

Reading digital data on the Arduino platform

Learn how to read external data in an Arduino project



John Wargo

@johnwargo

John is a professional software developer, writer, presenter, father, husband, and geek. He is currently a Program Manager at Microsoft, working on Visual Studio Mobile Center. You can find him at johnwargo.com

```
SerialCallResponse | Arduino 1.8.5
File Edit Sketch Tools Help
SerialCallResponse $
// BTNPIN defines the Arduino input pin to which the
// button is connected
const int BTNPIN = 2;

// Specifies the amount of time the button must stay pushed for it
// to trigger the LED on or off. Increase this value if your LED
// flickers
const unsigned long DEBOUNCE_DELTA = 50; // milliseconds

// btnState stores the current button state (HIGH or LOW)
// initialize it to LOW so the LED stays off until the sketch
// reads a HIGH state for the button input
int btnState = LOW;
// A place to store the previous loop's button state
int prevBtnState = LOW;

Done compiling.
Sketch uses 222165 bytes (21%) of program storage space. Maximum is 1044464
Global variables use 31572 bytes (38%) of dynamic memory, leaving 50348 bytes
59 NodeMCU 1.0 (ESP-12E Module), 80 MHz, 115200, 4M (3M SPIFFS) on COM3
```

Left The Arduino development environment includes code highlighting, to help you spot typos in your code

In the previous article in this series, we showed you how to blink the built-in LED on an Arduino device. Here, we'll show you how to use a push button to toggle the LED on and off. This article illustrates one way to read digital data using an Arduino.

Arduino boards offer several ways to interact with external hardware components, in all cases this means sending a signal to, or reading data from, an external device. Those inputs and outputs, coupled with the logic you've coded in your project's sketch, are the meat of any Arduino project. Arduino inputs come in two formats: analogue and digital, in this article, we'll cover one way to use digital inputs.

Each digital input on the Arduino can read two values: LOW and HIGH. LOW is a constant defined within the Arduino IDE that essentially means zero (or very little) voltage. A value of HIGH references

the highest voltage value the Arduino can support (typically 3V (volts) on an Arduino operating at 3V, and 5V on an Arduino operating at 5V).

Note: Any Arduino device you use for your projects will have one or more digital inputs; these usually double as digital outputs as well. You learned how to use a digital output in the series' previous article.

You might be saying to yourself: "How useful is a digital input if it's only either on or off? That's only one bit, right?" On the Arduino, digital inputs are used in two different ways: to read point-in-time input values, such as the status of a button, or to read a stream of binary digits (bits) values which an application converts into more useful data such as bytes, or numbers. In this article, you'll find out how to use a digital input to read the status of a push button.

YOU'LL NEED

- ◆ **An Arduino or Arduino-compatible device**
We recommend the Arduino Uno for first-time users
- ◆ **Momentary push button**
- ◆ **A 10 kΩ resistor**
- ◆ **Breadboard**
- ◆ **Breadboard jumper wires**

DIGITAL INPUTS CAN READ SINGLE VALUES OR STREAMS OF DATA

In the previous article in this series, we showed you how to use the default Arduino Blink sketch to turn an Arduino's on-board LED on and off on a specific interval. In this project, we'll extend that project, and use a button to turn the LED toggle the status of the LED. When the push button is depressed (pushed), the LED turns on. When the push button is up (open), the LED turns off.

Before we wire up the circuit, let's take a look at the code (you can find the complete code for the example at hsmag.cc/KTioNX).

The sketch defines the **BTNPIN** constant used to identify the Arduino digital input pin to which the button is connected. Following a common convention, we created the constant name in all capital letters, making it easy to distinguish constants from variables in a sketch. You'll populate this constant value with the pin number for your particular hardware implementation.

