ME 120: Button Input for an Arduino
November 14, 2018

Gerald Recktenwald
gerry@pdx.edu

1 Introduction to Button Input on an Arduino

These notes explain how to use a momentary button as an input on an Arduino. We begin with the simple, though not practically useful, example of using a momentary switch to close a circuit that lights an LED. That example is the then extended until we demonstrate the use of interrupts to monitor button presses without interfering with the execution of a continuously running code.

1.1 Switches versus Buttons

We are all familiar with switches that turn devices on and off. Figure 1 shows two common on/off switches that stay in one position – either on or off – until they are switched again. On-off switches are very important because they allow us to turn power to an entire device. However, once a device is running we may need to interact with it to change it's state. For example, we may want to toggle between operating states, e.g. switch to a new speed setting or change direction of a motor. In other words, simple on-off switches are not the best mode of user input for many control features on modern devices.

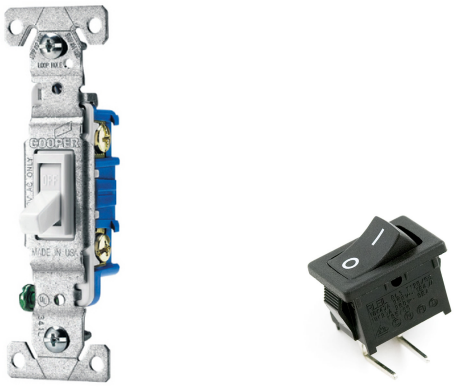

Figure 1: Examples of on/off switches (not to scale). A wall switch (left) for a typical 110 VAC household circuit (`lowes.com`). A panel-mount rocker switch (right) used as turning on an electrical device (`sparkfun.com`).

1.2 Momentary Buttons

Many electronic devices have momentary switches that toggle between on and off states, but do not retain their state. Examples are the "home" and enter buttons on your cell phone, push-buttons on many consumer devices, and keypads for data entry. Momentary switches typically have a tactile and audible "click".

Figure 2 shows a spring-loaded momentary switch. The button shown in Figure 2 has four electrical pins, which is typical for these small buttons. Two pins on each side of the button are permanently connected, so electrically there is only one circuit that is either open or closed. In

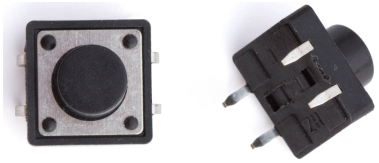

Figure 2: The push-button style momentary switch.

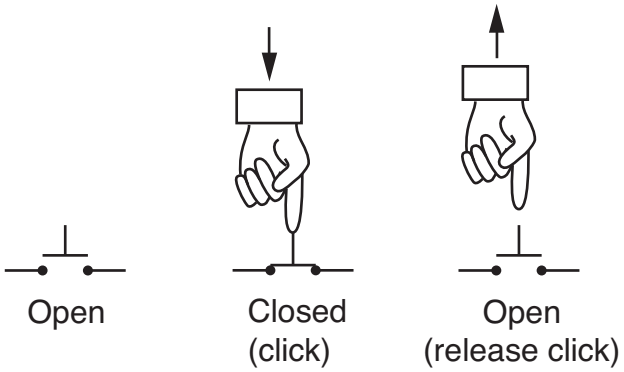

Figure 3: Actuation of a normally open (NO) push button.

other words, only two of the pins are needed for the electrical functioning of the button. Four pins provide better mechanical support when this type of button is soldered to a circuit board.

Figure 3 depicts the sequence of clicking a normally open momentary button. Initially the button is open, meaning no electrical connection is made between the two terminals. Pressing the button closes the circuit between the two terminals and causes a click sound. As long as the button is pressed, the circuit is closed. Releasing the button opens the circuit again, and causes a second click sound.

Simple buttons and switches come in either normally-open (NO) and normally closed (NC) configurations. Figure 4 shows schematic representations of NO and NO switches. A NO switch is an open circuit in its default state. Engaging a NO switch, e.g., by clicking the button, changes it to a closed circuit. A NC switch is a closed circuit in its default state. Engaging a NC switch changes it to an open circuit. The typical momentary push button in an Arduino kit is a NO switch. You can easily verify the type of switch, either NO or NC, by testing for continuity with a multimeter.

