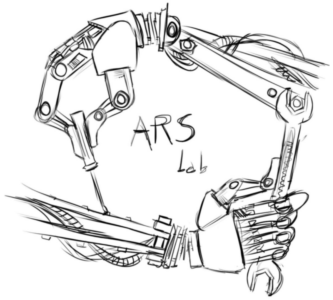
Handling Asynchronous Events in MCUs
A Light Keyboard Driver

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



L.A.P. 1 Course

Blocking vs Non-blocking operations

The “toggle” Example

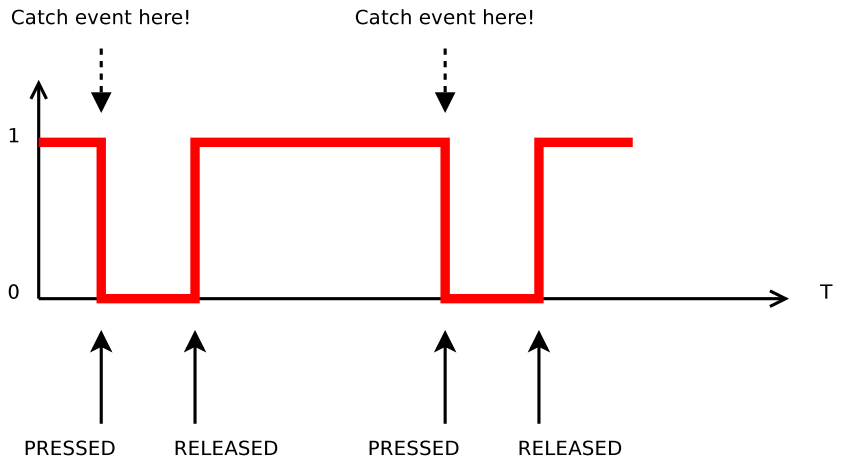
We want to **toggle** LED in PB8 each time the button in PA10 is pressed.

```
#include "stm32_unict_lib.h"

int main()
{
    int ledstate = 0;
    // pushbutton on PA10; LED on PB8
    // initialize ports
    GPIO_init(GPIOA);
    GPIO_init(GPIOB);

    GPIO_config_input(GPIOA, 10); // PA10 as input
    GPIO_config_output(GPIOB, 8); // PB8 as output

    // infinite loop
    for (;;) {
        if (GPIO_read(GPIOA, 10) == 0) {
            ledstate = !ledstate;
            GPIO_write(GPIOB, 8, ledstate);
        }
    }
}
```

This code **won’t work**: the loop is executed a very high speed, so the “if” is true many times and the LED toggles continuously!

Let's analyse the signal...

Pressing/releasing a button implies the following timing:

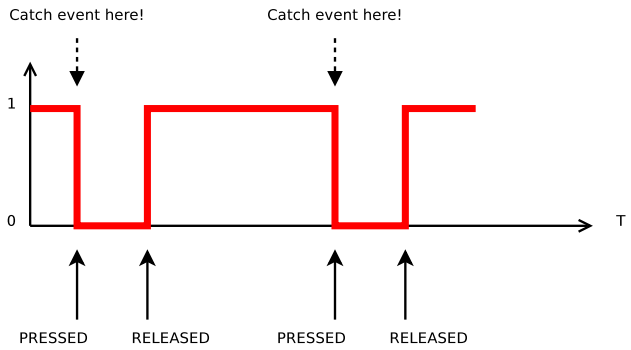


We have to identify the **PRESSED** event, i.e. **a transition from high-to-low**.

Solution 1

- Identify that the button is pressed (check that PA10 is "0")
- Do the proper actions (i.e. toggle the LED)
- **wait for button release** before continuing (ensure that PA10 is "1")

Toggle Example: Solution 1




```
int main()
{
    int ledstate = 0;

    // initialize ports
    GPIO_init(GPIOA);
    GPIO_init(GPIOB);

    GPIO_config_input(GPIOA, 10); // PA10 as input
    GPIO_config_output(GPIOB, 8); // PB8 as output
    // infinite loop
    for (;;) {
        if (GPIO_read(GPIOA, 10) == 0) {
            ledstate = !ledstate;
            GPIO_write(GPIOB, 8, ledstate);

            while (GPIO_read(GPIOA, 10) == 0) {} // wait
        }
    }
}
```

The “toggle” Example 2

We want to:

- Toggle LED in PB8 each time the button in PA10 is pressed,
- ...meanwhile, we want to make LED in PB9 **flash** at a on/off period of 200*ms*!!

