**ESP32 asynchronous multi-button library**

**J. Lemaire**

Pierrefeu – November 2019

# Objectives

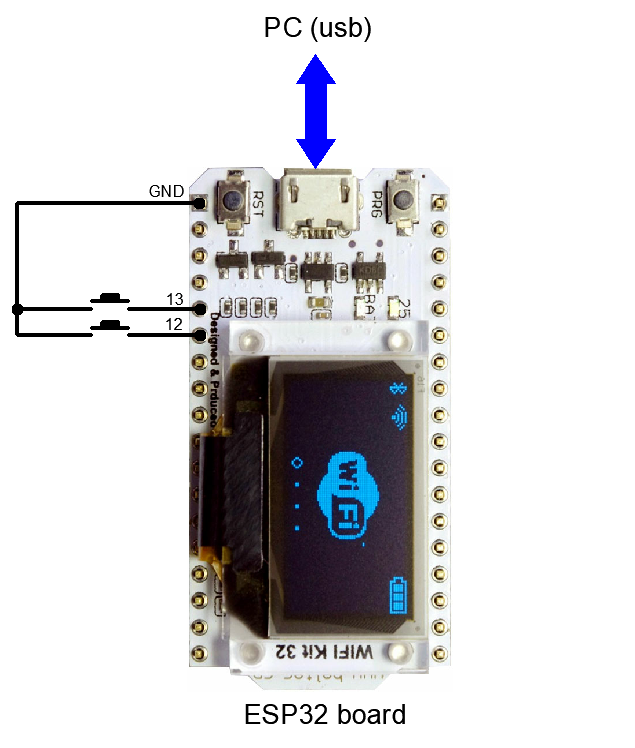
A button library is an old subject and there are many candidates on internet[[1]](#footnote-2) [[2]](#footnote-3)!

But, with the following constraints :

* Can use many push buttons, each on a different pin ;
* Short and long press detection[[3]](#footnote-4) ;
* Debounce treament ;
* Interrupt driven with embeded callbacks in the library (asynchronous treatment) ;
* Can run on ESP32 with the Espressif Arduino Core ;
* As simple as possible !

apparently no candidates… But may be I omitted one and asked the author to excuse me !

So, I choose to develop such a library, convenient for me, and tested it using this very simple circuit with an ESP32 board[[4]](#footnote-5) :

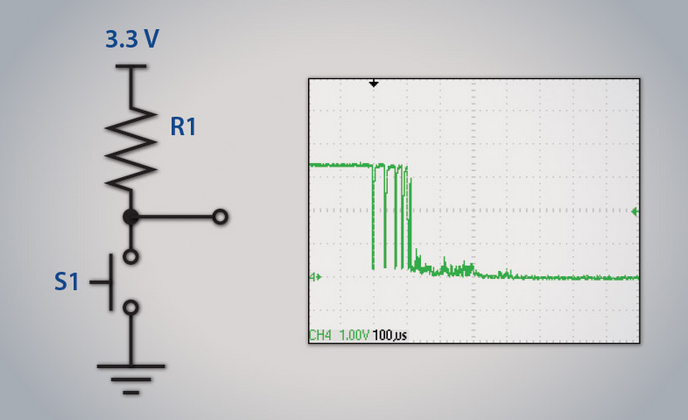


# Debounce management and detect the press action

Here, each push button is connected to the ground and to a pin which GPIO has an internal pull-up resistance[[5]](#footnote-6) ; hence, the **digitalRead** fonction will return its level :

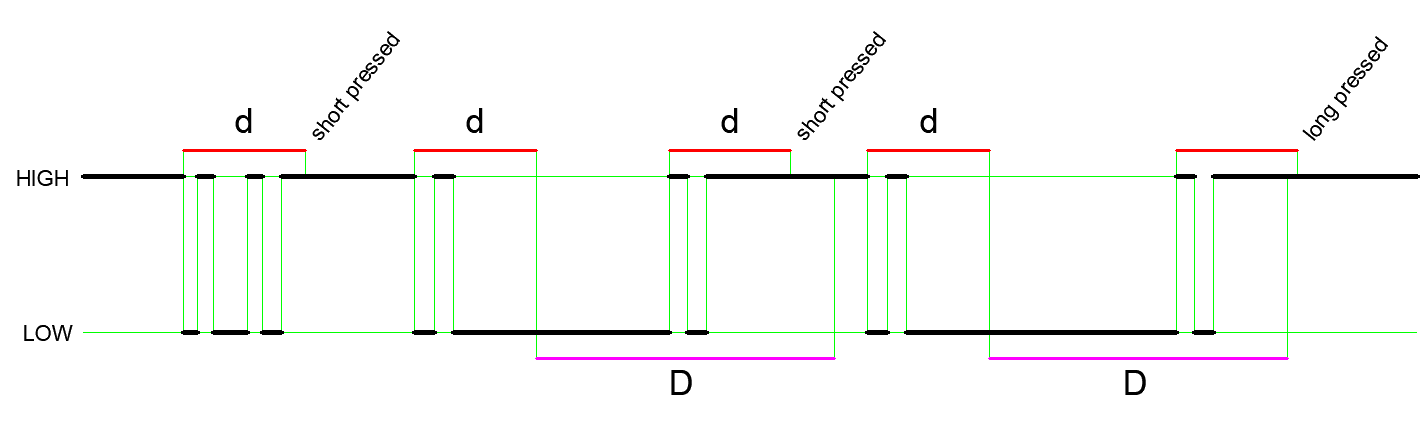
* LOW if the button is pushed
* HIGH if the button is released