1.3 Categories of Button Implementations

Buttons and switches can be categorized by whether they retain their state mechanically. Switches retain their state, and buttons do not. We can also categorize whether the system waits or does not

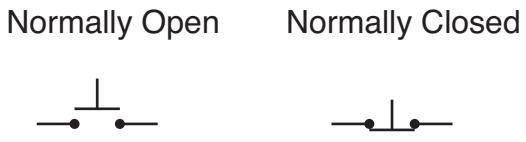

Figure 4: Schematic representations of normally open and normally closed push buttons.

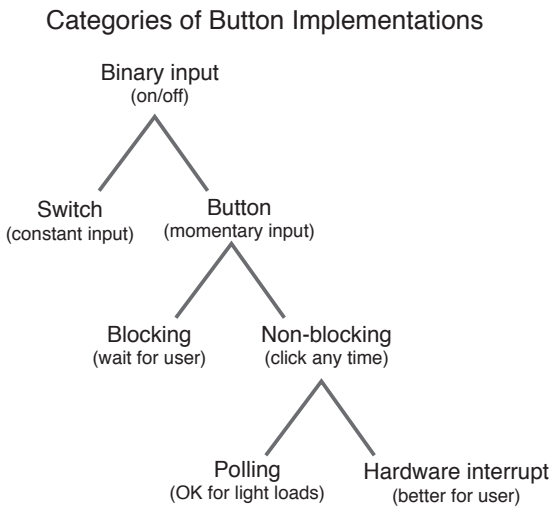

Figure 5: Categorization of button implementations with microcontrollers.

wait for the button or switch input. The waiting for user input is referred to as *blocking input*. Not waiting for user input, which means the system is more responsive, is referred to as *non-blocking* input.

1.3.1 Retaining State

Buttons and switches provide binary input to the microcontroller, i.e., via the button or switch a user indicates whether a feature is either on or off. The on or off condition is a *state* under which the microcontroller is operating. Imagine that the button or switch selects one of two speeds of a motor. The button or switch would allow the user to choose whether the motor operates in the low speed state, or the high speed state[1].

Figure 5 is a hierarchical representation of how buttons (and switches) might be implemented with an Arduino or other microcontroller. At the highest level, switches and buttons are distinguished by whether the input to the microcontroller is constant (with a switch) or momentary (with a momentary button). Because a switch retains its mechanical position (either on or off), a switch retains its state. In other words, a switch provides a constant input to the microcontroller, and that constant state is changed only when the user changes the position of the switch.

Momentary buttons, while providing binary input, do not retain the state associated with the button input. In other words, when a user clicks a momentary button, the signal is received by a circuit and program, and it is up to the circuit or program to remember the state. Once pressed, the button returns to its neutral position. The button has no mechanical memory of its state.

1.3.2 Blocking and Non-blocking Response

When using a momentary button, the response of the device can also be categorized by whether the microcontroller code to detect the button input is either *blocking* or *non-blocking*. Blocking code

[1] Although buttons and switches provide binary input, it is possible with a button (or multiposition switch) to select from multiple discrete states (multiple speeds in the example), but we will defer that discussion until after laying the foundation with binary states.

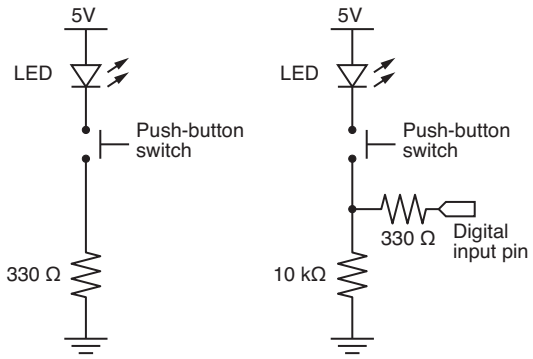

Figure 6: Two circuits that use a momentary input button to control an LED circuit. The circuit on the right uses a pull-down resistor and a digital input to an Arduino to monitor the button state

stops all activity until a button press is registered. Non-blocking code allows the main activity of the code to continue until, and only when, a button press occurs.

One common blocking applications is to delay start of some activity until the user initiates the activity with the a button click. This would be useful if the application required the user to configure addition objects that are not under control of the microcontroller. For example, if a device had to sort cards, the control algorithm may wait until the user fills a hopper a nd then presses a start button. Another related application would be to synchronize some activity that requires user intervention. For example, a machine could perform some automated task, say drilling a series of holes, and then wait until some other task, controlled by another automated device had been completed. Once again, the user input via button press would indicate that the next (dependent) task can begin.