Next, the sketch defines the **btnState** variable, which is used to store the current state of the button; this value is used to determine whether to turn the LED on or off. Notice how we initialised the variable to **LOW**; this isn't required, but gives the sketch a fallback in case it can't read the button, setting the LED to off by default the first time through the loop.

```
// BTNPIN defines the Arduino input pin to which the
// button is connected
const int BTNPIN = 2;

// btnState stores the current button state (HIGH
or LOW)
// initialize it to LOW so the LED stays off until
the sketch
// reads a HIGH state for the button input
int btnState = LOW;
```

In the sketch's **setup** function, the code sets the mode for the Arduino I/O (input/output) pins used by the sketch. The sketch calls **pinMode** to set the default LED pin (defined in the Arduino IDE's constant **LED_BUILTIN**) to output mode, then calling **pinMode** again to set the push button pin to input mode. Finally, the function turns the LED off, through a call to **digitalWrite**, just to make sure we start with the LED in a known state before the first loop begins.

```
// The setup function runs once every time the
Arduino
// powers up or resets (after a sketch update, for
example)
```

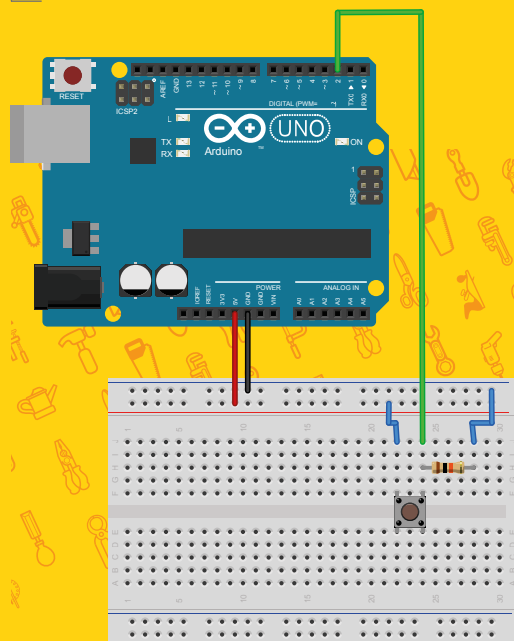
STAYING CONSTANT

A good practice for developers is to use constants to store values used in multiple places in a sketch. The **BTNPIN** constant is a good example for this; by pulling the value into a constant defined at the beginning of the sketch, you make it easy to change this value if the hardware configuration changes (if you connect the button to another digital input pin, for example). You could skip this step, but if you later changed the input pin for your project, you'd have to locate every place in the sketch where it's used, then change each instance. For this small sketch it's not that big of an issue, but for larger sketches it's much easier to do it this way and make one change that affects the whole sketch instead of many little edits, and potentially missing one.

```
void setup() {
// initialize digital pin LED_BUILTIN as an
output.
pinMode(LED_BUILTIN, OUTPUT);
// initialize the push button pin as an input:
pinMode(BTNPIN, INPUT);
// set the initial state of the LED (off)
digitalWrite(LED_BUILTIN, btnState);
}
```

In the sketch's **loop** function, the code reads the button status through a call to **digitalRead** and stores the result in the **btnState** variable. Next, the code →

Figure 1 The Fritzing tool (fritzing.org) can be a great way of designing your circuits before starting on your breadboard



QUICK TIP


The resistor is used in this circuit to help force consistency of digital input values. Without the shunted circuit to ground, there's no clear definition of LOW vs. HIGH, and the input could 'float' at an indeterminate value without an input value applied. With the resistor in place, there's a clear definition of LOW when the button is open through the connection to ground. With the button pushed, the 'slower' path (through the resistor) is ignored because it's a more expensive route than the direct route to the digital input.