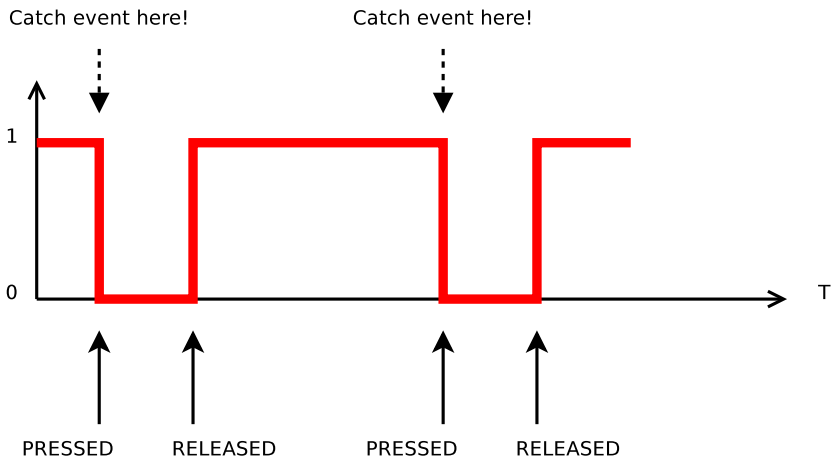
To do this, we should employ a hardware or software timer, but it won’t work since we have **busy-waits!!!**

Indeed, during busy-waits, the program cannot do anything else: I could leave my finger on the button for a very long time!!!! :-D

The **non-working** “toggle” Example 2

```
int main()
{
  int ledstate = 0;
  // initialize ports
  GPIO_init(GPIOA);
  GPIO_init(GPIOB);

  GPIO_config_input(GPIOA, 10); // PA10 as input
  GPIO_config_output(GPIOB, 8); // PB8 as output
  GPIO_config_output(GPIOB, 9); // PB9 as output
  // infinite loop
  for (;;) {

    if (GPIO_read(GPIOA, 10) == 0) {
      ledstate = !ledstate;
      GPIO_write(GPIOB, 8, ledstate);

      while (GPIO_read(GPIOA, 10) == 0) {} // wait
    }

    delay_ms(200);
    GPIO_toggle(GPIOB, 9);
  }
}
```

No Busy-Waits!!



Solution 2: Let's identify transitions using a STATE

- Use a state variable to **store** the button state at the **previous iteration** (the **past**)
- Read current button state (the **present**)
- Check that the **past** and **present** are different
- Update the state variable (the **present** becomes the **past**)

The **working** "toggle" Example 2

```
int main()
{
  int last_button_state, ledstate = 0;
  // initialize ports
  GPIO_init(GPIOA);
  GPIO_init(GPIOB);

  GPIO_config_input(GPIOA, 10); // PA10 as input
  GPIO_config_output(GPIOB, 8); // PB8 as output
  GPIO_config_output(GPIOB, 9); // PB9 as output

  last_button_state = GPIO_read(GPIOA, 10);
  for (;;) {

    int current_button_state = GPIO_read(GPIOA, 10);

    // Transition 1-to-0 identification
    if (last_button_state == 1 && current_button_state == 0) {
      ledstate = !ledstate;
      GPIO_write(GPIOB, 8, ledstate);
    }
    last_button_state = current_button_state; // present becomes past

    delay_ms(200);
    GPIO_toggle(GPIOB, 9);
  }
}
```

Towards a “light” keyboard device driver

A mess of concerns!!