For most non-blocking applications, the button input can happen at any time. Those applications require the microcontroller program to continue its normal task without waiting for a button press. In other words, the button input cannot block normal operation of the code. For this application, the best implementation strategy is to use hardware interrupts – a feature of most microcontrollers. Novice users may attempt to write their own code to watch for a button press event, which is *not* a good idea. We will describe how to use interrupts in a later section.

1.4 Overview of These Notes

In the next section, we demonstrate a simple way of using a button to provide a binary input signal to a microcontroller. This simple applications is used to introduce some basic ideas, including pull-up and pull-down resistors. Building on the simple example, we introduce complexities to produce more desirable system response. Along the way we provide demonstration codes.

An important practical consideration of using buttons is debouncing, which refers to either hardware or software solutions to the high frequency, mechanical bouncing that occurs when a momentary button is pressed.

The ultimate goal of these notes is to show how to use interrupts to manage button inputs, while allowing the main microcontroller code to execute continuously.

2 Button Input to Complete an LED Circuit

As a warm-up exercise, consider the use of a momentary input switch to complete the LED circuits in Figure 6. Both circuits in Figure 6 use a push-button type switch to control whether electrical current flows through an LED. The circuit on the left has a current-limiting 330 Ω resistor.

The circuit on the right side of Figure 6 also uses a push button to control whether current flows through the LED. In addition, the circuit on the right provides an input signal to one of the digital I/O pins on an Arduino. The LED can be removed from the circuit and the input feature of the circuit will continue to work. The circuit on the right has a 10 k *pull down* resistor that causes the Arduino input to be `LOW` (close to 0V) when the button is not pressed. When the button is pressed, the Arduino input pin is connected to the 5V supply via the 330 Ω resistor, which has the purpose of limiting the current flowing to the input pin.

The `button_status.ino` sketch in Listing 1 is an Arduino program to read and display the status of the button in the circuit on the right hand side of Figure 6. The digital I/O pin is configured with

```
pinMode( buttonPin, INPUT);
```

and the state of the button is read with

```
pinState = digitalRead( buttonPin )
```

The output of `digitalRead` is a logical value, i.e., either `HIGH` or `LOW`, which are equivalent to `TRUE` or FALSE. When the output of `digitalRead` is store in `pinState` the value is either 1 or 0. The state of the button is determined by testing the value of `pinState`. The logical test

```
pinState == HIGH
```

returns `TRUE` if the button is pressed, i.e.,, if the value stored in `pinState` is `HIGH`.

The print commands in the `button_status` sketch are used to expose the status of the button for debugging purposes. In a practical sketch, some other actions, e.g., turning on a motor or initiating a sensor measurement, would be taken depending on the state of the button. Note that

```
//  File:  button_status.ino
//
//  Print the state of a momentary switch

int buttonPin = 11;                //  Specify digital I/O pin connected to the button

void setup() {
  pinMode( buttonPin, INPUT);      //  Configure digitial I/O pin for input
  Serial.begin(9600);
}

void loop() {
  int buttonState;

  buttonState = digitalRead( buttonPin );    //  Read the current button state
  if ( buttonState == HIGH ) {
    Serial.println("Button pin is HIGH");
  }
  else {
    Serial.println("Button pin is LOW");
  }
}
```

Listing 1: Arduino code to monitor and display status of a momentary input button.

Table 1: Physical and logical values corresponding to digital inputs.

Physical or logical condition	True	False
Voltage on digital I/O pin	near 5V	near 0V
`digitalRead` returns	`HIGH`	`LOW`
Return value stored as `int`	`1`	`0`
Return value stored as `boolean`	`true`	`false`
Equivalent `int` values	Anything $\neq$ `0`	`0`

the `button_status` sketch does not use any digital I/O to turn on the LED. For the circuits in Figure 6, the LED turns on whenever the button is pressed regardless of whether an Arduino is connected to the circuit or not.

2.1 Using Logical Variables to Indicate State

The on/off state of a feature in an Arduino program corresponds to the logical values of `HIGH` and `LOW`, or `TRUE` and `FALSE`. Table 1 lists the equivalent forms of `TRUE` and `FALSE`.

