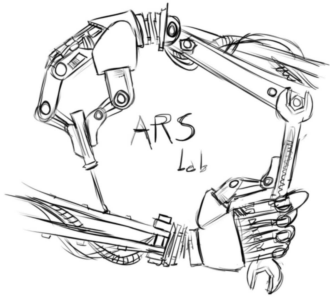
Handling Asynchronous Events in MCUs
The Finite-State Machine Design Pattern

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

The “toggle” Example

We want to **toggle** LED in RB0 each time the button in RA3 is pressed.

Electrical considerations:

- the logic of the LED is **inverted**
  - `LATBbits.LATB0 = 1;` implies LED off;
  - `LATBbits.LATB0 = 0;` implies LED on;
- the logic of the pushbutton is also **inverted**
  - Button pressed implies `PORTAbits.RA3 == 0`;
  - Button NOT pressed implies `PORTAbits.RA3 == 1`;

The “toggle” example

The listing

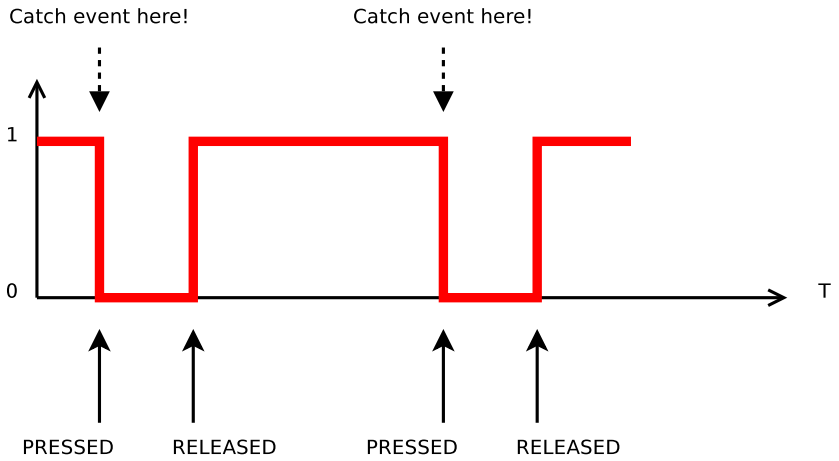
```
main()
{
  TRISAbits.TRISA3 = 1; // RA3 as input
  TRISBbits.TRISB0 = 0; // RB0 as output
  LATBbits.LATB0 = 1; // turn led off initially

  for (;;) {  // loop forever
    if (PORTAbits.RA3 == 0)
      LATBbits.LATB0 = !LATBbits.LATB0;
  }
}
```

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:

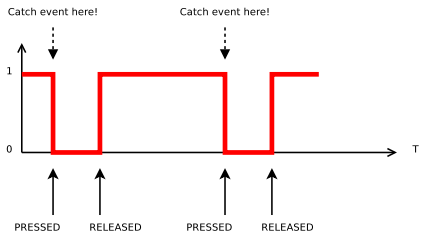


We have to identify the **PRESSED** event, i.e.**a transition from high-to-low**.

Solution 1

- Identify that the button is pressed (check that RA3 is "0")
- Do the proper actions (i.e. toggle the LED)
- **wait for button release** before continuing (ensure that RA3 is "1")

Let's analyse the signal...



Solution 1

- Identify that the button is pressed (check that RA3 is "0")
- Do the proper actions (i.e. toggle the LED)
- **wait for button release** before continuing (ensure that RA3 is "1")

```
main()
{
  TRISAbits.TRISA3 = 1; // RA3 as input
  TRISBbits.TRISB0 = 0; // RB0 as output
  LATBbits.LATB0 = 1; // turn led off initially
  for (;;) {  // loop forever
    if (PORTAbits.RA3 == 0) {
      LATBbits.LATB0 = !LATBbits.LATB0; // toggle the led
      while (PORTAbits.RA3 == 0) {} // busy-wait for release
    }
  }
}
```

The “toggle” Example v2

We want to:

- Toggle LED in RB0 each time the button in RA3 is pressed.
- Toggle LED in RB1 each time the button in RA2 is pressed.

The solution is quite simple now: let’s duplicate the code!

The “toggle” Example v2

```
main()
{
  TRISAbits.TRISA3 = 1; // RA3 as input
  TRISAbits.TRISA2 = 1; // RA2 as input
  TRISBbits.TRISB0 = 0; // RB0 as output
  TRISBbits.TRISB1 = 0; // RB1 as output

  LATBbits.LATB0 = 1; // turn led off initially
  LATBbits.LATB1 = 1; // turn led off initially

  for (;;) {  // loop forever

    if (PORTAbits.RA3 == 0) {
      LATBbits.LATB0 = !LATBbits.LATB0; // toggle the led
      while (PORTAbits.RA3 == 0) {} // busy-wait for release
    }

    else if (PORTAbits.RA2 == 0) {
      LATBbits.LATB1 = !LATBbits.LATB1; // toggle the led
      while (PORTAbits.RA2 == 0) {} // busy-wait for release
    }

  }
}
```

The “toggle” Example v3

We want to:

- Toggle LED in RB0 each time the button in RA3 is pressed.
- Toggle LED in RB1 each time the button in RA2 is pressed.
- ...meanwhile, we want to make LED in RB2 **flash** at a certain frequency!!