Unfortunately and due to the contact and its imperfections, an action on such a button doesn’t produce immediatly the expected change in the GPIO level, but in fact a serie of alternances during a short period ; e.g. :

 [[6]](#footnote-7)

Knowing that these changes can be detected using an external interrupt and an attached callback procedure (ISR[[7]](#footnote-8)), we just need here to do nothing during a laps **d** of time, inside the ISR so that only one change can be registred.

Otherwise, it’s also important to evaluate the GPIO level after this delay **d** of stabilisation and to measure the delay between 2 consécutive evaluations in view to make the difference between short and long press (delay >= **D**) actions on the button :



More precisely :

|  |  |  |  |
| --- | --- | --- | --- |
| **From** | **To** | **Delay from**  **previous eval.** | **Action** |
| HIGH | HIGH |  | short pressed |
| LOW | HIGH | < D | short pressed |
| LOW | HIGH | >= D | long pressed |

Practically, in the ISR procedure, an alarm timer is enabled which calls, only one time after the delay **d**, a second ISR procedure with the following tasks :

* Evaluate the new level, using **digitalRead**
* Discard the case where different buttons are pressed at the same time
* If LOW, store the time using **Millis**
* Determine the action on the button as previously explained and store it
* Reactive the treatment of the change interrupt on the GPIO

# The MButton Class

A simple library declares the **MButton** class in the the file **MButton.h**:

// MButton.h

/\*

ESP32 LIBRARY TO DETECT ASYNCHRONOUSLY SHORT AND LONG PRESS OF SEVERAL PUSH BUTTONS

Restrictions :

- Each button is connected to the ground and a different pin

- Their gpios need to have a pull-up resistance (e.g. pin number 12, 13, etc.)

- Only one button can be pressed at a time (no buffer to register the actions)

- Need to wait that an action has been processed before a new action

Usage :

- declare an instance of the class

- in the setup function call begin(<no Buttons>, <num1>, <num2>, ...) to precise the pins and attach their callbacks

- in the loop function call successively toProccess(), getNum(), getAction(), processed()

- can call end() to detach callbacks

This program is free software: you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation, either version 3 of the License, or

(at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program. If not, see <http://www.gnu.org/licenses/>.

\*/

# pragma once

#include "Arduino.h"

#include <Ticker.h>

#define DELAY\_DEBOUNCE 200 // Delay for debounce (ms)

#define DELAY\_LONG\_PRESS 1000 // Delay for long press (ms)

class MButton

**{**

public**:**

MButton**();** // Constructor

void begin**(**int noButtons**,** **...);** // To define the pin numbers and attach callbacks

void end**();** // To detach the callbacks

bool toProcess**();** // True if there is a pending action on the button

int getNum**();** // Pin number

int getAction**();** // Action number : 1 => pressed, 2 => long pressed

void processed**();** // To call after action treatment (needed !)

private**:**

static volatile int \_num**;** // Pin number

static volatile bool \_toProcess**;** // True if action to process

static volatile int \_action**;** // 1 => pressed, 2 => longpressed

static bool \_okButton**;** // To avoid debounce

static int \_oldNum**;** // To save pin number so that mixed action are forbidden

static unsigned long \_oldTime**;** // To save time (ms) for detecting long press action

static int \_oldLevel**;** // To save pin level for detecting actions (LOW or HIGH)

static Ticker \_timer**;** // Alarm timer

static void IRAM\_ATTR buttonInterrupt12**();**

static void IRAM\_ATTR buttonInterrupt13**();**

// Here can add ISR for other pins

static void IRAM\_ATTR timerInterrupt**();**

**};**

The implémentation is in the file **MButton.cpp**:

