Arduino

Switch Library User Guide

A Library Supporting the Reading of Multiple

Mixed-type Simple Switches & Circuits

|  |
| --- |
| Image result for arduino switches |
| https://th.bing.com/th/id/OPE.ggrzMMUtvfVLuA300C300?w=200&h=150&rs=1&pid=21.1 |
| Image result for arduino toggle switches |
| Image result for arduino rotary switches |
| See the source image |

# Warranties & Exceptions

This document and its content are in the public domain and may be used without restriction and without warranty.

Contents

[Warranties & Exceptions 2](#_Toc65332383)

[Introduction 3](#_Toc65332384)

[Overview 3](#_Toc65332385)

[Design Objectives 3](#_Toc65332386)

[Constraints & Limitations 4](#_Toc65332387)

[Switch Types Supported 4](#_Toc65332388)

[Common Switch Wiring Schemes 5](#_Toc65332389)

[Using the <switch\_lib> Library 5](#_Toc65332390)

[Specifications 5](#_Toc65332391)

[Specifications - Reserved Macro Definitions 5](#_Toc65332392)

[Specifications – Switch Control Structure (SCS) 6](#_Toc65332393)

[Specifications - Switch Control Functions 7](#_Toc65332394)

[Example Sketches 11](#_Toc65332395)

# Introduction

Implementing switches, of any type, can be troublesome as not all switches are equal! Some are ‘fleeting’ or momentary, like button switches, and some are simply either on or off until they are ‘flipped’ at their next actuation. Button switches are fairly standard in their design, but toggle type switches are many varied – simple toggle, slide, tilt, rotary, etc. Issues such as switch design, transition ‘noise’ and wiring schemes all come into play at some point in a project’s design.

The good news is that both types of switch can be brought to heel by the <switch\_lib> library which provides a software solution for connecting a mix of switch types wired in a variety of circuit schemes. The end resut is that, by using the <switch\_lib>, the only components required are switches, connecting wires and, if wished 10k ohm pull down resistors. However, even the 10k ohm resistors can be left out by choice of the right circuit (see below, Common Switch Wiring Schemes).

This guide describes the <switch\_lib> library for Arduino, detailing the functions and definitions available to the end user for implementing switches of either style and in a choice of wiring schemes - any number of switches of any style and of varying common wiring designs may be configured, the only limitation being the number of digital pins available.

# Overview

The essence of this guide

# Design Objectives

At the outset a number of key objectives were established for the design of a switch library, these being a library that supported:

* ease of configuration, irrespective of type of switch or how connected (wired)
* different switch types - the ubiquitous button switch and a variety of different types of toggle switch
* support for common wiring schemes – simply connected with or without a 10k ohm pull down resistor
* mixed switch/circuit implementations – support for a mix of switch types and wiring schemes
* software auto-debounce of noisy switch transitions – removing from end user design considerations issues relating to noisy switch transition by incorporating transparent debounce features
* one switch read function irrespective of switch type or wiring scheme – providing a simple to use function to read all switches
* developing a user guide that is informative and such that it is easy to ‘dip’ into.

# Constraints & Limitations

Nothing in this world is perfect and <switch\_lib> is far from that. Whilst it does provide a set of useful capabilities to aid and assist Arduino projects involving switches there are several constraints and limitations in its design and use to be aware of:

1. Every switch to be configured requires its own digital pin. Whilst this is not an issue for say, a mega 2560 microcontroller, lesser boards are more constraining in the number of digital pins they support. Certainly for UNO microcontrollers and better, there should not be a practical issue in mapping switches to digital pins for most switch hungry projects.
2. Development of the library was limited to six switches –

* two x button switches wired with a 10k ohm pull down resistor
* one x button switches wired without a 10k ohm pull down resistor
* two x toggle switches wired without a 10k ohm pull down resistor
* one x toggle switches wired with a 10k ohm pull down resistor

but, there is no reason to believe that more could not be configured within the limits of the chosen microcontroller.