- The previous example, while working, mingles two different aspects of programming:
  1. **Key handling/event recognition**, general purpose, not application-specific
  2. **Action execution** (LED toggle), application-specific
- If we need to handle keys in another application, we need to “extract” the code... this is not so “elegant”
- Indeed, usually aspect (1) is managed by an operating system or a keyboard device driver

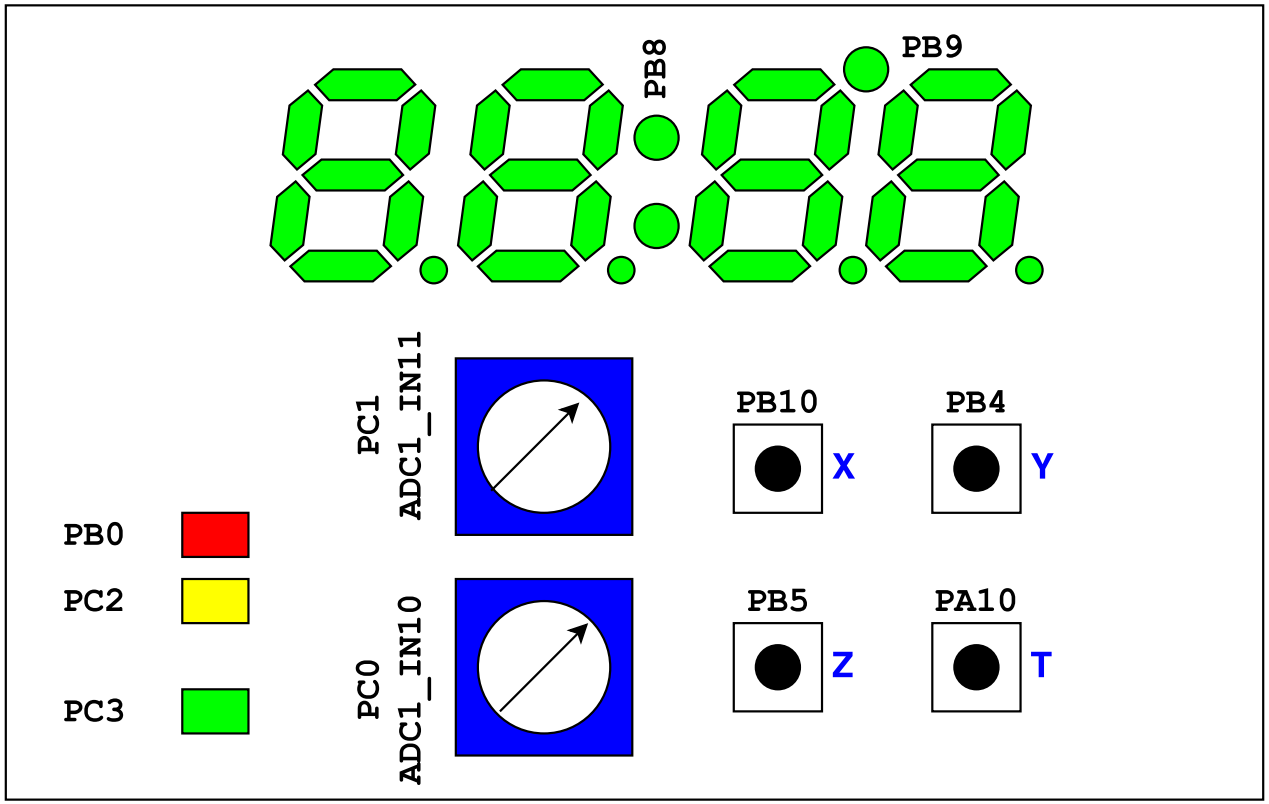
A mess of concerns!!

Red lines are related to key handling, others are application-specific

```
int main()
{
  int last_button_state;
  int ledstate = 0;

  GPIO_init(GPIOA);
  GPIO_init(GPIOB);

  GPIO_config_input(GPIOA, 10); // PA10 as input
  GPIO_config_output(GPIOB, 8); // PB8 as output
  GPIO_config_output(GPIOB, 9); // PB9 as output

  last_button_state = GPIO_read(GPIOA, 10);
  for (;;) {

    int current_button_state = GPIO_read(GPIOA, 10);

    // Transition 1-to-0 identification
    if (last_button_state == 1 && current_button_state == 0) {
      ledstate = !ledstate;
      GPIO_write(GPIOB, 8, ledstate);
    }
    last_button_state = current_button_state; // present becomes past

    delay_ms(200);
    GPIO_toggle(GPIOB, 9);
  }
}
```

Let's separate our concerns!!