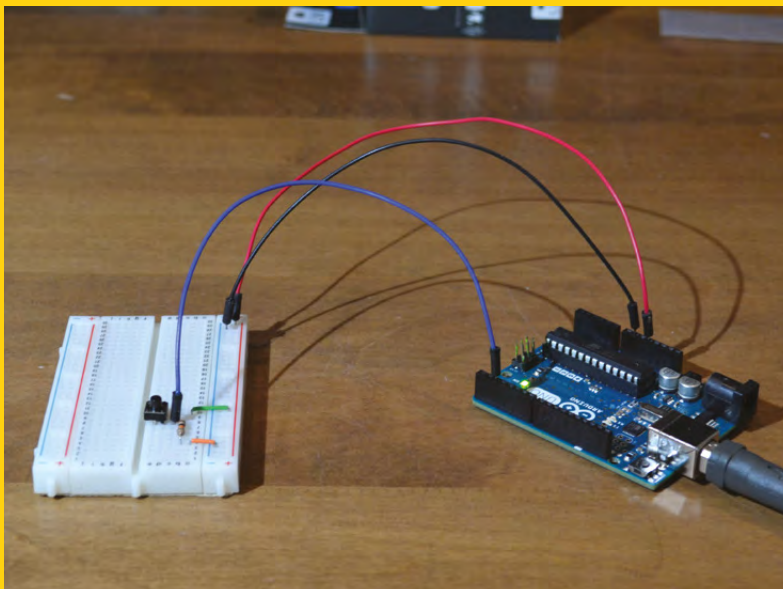
BOUNCING ALONG

Bouncing and debouncing are terms used when describing interactions with electrical connections like the one we have in the push button used in this project. As a button or switch begins a connection or disconnection, there's an uncertainty in the connection as the contacts move. A button potentially makes multiple intermittent connections until the button contacts connect solidly; this is called bouncing. To mitigate bouncing, Arduino developers implement debouncing, a mechanism used to force a single signal from the button through some extra code. In this example, the code debounces the button connection by forcing the application to wait a minimum amount of time with a connection before considering it to be accurate.

uses the value in `btnState` to set the LED status using a call to `digitalWrite`. When `btnState` is `LOW`, the code turns the LED off; when `HIGH`, it turns it on.

```
// The loop function runs repeatedly as long as a sketch is
// loaded and the Arduino has power.
void loop() {
  // Read the state of the button; it's a digital input,
  // so possible returned values are HIGH or LOW.
  btnState = digitalRead(BTNPIN);
  // Use the measured value to set the LED state
  digitalWrite(LED_BUILTIN, btnState);
  // This whole function can be simplified to the following
  // single line of code:
  // digitalWrite(LED_BUILTIN, digitalRead(BTNPIN));
}
```

Below  The complete circuit assembled and running with an Arduino Uno



The code, as shown, breaks that action into two steps: reading the value from the input pin into a variable, then using that variable value to set the output on the default LED pin. That's a great way to do it when you're illustrating how to do something, but you'll use less memory and get better performance in your sketch if you consolidate the two steps into one, as shown in the commented line in the code (shown here uncommented):

```
digitalWrite(LED_BUILTIN, digitalRead(BTNPIN));
```

Here, the result from the call to `digitalRead` is passed as an input to `digitalWrite`. You won't get tremendous performance benefit doing this here but, for larger sketches, especially when you're bumping up against memory limits on the Arduino device, it's a useful approach.

PUSH TO START

Push buttons are mechanical devices, and as you're pushing or releasing the button, there's no guarantee that the Arduino can get a solid reading every time the button is pushed or released. To accommodate this, you can adjust your sketch so it debounces the button connection, ensuring that the button has been pressed for a minimum number of time before triggering a change in LED status.

In the following example, we've enhanced the previous example to include debouncing; you can find the complete code for the following example at hsmag.cc/pEzXyu.