1. For every switch configured, 10 bytes of free memory will be allocated at run time when the <switch\_lib> class is initiated. This memory requirement is in addition to the size of the compiled sketch.
2. The period of time defined for switch noise debounce is global and applicable to all switches, irrespective of type. It is preset ‘out of the box’ (OOTB) to 10 milliseconds but it may be programmatically adjusted by the end user code as required (see function set\_debounce, below).
3. The library supports two simple and commonly seen switch wiring schemes (see Common Switch Wiring Schemes), these being without the use of any hardware components other than 10k ohm resistors. Even these can be dispensed with! Having said that, the library should also support (but not tested) switches connected with hardware debounce circuits. If this is the case then set the software debounce period to 0 milliseconds (see function set\_debounce, below).
4. For switches to be responsive in something like real-time, they need to be tested frequently and, for button switches particularly, processed when a switch cycle is detected. Non-button style switches may have their current status examined at any point and any time. A software design based on a switch polling loop should be an ideal harness to ensure continuous switch testing and processing.

# Switch Types Supported

There are so, so many switches available, many for specific purposes but most of a general nature and suitable for the majority of needs.

The switch library was developed to support two types of common and general use switches – button, or momentary switches and toggle switches. Of course this latter type of switch, toggle, itself comes in all kinds of designs, for example, simple single lever, pop-on pop-off, rotary, slide, tilt, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| Image result for arduino switches | https://th.bing.com/th/id/OPE.ggrzMMUtvfVLuA300C300?w=200&h=150&rs=1&pid=21.1 | Image result for arduino toggle switchesImage result for arduino rotary switches | See the source image |
| *Example of common types of switch* | | | |

The principal distinction between button (momentary) and toggle type switches is that button switches have a switch cycle of OFF-ON-OFF which signifies switch activation, whereas toggle switches go through either OFF-ON or ON-OFF representing two switch transitions. The status of toggle switches therefore persists after physically switched – they stay ON or OFF. Switch\_lib automatically handles this feature.

# Common Switch Wiring Schemes

If you now appreciate the differences between switch types, it is necessary to understand how they should be connected to the microcontroller.

The two commonly seen switch wiring schemes *without* hardware debounce are shown below, as ‘circuit\_C1’ and ‘circuit\_C2’:

|  |  |
| --- | --- |
|  |  |

Either circuit can be used for either type of switch, but the key to configuring correctly lies in they way they are software configured via the pinMode function, as follows.

The pinMode setting for initialising circuit\_C1 is pinMode(<pin>, INPUT). This has the effect of setting the digital pin <pin> to 0v, representing ‘off’. The 10k ohm resister ensures that the pin stays at 0v until switched. Otherwise, the input pin will be susceptible to spurious firing from extraneous fields. When the switch is actuated the input rises to +5v which will be detected as the switch transitioning to ‘on’.

For circuit\_C2 the pinMode setting is pinMode(<pin>, INPUT\_PULLUP). This brings intp play an internal microcontroller pull up resistor resulting in the digital pin floating at 5v, representing ‘off’. No external resistor is required and when the switch is actuated the pin will be brought to 0v which will be detected as the switch transitioning to ‘on’.

Note the reversed conditions for ‘off’ and ‘on’ between the two circuit schemes. The <switch\_lib> will account for this automatically.

When defining switches using <switch\_lib>, t this is achieved very easily, see below.

# Using the <switch\_lib> Library

# Specifications

## Specifications - Reserved Macro Definitions

| Macro definitions | Values | Significance |
| --- | --- | --- |
| #define button\_switch | 1 | differentiates switch type, this being of type ‘button’ |
| #define toggle\_switch | 2 | differentiates switch type, this being of type ‘toggle’ |
| #define circuit\_C1 | INPUT | switch circuit configured with an external pull down 10k ohm resistor |
| #define circuit\_C2 | INPUT\_PULLUP | switch circuit configured without an external pull down resistor |
| #define switched | true | signifies switch has been pressed and switch cycle complete, otherwise !switched |
| #define on | true | used for toggle switch status. Off is !on |
| #define not\_used | true | ‘not used’ indicator – marks if a field in the switch control structure is used or not |
| #define bad\_params | -2 | invalid add\_switch parameters |
| #define add\_failure | -1 | add\_switch could not insert a given switch, ie no space left |
| #define configured | true | Indicates a switch control entry is used, otherwise !configured |

## Specifications – Switch Control Structure (SCS)

At the heart of the <switch\_lib> library is a struct(ure) ‘table’ - the switch control structure (SCS), that is used to hold the data attributes for all defined switches.

At initiation of the class, the SCS is created from free memory using a malloc call. Thereafter, it may be populated by use of the add\_switch function (see below) up to the maximum number of switches required/declared.