- First let give a sybolic name to each one of our keys:



Let's separate our concerns!!

- To handle the "mess", we can imagine a light **keyboard driver**, represented by a function:
  `int read_key();`
- The function checks if a key has been pressed
- It recognises high-to-low and low-to-high key transistions
- It is non-blocking
- It returns the following possible (symbolic) values:
  - `KEY_NONE` no key event detected
  - `KEY_PRESS_X` Button "X" has been pressed
  - `KEY_RELEASE_X` Button "X" has been released
  - `KEY_PRESS_Y` Button "Y" has been pressed
  - `KEY_RELEASE_Y` Button "Y" has been released
  - `KEY_PRESS_Z` Button "Y" has been pressed
  - `KEY_RELEASE_Z` Button "Y" has been released
  - `KEY_PRESS_T` Button "Y" has been pressed
  - `KEY_RELEASE_T` Button "Y" has been released

Key Handling

- To manage the keyboard, we must use a different ``last_button_state'' for each key

```
int last_X_state, last_Y_state, last_Z_state, last_T_state;

int key_read(void)
{
    int curr_state, key_event = KEY_NONE;

    current_state = GPIO_read(GPIOB, 10); // read "X"
    if (last_X_state == 1 && current_state == 0) key_event = KEY_PRESS_X;
    if (last_X_state == 0 && current_state == 1) key_event = KEY_RELEASE_X;
    last_X_state = current_state;

    curr_Y_state = GPIO_read(GPIOB, 4); // read "Y"
    if (last_Y_state == 1 && current_state == 0) key_event = KEY_PRESS_Y;
    if (last_Y_state == 0 && current_state == 1) key_event = KEY_RELEASE_Y;
    last_Y_state = current_state;

    current_state = GPIO_read(GPIOB, 5); // read "Z"
    if (last_Z_state == 1 && current_state == 0) key_event = KEY_PRESS_Z;
    if (last_Z_state == 0 && current_state == 1) key_event = KEY_RELEASE_Z;
    last_Z_state = current_state;

    current_state = GPIO_read(GPIOA, 10); // read "T"
    if (last_T_state == 1 && current_state == 0) key_event = KEY_PRESS_T;
    if (last_T_state == 0 && current_state == 1) key_event = KEY_RELEASE_T;
    last_T_state = current_state;

    return key_event;
}
```

Key Handling

- Let's add the "initialization" part

```
// X = PB10
// Y = PB4
// Z = PB5
// T = PA10
void key_init(void)
{
    GPIO_init(GPIOA);
    GPIO_init(GPIOB);
    GPIO_config_input(GPIOB, 10);
    GPIO_config_input(GPIOB, 4);
    GPIO_config_input(GPIOB, 5);
    GPIO_config_input(GPIOA, 10);
    last_X_state = GPIO_read(GPIOB, 10);
    last_Y_state = GPIO_read(GPIOB, 4);
    last_Z_state = GPIO_read(GPIOB, 5);
    last_T_state = GPIO_read(GPIOA, 10);
}
```

Key Handling

- ... and a `main()` to test the driver

```
int main(void)
{
    key_init();

    GPIO_config_output(GPIOB, 8);
    GPIO_config_output(GPIOB, 9);

    for (;;) {
        switch (key_read()) {
          case KEY_PRESS_X:
            GPIO_toggle(GPIOB, 8);
            break;
          case KEY_RELEASE_T:
            GPIO_toggle(GPIOB, 9);
            break;
        }
    }
}
```