// MButton.cpp

#include "MButton.h"

#include <Ticker.h>

// Implement static members

volatile int MButton**::**\_num**;**

volatile bool MButton**::**\_toProcess**;**

volatile int MButton**::**\_action**;**

bool MButton**::**\_okButton**;**

int MButton**::**\_oldNum**;**

unsigned long MButton**::**\_oldTime**;**

int MButton**::**\_oldLevel**;**

Ticker MButton**::**\_timer**;**

MButton**::**MButton**()**

**{**

\_okButton **=** **true;**

\_oldLevel **=** HIGH**;**

\_toProcess **=** **false;**

**}**

void MButton**::**begin**(**int noMButtons**,** **...)**

**{**

va\_list arg**;**

va\_start**(**arg**,** noMButtons**);**

**for** **(**uint8\_t i **=** 0**;** i **<** noMButtons**;** i**++)**

**{**

uint8\_t num **=** va\_arg**(**arg**,** int**);**

pinMode**(**num**,** INPUT\_PULLUP**);**

**switch** **(**num**)**

**{**

**case** 12**:**

attachInterrupt**(**12**,** buttonInterrupt12**,** CHANGE**);**

**break;**

**case** 13**:**

attachInterrupt**(**13**,** buttonInterrupt13**,** CHANGE**);**

**break;**

// Here can add attachments for other pins

**}**

**}**

**}**

void MButton**::**end**()**

**{**

detachInterrupt**(**12**);**

detachInterrupt**(**13**);**

// Here can add detachments for other pins

\_timer**.**detach**();**

**}**

int MButton**::**getNum**()**

**{**

**return** \_num**;**

**}**

int MButton**::**getAction**()**

**{**

**return** \_action**;**

**}**

bool MButton**::**toProcess**()**

**{**

**return** \_toProcess**;**

**}**

void MButton**::**processed**()**

**{**

\_toProcess **=** **false;**

**}**

void IRAM\_ATTR MButton**::**buttonInterrupt12**()**

**{**

**if** **(**\_okButton **&&** **!**\_toProcess**)**

**{**

\_num **=** 12**;**

\_okButton **=** **false;**

\_timer**.**once\_ms**(**DELAY\_DEBOUNCE**,** timerInterrupt**);**

**}**

**}**

void IRAM\_ATTR MButton**::**buttonInterrupt13**()**

**{**

**if** **(**\_okButton **&&** **!**\_toProcess**)**

**{**

\_num **=** 13**;**

\_okButton **=** **false;**

\_timer**.**once\_ms**(**DELAY\_DEBOUNCE**,** timerInterrupt**);**

**}**

**}**

void MButton**::**timerInterrupt**()**

**{**

int level **=** digitalRead**(**\_num**);**

**if** **(!**\_toProcess**)**

**{**

**if** **(**\_oldLevel **==** HIGH**)**

**{**

**if** **(**level **==** HIGH**)**

**{**

// Short pressed

\_action **=** 1**;**

\_toProcess **=** **true;**

**}**

**else**

**{**

// May start a long press action

\_oldNum **=** \_num**;**

\_oldTime **=** millis**();**

\_oldLevel **=** LOW**;**

**}**

**}**

**else** **if** **((**level **==** HIGH**))**

**{**

**if** **(**\_num **!=** \_oldNum**)** // Forbidden mixed case

**{**

\_num **=** \_oldNum**;**

**}**

**else**

**{**

**if** **(**millis**()** **>=** **(**\_oldTime **+** DELAY\_LONG\_PRESS**))**

**{**

// Long press

\_action **=** 2**;**

**}**

**else**

**{**

// Short press not long enough...

\_action **=** 1**;**

**}**

\_oldLevel **=** HIGH**,**

\_toProcess **=** **true;**

**}**

**}**

**}**

\_okButton **=** **true;** // Enable button interrupts

**}**

It’s not difficult to understand what is realized in this library and why.

* As announced, an **MButton** class instance can manage several push buttons. They are precised on calling the **begin** method which accepts a variable number of parameters ; e.g. **begin(2, 12, 13)** in our case.
* The crucial point, as previously introduced, is the asynchronous treatment using interrupts and the definition of their attached ISR functions. They need to be static[[8]](#footnote-9) and then only use static variables. For that reason, we have opted to declare static variables members ; we can do it because only one button can be pressed at a time and because the actions are not bufferized.
* Each pin, connected to a push button, has its own ISR. Observe that it can do something only when **\_okButton** is true (debounce condition) and **\_toProcess** is false (cannot register a new action until the previous action is processed). This ISR stores in the variable **\_num** the number of the pin, sets **\_okButton** at false and starts the alarm timer. Obviously, if other pins are considered, the library need to be upgraded a little.
* Here the alarm timer is an hardware one[[9]](#footnote-10) and we use a static instance **\_timer** of the **Ticker** class in the **Ticker** library added with the Arduino Core for ESP32[[10]](#footnote-11) ; we just have to call :

\_timer**.**once\_ms**(**DELAY\_DEBOUNCE**,** timerInterrupt**);**

to start the timer so that the ISR **timerInterrupt** should be called only one time, after **DELAY\_DEBOUNCE** ms.