The SCS has the following construction and layout:

|  |
| --- |
| struct switch\_control {  byte switch\_type; // type of switch connected  byte switch\_pin; // digital input pin assigned to the switch  byte switch\_circuit\_type; // the type of circuit wired to the switch  bool switch\_on\_value; // used for BUTTON SWITCHES only –  // defines what "on" means  bool switch\_pending; // records if switch in transition or not  long unsigned int switch\_elapse\_timer;// records debounce timer count  // when associated switch is in transition  bool switch\_status; // used for TOGGLE SWITCHES only – current  // state of toggle switch.  } |

Elements of the structure may be directly accessed from the end user sketch, as required.

## Specifications - Switch Control Functions

| Type | int | Name | add\_switch |
| --- | --- | --- | --- |
| Parameters | byte sw\_type, byte sw\_pin, byte circ\_type  parameter choices are:  sw\_type - is either ‘button\_switch’ or ‘toggle\_switch’,  sw\_pin - is the digital pin assigned to the switch,  circ\_type - is either ‘circuit\_C1’ or ‘circuit\_C2’. | | |
| Purpose / functionality | This function will add (create) the specified switch (parameters) to the switch control structure, if possible.  There are three possible outcomes from an add\_switch call:   1. Successful addition of switch. In this case the return value is >= 0 and represents the physical slot (location ‘switch\_id/token’) of the switch in the switch control structure. This may be retained by the calling code/design if specific switches are to be tested. 2. No further slots available in the switch control structure, all are used. 3. The supplied parameters are ‘bad’.   The results of an add\_switch call are as below. | | |
| Return values | Return values are:  >= 0 the switch control structure entry number (switch\_id/token)  for the switch added,  -1 add\_failure - no slots available in the switch  control structure,  -2 bad\_params - given parameter(s) for switch are not valid. | | |
| Example | | | |
| void create\_my\_switches() {  for (int sw = 0; sw < num\_switches; sw++) {  int switch\_id = my\_switches.add\_switch(my\_switch\_data[sw][0], // switch type  my\_switch\_data[sw][1], // digital pin number  my\_switch\_data[sw][2]);// circuit type  if (switch\_id < 0)  { // There is a data compatibilty mismatch (-2),  // or no room left to add switch (-1).  **Serial**.print("Failure to add a switch:\nswitch entry:");  **Serial**.print(switch\_id);  **Serial**.print(", data line = ");  **Serial**.print(my\_switch\_data[sw][0]);  **Serial**.print(", ");  **Serial**.print(my\_switch\_data[sw][1]);  **Serial**.print(", ");  **Serial**.println(my\_switch\_data[sw][2]);  **Serial**.println("!! PROGRAMME TERMINATED !!");  **Serial**.flush();  exit(1);  } else {  // 'switch\_id' is the switch control slot entry for this switch,  // so we can use this, if required, to know where our switches are  // in the control structure by keeping a note of them.  }  }  } // End create\_my\_switches | | | |

| Type | int | Name | num\_free\_switch\_slots |
| --- | --- | --- | --- |
| Parameters | none | | |
| Purpose / functionality | Returns the number of free slots available in the switch control structure. | | |
| Return values | 0 to the maximum number of switches defined | | |
| Example | | | |
| **Serial**.print("\nNumber of free switch slots = ");  **Serial**.println(my\_switches.num\_free\_switch\_slots()); | | | |

| Type | bool | Name | read\_switch |
| --- | --- | --- | --- |
| Parameters | byte switch\_id | | |
| Purpose / functionality | Function will read the given switch returning a result as below.  Note that the sw parameter is the switch entry number (switch\_id) in the switch control structure of the switch to be read. This is the returned value from the add\_switch function call.  If an invalid switch\_id is given the read function exits with a return value of !switched.  See add\_switch() for further information. | | |
| Return values | switched or !switched | | |
| Example | | | |
| ...  // Poll all switches - examine each switch control entry in turn, so switch\_id  // will run from 0 to the number of switches ser up less 1.  for (int switch\_id = 0; switch\_id < num\_switches; switch\_id ++) {  if (my\_switches.read\_switch(switch\_id) == switched) {  // This switch (switch\_id) has been pressed, so process via a switch case  if (my\_switches.switches[switch\_id].switch\_type == button\_switch) {  **Serial**.print("\nbutton switch on digital pin ");  } else {  **Serial**.print("\ntoggle switch on digital pin ");  }  byte my\_switch\_pin = my\_switches.switches[switch\_id].switch\_pin;  **Serial**.print(my\_switch\_pin);  **Serial**.println(" triggered");  // Move to switch's associated code section  switch (my\_switch\_pin) {  case my\_button\_switch\_1:  **Serial**.println("case statement 1 entered");  break;  case my\_button\_switch\_2:  **Serial**.**println**("case statement 2 entered");  break;  ...  ...  case my\_toggle\_switch\_2:  **Serial**.print("case statement 5 entered, switch is ");  **Serial**.println(my\_switches.switches[switch\_id].switch\_status);  break;  case my\_toggle\_switch\_3:  **Serial**.print("case statement 6 entered, switch is ");  **Serial**.println(my\_switches.switches[switch\_id].switch\_status);  break;  default:  // Spurious switch index! Should never arise as this is  // controlled by the for loop within defined upper bound  break;  }  **Serial**.flush(); // flush out the output buffer  }  } | | | |