Refactor to hav a “better” code

- The **`key_read()`** function uses a “cut-and-paste” style programming: very bad!!
- If we need to add another key, another block must be “pasted”

```
int last_X_state, last_Y_state, last_Z_state, last_T_state;

int key_read(void)
{
    int curr_state, key_event = KEY_NONE;

    current_state = GPIO_read(GPIOB, 10); // read "X"
    if (last_X_state == 1 && current_state == 0) key_event = KEY_PRESS_X;
    if (last_X_state == 0 && current_state == 1) key_event = KEY_RELEASE_X;
    last_X_state = current_state;

    curr_Y_state = GPIO_read(GPIOB, 4); // read "Y"
    if (last_Y_state == 1 && current_state == 0) key_event = KEY_PRESS_Y;
    if (last_Y_state == 0 && current_state == 1) key_event = KEY_RELEASE_Y;
    last_Y_state = current_state;

    current_state = GPIO_read(GPIOB, 5); // read "Z"
    if (last_Z_state == 1 && current_state == 0) key_event = KEY_PRESS_Z;
    if (last_Z_state == 0 && current_state == 1) key_event = KEY_RELEASE_Z;
    last_Z_state = current_state;

    current_state = GPIO_read(GPIOA, 10); // read "T"
    if (last_T_state == 1 && current_state == 0) key_event = KEY_PRESS_T;
    if (last_T_state == 0 && current_state == 1) key_event = KEY_RELEASE_T;
    last_T_state = current_state;

    return key_event;
```

The “Key” concept

- In our driver, the basic “concept” is the **key**, that is characterized by:
  - **The GPIO port** (static info)
  - **The GPIO pin** (static info)
  - **The “last state”** (dynamic info)
- So let’s encapsulate this information into a **struct**:

```
struct key_state {
    GPIO_TypeDef * port;
    int pin;
    int last_state;
};
```

The “Key” concept

- And let’s use an array statically initialized with keyboard configuration data:

```
#define NUMBER_OF_KEYS 4

struct key_state {
    GPIO_TypeDef * port;
    int pin;
    int last_state;
} keyboard[NUMBER_OF_KEYS] = { {GPIOB, 10, 1},    // key "X"
                               {GPIOB, 4, 1},     // key "Y"
                               {GPIOB, 5, 1},     // key "Z"
                               {GPIOA, 10, 1} }; // key "T"
```

The “Key” concept

- Now keyboard management is made very easy and easily upgradable:

```
#define KEY_NONE -1

#define KEY_PRESS_BASE 0
#define KEY_PRESS_X  0
#define KEY_PRESS_Y  1
#define KEY_PRESS_Z  2
#define KEY_PRESS_T  3

#define KEY_RELEASE_BASE 10
#define KEY_RELEASE_X  10
#define KEY_RELEASE_Y  11
#define KEY_RELEASE_Z  12
#define KEY_RELEASE_T  13

int key_read(void)
{
    int i, key_event = KEY_NONE;

    for (i = 0;i < NUMBER_OF_KEYS;i++) {
        int curr_state = GPIO_read(keyboard[i].port, keyboard[i].pin);
        if (keyboard[i].last_state == 1 && curr_state == 0)
            key_event = KEY_PRESS_BASE + i;
        if (keyboard[i].last_state == 0 && curr_state == 1)
            key_event = KEY_RELEASE_BASE + i;
        keyboard[i].last_state = curr_state;
    }
    return key_event;
}
```

The “Key” concept

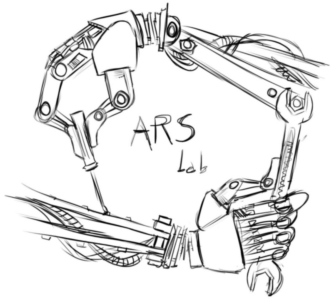
- Also initialization is made easy and upgradable:

```
void key_init(void)
{
    int i;
    for (i = 0;i < NUMBER_OF_KEYS;i++) {
        GPIO_init(keyboard[i].port);
        GPIO_config_input(keyboard[i].port, keyboard[i].pin);
        keyboard[i].last_state = GPIO_read(keyboard[i].port, keyboard[i].pin);
    }
}
```

Handling Asynchronous Events in MCUs
A Light Keyboard Driver

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



L.A.P. 1 Course