In an Arduino program, `int` and `boolean` variables can be used to store the output of the `digitalRead` function. A `boolean` variable occupies only one bit of memory and can only take on the values of 1 or 0 (True or False). A signed integer can take on values between -32768 and $+32767$. Any `int` value that is not zero is considered to be equivalent to True.

In the `button_status` program in Listing 1, the state of the button is read by the `digitalRead` command and stored in `buttonState`. The state of the button is examines with an `if` construct.

```
if ( buttonState == HIGH ) {
  ...
}
```

The `buttonState == HIGH` expression is either True or False, depending on the value stored in `buttonState`. The `== HIGH` test is not needed, however, since the value stored in `buttonState` is equivalent a logical True or False. Thus, the test could be written more simply as

```
if ( buttonState ) {
  ...
}
```

The two codes in Listing 2 show this alternate implementation. The code on the right hand side of Listing 2 shows how to declare `buttonState` as a `boolean` variable type.

```
//  File:  button_status_alt.ino
//
//  Print the state of a momentary switch

int buttonPin = 11;

void setup() {
  pinMode( buttonPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  int buttonState;

  buttonState = digitalRead( buttonPin );
  if ( buttonState ) {
    Serial.println("Button pin is HIGH");
  }
  else {
    Serial.println("Button pin is LOW");
  }
}
```

```
//  File:  button_status_boolean.ino
//
//  Print the state of a momentary switch

int buttonPin = 11;

void setup() {
  pinMode( buttonPin, INPUT);
  Serial.begin(9600);
}

void loop() {
  boolean buttonState;

  buttonState = digitalRead( buttonPin );
  if ( buttonState ) {
    Serial.println("Button pin is HIGH");
  }
  else {
    Serial.println("Button pin is LOW");
  }
}
```

Listing 2: Alternative implementations of Arduino code to monitor and display status of a momentary input button. The only difference between these codes is in the definition of `buttonState`. In the code on the left, `buttonState` is declared as an `int`. In the code on the right, `buttonState` is declared as a `boolean`. Both implementations are correct.

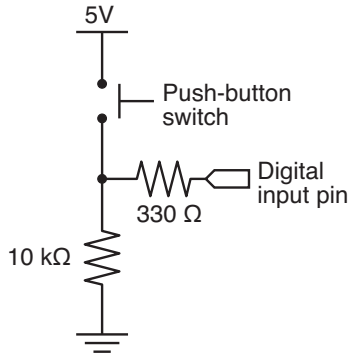

Figure 7: Pull-down configuration of a momentary input button.

3 Wait for Input

The `button_wait_to_start` sketch in Listing 3 demonstrates how a button input circuit could be used as a sort of "start" button. The sketch waits in the `setup` function until the user presses the button. The code block that implements the button input is

```
while ( !buttonStatus ) {
  buttonStatus = digitalRead( buttonPin );
}
```

The `buttonStatus` variable is initially set to `false`. Therefore, on the first test of the `while` condition gives `!buttonStaus = !false = true`. Therefore, the first execution of the `while` clause causes the body of the `while` loop

```
buttonStatus = digitalRead( buttonPin );
```

to be executed. Suppose that the user has not pressed the button. The value retuned by `digitalRead(buttonPin)` will be `LOW` or `false`. Therefore, when the `while` clause is evaluated the next time, `LOW` is stored in `buttonStatus` which means that

```
!buttonStatus = !LOW = HIGH = true
```

Sincd `!buttonStatus` returns `true`, the body of the loop is executed again. As long as the button is not pressed, the `while` loop keeps being repeated. When the button is pressed, a value of `HIGH` is stored in `buttonStatus` which gives

```
!buttonStatus = !HIGH = LOW = false.
```

Since the logical clause in the `while` expression is `false`, execution continues after the body of the loop. In summary, the code block

```
while ( !buttonStatus ) {
  buttonStatus = digitalRead( buttonPin );
}
```

causes the program to pause until the button is pressed.