At the beginning of the code, the sketch defines the same `BTNPIN` constant and `btnState` variable used in the previous example. We've also added the `prevBtnState` variable to keep track of the previous state of the button, and the `ledState` to track the current state of the LED. The `lastToggle` variable keeps track of the time the button state changed. Finally, the `DEBOUNCE_DELTA` constant defines the number of milliseconds the sketch waits before it believes in a reading from the button. You'll see all of these in action later in the sketch.

```
// BTNPIN defines the Arduino input pin to which the
// button is connected
const int BTNPIN = 2;
// btnState stores the current button state (HIGH or LOW)
// initialize it to LOW so the LED stays off until the sketch
// reads a HIGH state for the button input
```



```
int btnState = LOW;
// A place to store the previous loop's button
state
int prevBtnState = LOW;
// Used to track the current state of the LED
int ledState = LOW;
// Stores the last time the status of the button
changed
unsigned long lastToggle = 0;
// Specifies the amount of time the button must
stay pushed for it
// to trigger the LED on or off. Increase this
value if your LED
// flickers
const unsigned long DEBOUNCE_DELTA = 100; //
milliseconds
```

The `setup` function is precisely the same as the previous example.

```
void setup() {
// initialize digital pin LED_BUILTIN as an
output.
pinMode(LED_BUILTIN, OUTPUT);
// initialize the push button pin as an input:
pinMode(BTNPIN, INPUT);
// set the initial state of the LED
digitalWrite(LED_BUILTIN, ledState);
}
```

In the `loop` function, the code reads the button using `digitalRead`, just like the previous example. Next, the code checks to see if the current state of the button is the same as it was the previous time the loop executed. If it isn't, the code stores the current time in the `lastToggle` variable.

AROUND AGAIN

The next time through the loop, if the button state hasn't changed, the sketch checks to see how long it's been since the last toggle (by subtracting the value in `lastToggle` from the current time). If the button state hasn't changed in more than `DEBOUNCE_DELTA` milliseconds (`if ((millis() - lastToggle) > DEBOUNCE_DELTA)`), then the sketch knows it has an accurate button reading, and it toggles the LED.

```
void loop() {
// Read the current state of the button
btnState = digitalRead(BTNPIN);

// Is the button in the same state as the last
time
// we came through the loop? No? Then we need
```

```
to record
// the current time (in milliseconds)
if (btnState != prevBtnState) {
// store the current time in milliseconds
//It doesn't matter what the actual time is,
all we need
// to know is how long did the button stay in
this state
lastToggle = millis();
//Reset our previous state, so this check
skips next time
prevBtnState = btnState;
} else {
// OK, the button states (current and
previous) are the same
// Lets see if they've been the same for
DEBOUNCE_DELTA
// milliseconds
if ((millis() - lastToggle) > DEBOUNCE_DELTA)
{
// the button's been pushed (or not pushed)
for at
// least debounceDelta milliseconds, so its
time to
// toggle the LED if needed
//Is the LED at the same state as the button?
if (ledState != btnState) {
// No? Then toggle it
digitalWrite(LED_BUILTIN, btnState);
//Then reset the LED status
ledState = btnState;
}
}
}
```

To test either of these sketches, wire a button into an Arduino (see **Figure 1**, page 79). On one side of the button, the connection shunts from the 5V connection through the 10kΩ resistor to ground (GND). The other button connection routes to the digital input pin 2. With the button pushed, a connection is made from the 5V source to the digital input, bypassing the resistor and forcing the circuit to **HIGH**. When the button is released, the connection to the digital input pin disappears, and the voltage runs through the resistor to ground, making it **LOW**.

Using the Arduino IDE, upload the code to the Arduino device and try pushing the button to toggle the LED on and off. Play around with the value in the `DEBOUNCE_DELTA` constant to see how it affects the sketch's reaction to the button.

Don't forget, all of the project source code is available at hsmag.cc/dMDWFx. □

QUICK TIP

The Arduino's `millis()` method retrieves the current time in milliseconds since the Arduino started running the current sketch; it doesn't give the sketch an accurate time, but does let the sketch track how long it has been since a previous measurement.

BIG DELTA

`lastToggle` and `DEBOUNCE_DELTA` are both long integers because the sketch uses them to calculate time deltas, and time values are very large integers. Even though `DEBOUNCE_DELTA` is a small number (comparatively), since the sketch will be doing arithmetic using those values, we made them the same type to avoid any conversion issues.