop I have a function called buttonInterrupt. If you recall when we enabled attachInterrupt above, we included this function name. When the button is pressed, it will jump to this function, where I just simply increment a counter. The variable qualifier volatile keeps the counter value in memory while sleeping. And then it jumps to the previous function or subroutine that the program as was last at. Typically, this will be the goToSleep function. Which will return back to the void loop.

The serialOut works like before, but there is one more function I need to talk about and that's the ISR function. The function name must be as you see it here with WDT_vect as the passed parameter. Whenever the watchdog is activated, it will go here first and the jump back to whatever line of code the sketch was at before watchdog was activated. In this case, it happens every 8 seconds.

OK, I know that was very technical. But let's run the sketch and check it out. Oh, and I'll turn on the serial monitor while I play with the circuit to show what's displayed on the Serial Monitor.

(switch to physical circuit)

Ok you can see the circuit and about every 8 seconds the LED light up, then turns off. If we press the button, it wakes up the Arduino and the LED lights up, then turns off and goes back to sleep. We'll do this a couple more time and then check the serial monitor.

Sorry, my video skills are somewhat lacking. And I wasn't able to show you both the circuit board and serial monitor at the same time. However, with a little video editing I can show you what had happened. You can see the first line is from setup. It starts the void loop, show that the LED is on with zero button presses. Then it shows that the LED is OFF with zero button presses. Next it displays "going to sleep", which is the start of the goToSleep function. (Now I assume it goes to sleep.) Then the Watchdog timer wakes up the board and it finishes the goToSleep function and returns to the void loop and display the last line in the void loop which is "Void loop ends". It then repeats the void loop.

Now here I’ve pressed the button and it jumps back to the end of the goToSleep function and back to the void loop.

Notice now the button pressed has increased. So, each time I press the button to wake the circuit, this same thing happens and the button pressed count increases.

Well that’s it for this lesson.

Join me next time for Lesson number 6 – Active Buzzer.

If you like this video don’t forget to rate and subscribe. I’ll try to put out a new video each week.

Thanks and see ya next time.