When a button is used as a "wait-to-start" user input, the Arduino can perform no useful work while waiting. It is possible to put some useful calculations inside the body of the loop, e.g., one could attempt to read a sensor or change a motor speed. However, any substantial work will take time, and it may cause the Arduino code to miss a quick button click. The short `while` loop in the `button_wait_to_start` code works because the Arduino is doing *nothing but waiting* and monitoring the status of the digital I/O pin.

```
//  button_wait_to_start.ino   Demonstrate button input to an Arduino
//
//  Execution is suspended in the start() function until the
//  the button connected to buttonPin (11) is pressed.  The button must
//  be configured with a pull-down resistor on pin 11 so that
//  the pin state is LOW until and unless the button is pressed.
//
//  The loop() function executes the standard blinking LED
//  algorithm.  An LED circuit (LED and current-limiting resistor)
//  must be connected to LEDpin (7).

int buttonPin = 11;     //  pin for input from the button
int LEDpin = 7;

void setup() {
  int buttonState = false;

  pinMode( buttonPin,  INPUT);
  pinMode( LEDpin, OUTPUT );

  while ( !buttonState ) {
    buttonState = digitalRead( buttonPin );
  }
}

void loop() {
  digitalWrite( LEDpin, HIGH );
  delay(1000);
  digitalWrite( LEDpin, LOW );
  delay(500);
}
```

Listing 3: Arduino code that uses a (momentary) push button as wait-to-start switch. The code in the `loop` is the standard code to blink an LED.

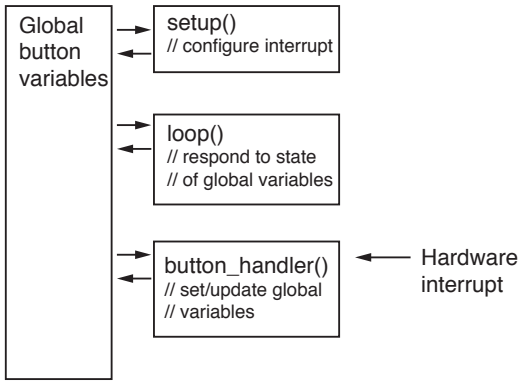

Figure 8: Conceptual model of an Arduino sketch using an interrupt to manage button input.

4 Using Interrupts

A hardware interrupt is a specialized circuit that waits for an electrical signal on a user-designated digital input channel. When the signal occurs, the hardware interrupt circuit stops the normal flow of the program, and gives control to an *interrupt handler*, which is a user-written function that designed to respond to the event that caused the interrupt. The interrupt handler is a function that quickly adjusts some internal flags and/or counters and then quits.

Although it is a somewhat counter-intuitive idea, the interrupt handler should not do the task implied by the interrupt. Instead the interrupt handler sets some global state variables that then influence the flow of execution of the normal code. We will show how this works with an example, but first we will provide an overview of the components necessary to implement button input with an interrupt.

Interrupts are useful in responding to digital input pulses that happen quickly. Examples applications are momentary buttons (buttons that uses press or "click" briefly) and encoders that send series of pulses to indicate angular position of a shaft or linear position along a rail.

4.1 Interrupts for Button Input

As with the examples from the preceding sections, a momentary button DC circuit is connected to a digital input channel so that the input goes from LOW to HIGH (or vice versa, depending on preference) when the button is pressed.

To use an interrupt with the button, the button output must be connected to one of the digital channels that have hardware interrupt capability. The interrupt hardware detects the change in state for the digital channel, and then sends a software signal to the microcontroller controller code that demands instant reaction. In other words, normal execution of the microcontroller code is interrupted.

Figure 8 is a schematic representation of an Arduino sketch that uses an interrupt. The key features of this model are global variables that indicate the button state, and user written interrupt handler (in this case `button_handler`). Adding an interrupt to an Arduino code typically involves these changes

1. Defining a global variable that indicates the button state: either "it was pressed" or "it has not been pressed"

2. Attaching an interrupt in the `setup` function

Table 2: Interrupt pins on an Arduino UNO.

Interrupt number	Interrupt number
0	2
1	3

Table 3: Interrupt modes expressed in terms of changes in logical value on a pin. The logical values correspond to voltage thresholds with `HIGH` being *near* 5V and `LOW` being *near* 0V.

`LOW`	Triggers when the logical value on the pin becomes `LOW`.
`CHANGE`	Triggers when logical value on the pin changes from `LOW` to `HIGH` or from `HIGH` to `LOW`.
`RISING`	Triggers when logical value on the pin changes from `LOW` to `HIGH`.
`FALLING`	Triggers when logical value on the pin changes from `HIGH` to `LOW`.
`HIGH`	*Only available on Arduino Due.* Triggers when the logical value on the pin becomes `HIGH`.