* The ISR **timerInterrupt** performs the tasks described in the previous paragraph.
* And finally, some public methods are provided, **constructor**, **begin**, **toProcess**, **getNum**, **getAction**, **processed** and **end**, to access to the library as described in the following example.

The Arduino sketch :

// MButton\_test.ino

// Multi button (on pins 12 and 13 here) library test

#include "MButton.h"

#include <elapsedMillis.h>

#define DELAY\_TEST 60000 // Delay for the test (ms)

MButton mButton**;**

elapsedMillis timerTest**;**

bool active**;**

void setup**()**

**{**

Serial**.**begin**(**115200**);**

**while(!**Serial**);**

mButton**.**begin**(**2**,** 12**,** 13**);**

timerTest **=** 0**;**

active **=** **true;**

Serial**.**println**(**"Test started"**);**

**}**

void loop**()**

**{**

**if** **(**timerTest **<** DELAY\_TEST**)**

**{**

**if** **(**mButton**.**toProcess**())**

**{**

Serial**.**print**(**mButton**.**getNum**());**

**switch** **(**mButton**.**getAction**())**

**{**

**case** 1**:**

Serial**.**println**(**" short pressed"**);**

**break;**

**case** 2**:**

Serial**.**println**(**" long pressed"**);**

**break;**

**}**

mButton**.**processed**();**

**}**

**}**

**else** **if** **(**active**)**

**{**

mButton**.**end**();**

active **=** **false;**

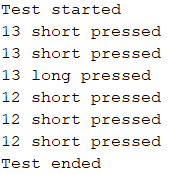
Serial**.**println**(**"Test ended"**);**

**}**

delay**(**500**);**

**}**

And finally an output example :



**Remarks**

* Here, no critical section seems to be needed if we respect the constraints on the actions. But, often they are needed and in this situation a classical solution consist in using **portMUX\_TYPE** variables[[11]](#footnote-12), because the Arduino core for ESP32 doesn’t implement the classical **cli()** and **sti()** functions :
* Declaration :

portMUX\_TYPE timerMux **=** portMUX\_INITIALIZER\_UNLOCKED**;**

* Usage :

portENTER\_CRITICAL\_ISR**(&**timerMux**);**

// Here critical section

portEXIT\_CRITICAL\_ISR**(&**timerMux**);**

* We opted to declare an ISR for each pin connected to a push button : it’s simple and not really difficult to consider other pins. However, there is a solution to avoid this constraint and use only one ISR for all the pins : store the state of all the pins and detect what pin has changed, as proposed in the Alarm-Siren library[[12]](#footnote-13). May be there exist simpler solutions but I don’t found !

1. E.g. <https://github.com/LennartHennigs/Button2> but with no interrupts. [↑](#footnote-ref-2)
2. E.g. <https://github.com/Alarm-Siren/arduino-buttons> but without managing long press action… and with empty arduino sketches ! [↑](#footnote-ref-3)
3. The « double clic » on the buttons has not been considered here, but it can be easily managed using the presented method [↑](#footnote-ref-4)
4. HELTEC Wifi kit 32 : <https://heltec.org/project/wifi-kit-32/> . [↑](#footnote-ref-5)
5. E.g. gpio number 34 to 39 does’nt have internal resistance. [↑](#footnote-ref-6)
6. From <https://www.nuvation.com/resources/article/switch-debouncing-electronic-product-designs> . [↑](#footnote-ref-7)
7. ISR = Interrupt Service Routine. [↑](#footnote-ref-8)
8. E.g. <https://randomnerdtutorials.com/esp32-pir-motion-sensor-interrupts-timers/> [↑](#footnote-ref-9)
9. <https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/timer.html> [↑](#footnote-ref-10)
10. <https://github.com/espressif/arduino-esp32/tree/master/libraries/Ticker> [↑](#footnote-ref-11)
11. E.g. <https://techtutorialsx.com/2017/09/30/esp32-arduino-external-interrupts/> [↑](#footnote-ref-12)
12. <https://github.com/Alarm-Siren/arduino-buttons> [↑](#footnote-ref-13)