| Type | bool | Name | read\_button\_switch |
| --- | --- | --- | --- |
| Parameters | byte switch\_id | | |
| Purpose / functionality | This is used by the read\_switch function and deals specifically with reading momentary button style switches.  *The function can be used by end user code, but remember that the switch\_id parameter is the switch entry number in the switch control structure of the switch to be read.* | | |
| Return values | switched or !switched | | |
| Example | | | |
| if (my\_switches.read\_button\_switch(switch\_id) == switched){  // button switch pressed  ...  } | | | |

| Type | bool | Name | read\_toggle\_switch |
| --- | --- | --- | --- |
| Parameters | byte switch\_id | | |
| Purpose / functionality | This is used by the read\_switch function and deals specifically with reading toggle style switches.  *The function can be used by end user code, but remember that the switch\_id parameter is the switch entry number in the switch control structure of the switch to be read.* | | |
| Return values | switched or !switched | | |
| Example | | | |
| if (my\_switches.read\_toggle\_switch(switch\_id) == switched){  // toggle switch switched  ...  } | | | |

| Type | void | Name | print\_switch |
| --- | --- | --- | --- |
| Parameters | byte switch\_id | | |
| Purpose / functionality | The function prints the switch parameters of the switch defined at slot switch\_id in the switch control structure to the serial monitor. | | |
| Return values | none | | |
| Example | | | |
| my\_switches.print\_switch(3);  Example output a toggle switch, configured as circuit\_C2 and occupying slot 3 (switch\_id = 3)in the switch control structure:  slot: 3 sw\_type= 2 sw\_pin= 5 circ\_type= 2 pending= 0 elapse= 0 on\_value= 0 sw\_status= 0 | | | |

| Type | void | Name | print\_switches |
| --- | --- | --- | --- |
| Parameters | none | | |
| Purpose / functionality | The function prints the switch parameters of ALL switches held in the switch control structure to the serial monitor. | | |
| Return values | none | | |
| Example | | | |
| my\_switches.print\_switches();  Example output for 6 defined switches - 3 x button & 3 x toggle, configured as either circuit\_C1 or circuit\_C2:  slot: 0 sw\_type= 1 sw\_pin= 2 circ\_type= 0 pending= 0 elapse= 0 on\_value= 1 sw\_status= 1  slot: 1 sw\_type= 1 sw\_pin= 3 circ\_type= 2 pending= 0 elapse= 0 on\_value= 0 sw\_status= 1  slot: 2 sw\_type= 1 sw\_pin= 4 circ\_type= 0 pending= 0 elapse= 0 on\_value= 1 sw\_status= 1  slot: 3 sw\_type= 2 sw\_pin= 5 circ\_type= 2 pending= 0 elapse= 0 on\_value= 0 sw\_status= 0  slot: 4 sw\_type= 2 sw\_pin= 6 circ\_type= 0 pending= 0 elapse= 0 on\_value= 1 sw\_status= 0  slot: 5 sw\_type= 2 sw\_pin= 7 circ\_type= 2 pending= 0 elapse= 0 on\_value= 0 sw\_status= 0 | | | |

| Type | void | Name | set\_debounce |
| --- | --- | --- | --- |
| Parameters | int period | | |
| Purpose / functionality | The function may be used to set the debounce period, in milliseconds, for switch reading functions.  Note that:   1. the debounce value is set to 10 milliseconds, by default 2. the debounce setting is global and applies to ALL defined switches 3. the parameter value must be >= 0. Negative values are ignored. | | |
| Return values | none | | |
| Example | | | |
| my\_switches.set\_debounce(20); // set debounce for 20 msecs | | | |

# Example Sketches