3. Adding a user-defined function to act as an *interrupt handler*
4. Adding code to the `loop` function that decides what to do when the "it was pressed" variable is TRUE.

The interrupt handler is an key component that should only perform the minimal amount of work, and above all it should execute very quickly.

4.2 Interrupt on an Arduino Uno

The Arduino UNO has two interrupt circuits, on digital pins 2 and 3. Other microcontrollers in the Arduino family have more interrupts. Table 2 summarizes the somewhat confusing relationship between the *interrupt number* and the *interrupt pin*. When configuring an interrupt with the `attachInterrupt` function, you use the interrupt number, not the interrupt pin. When you wire your circuit, you connect the button circuit to the pin corresponding to the interrupt number in your code.

In electronics, a trigger is a circuit and code logic that responds to a change in an electrical signal. For example, a `RISING` trigger causes a response when the voltage rises above a threshold. Another way to think about a `RISING` trigger is that the system is watching for a digital line to make a `LOW` to `HIGH` transition.

Table 3 lists the different ways that an interrupt can be programmed to trigger. These modes are used in the `attachInterupt` function[2].

[2]See `http://arduino.cc/en/Reference/AttachInterrupt` and `https://www.arduino.cc/reference/en/language/functions/interrupts/interrupts/`.

4.3 Example Implementation

The `button_interrupt_count` sketch in Listing 4 shows how an interrupt handler can be used to count how many times a button has been pressed. More sophisticated applications are possible, but those applications share many of the ideas in the button counting example.

`handle_click` is the interrupt handler in Listing 4. When the button is pressed, `handle_click` first checks to see if there has been an interrupt with the last 200 milliseconds. If so, the interrupt is ignored. The decision to ignore two interrupts in quick succession is one way to *de-bounce* the button/interrupt handler. De-bouncing is necessary because the button is a mechanical switch with a spring. During normal operation the button will vibrate, or bounce, after the user initiates the button press.

If the interrupt occurs more than 200 milliseconds after the last interrupt, then the interrupt handler considers the interrupt to be a valid button press and the toggle-switching line is executed

```
toggle_on != toggle_on;
```

Before leaving the interrupt handler, the time of the interrupt is recorded so that the next interrupt can be checked for validity.

For a more general introduction to interrupts refer to the Wikipedia page and the on-line Arduino documentation[3].

[3]See `https://en.wikipedia.org/wiki/Interrupt`, `http://playground.arduino.cc/Code/Interrupts`, and `http://makezine.com/2012/01/25/how-to-arduino-interrupts/`.

```
//  File:  button_interrupt_count.ino
//
//  Use an interrupt to catch button status.  Count and print the number of
//  times the button has been pressed.

int  button_interrupt = 0;   //  Interrupt 0 is on pin 2 !!
int  toggle_on = false;      //  Button click switches state
int  button_count = 0;       //  Number of times the button has been pressed

// -----------------------------------------------------
//   Interrupt and Serial Monitor are configured in setup()

void setup() {
  Serial.begin(9600);
  attachInterrupt( button_interrupt, handle_click, RISING);  // Register interrupt handler
}

// -----------------------------------------------------
//   loop() uses global variables but does not assign values to them:
//      toggle_on indicates the state
//      button_count tells how many times the button has been pressed

void loop() {

  if ( toggle_on ) {
    Serial.print ("on   ");
  } else {
    Serial.print("off  ");
  }

  Serial.println(button_count);
}

// -----------------------------------------------------
//   handle_click() is the interrupt handler.  It responds to button clicks and
//   updates toggle_on and button_count as appropriate.  By keeping track of
//   the time between button clicks, handle_click avoids spurious events due
//   to high speed mechanical bouncing of the button

void  handle_click()
{
  static unsigned long last_interrupt_time = 0;       //  Zero only when code first runs

  unsigned long interrupt_time = millis();             //  Read the clock
  if ( interrupt_time - last_interrupt_time > 200 ) {  //  Only count clicks separated by 200 msec
    toggle_on = !toggle_on;
    button_count++;
  }
  last_interrupt_time = interrupt_time;
}
```

Listing 4: Arduino code that counts the number of presses of a (momentary) push button.