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 v3

```
main()
{
  long count = 0; // 32 bit is better!

  // ... set-up ports by TRIS registers

  for (;;) {  // loop forever
    if (PORTAbits.RA3 == 0) {
      LATBbits.LATB0 = !LATBbits.LATB0; // toggle the led
      while (PORTAbits.RA3 == 0) {} // busy-wait for release
    }
    else if (PORTAbits.RA2 == 0) {
      LATBbits.LATB1 = !LATBbits.LATB1; // toggle the led
      while (PORTAbits.RA2 == 0) {} // busy-wait for release
    }
    ++count;
    if (count == MAX) {
      LATBbits.LATB2 = !LATBbits.LATB2; // toggle the led
      count = 0;
    }
  }
}
```

No Busy-Waits!!

In the previous solution the variable **`count`** won't be incremented during busy-waits, so the flashing frequency is not guaranteed!

Solution 2: Let's encode a STATE

- Use a variable to **store** the previous **button state**
- Check the variable and the button's state are different
- Update the state variable

No Busy-Waits

```
main()
{
  int button_1_state;

  // ... set-up ports by TRIS registers

  int button_1_state = PORTAbits.RA3;
  for (;;) {  // loop forever
    ...
    if (button_1_state != PORTAbits.RA3) {
      // ok, the button has changed state!
      if (PORTAbits.RA3 == 0) {
        // OK!! We have got button press
        LATBbits.LATB0 = !LATBbits.LATB0; // toggle the led
      }
      button_1_state = PORTAbits.RA3;
    }
    ...
  }
}
```

Atomic Reads

In the previous solution, **`PORTAbits.RA3`** is read **three times** in the same loop.
It is not guaranteed that the value read is *always the same!!*

Solution 3: Let's encode a STATE

- Use a variable to represent the **button state** you wish to **identify**, e.g. FOR_PRESS, FOR_RELEASE
- Check the button's state according to the state variable
- Update the state variable

No Busy-Waits with atomic reads

```
#define FOR_PRESS   0
#define FOR_RELEASE 1
main()
{
  int button_1_state = FOR_PRESS;

  // ... set-up ports by TRIS registers

  for (;;) {  // loop forever
    ...
    if (button_1_state == FOR_PRESS && PORTAbits.RA3 == 0) {
      // we are checking ``PRESS'' and RA3 has been pressed
      LATBbits.LATB0 = !LATBbits.LATB0; // toggle the led
      button_1_state = FOR_RELEASE;
    }
    else if (button_1_state == FOR_RELEASE && PORTAbits.RA3 == 1) {
      // we are checking ``RELEASE'' and RA3 has been released
      button_1_state = FOR_PRESS;
    }
    ...
  }
}
```

Let's organise the code better

```
#define FOR_PRESS   0
#define FOR_RELEASE 1

int button_1_state = FOR_PRESS;

int is_button_1_pressed(void)
{
    if (button_1_state == FOR_PRESS && PORTAbits.RA3 == 0) {
      // we are checking ``PRESS'' and RA3 has been pressed
      button_1_state = FOR_RELEASE;
      return 1;
    }
    else if (button_1_state == FOR_RELEASE && PORTAbits.RA3 == 1) {
      // we are checking ``RELEASE'' and RA3 has been released
      button_1_state = FOR_PRESS;
    }
    return 0;
}

main()
{
  // ... set-up ports by TRIS registers

  for (;;) {  // loop forever
    if (is_button_1_pressed())
      LATBbits.LATB0 = !LATBbits.LATB0; // toggle the led
    ...
  }
}
```

No Busy-Waits with two buttons and flash

```
#define CHECK_FOR_PRESS ...
#define CHECK_FOR_RELEASE ...
main()
{
  long count = 0; // 32 bit is better!
  // ... set-up ports by TRIS registers
  for (;;) {  // loop forever

    if (is_button_1_pressed()) LATBbits.LATB0 = !LATBbits.LATB0;
    if (is_button_2_pressed()) LATBbits.LATB1 = !LATBbits.LATB1;
    if (is_button_3_pressed()) LATBbits.LATB2 = !LATBbits.LATB2;

    ++count;
    if (count == MAX) {
      LATBbits.LATB5 = !LATBbits.LATB5; // toggle the led
      count = 0;
    }

  }
}
```

The Finite-State Machine Abstraction

Systems which behave by responding to **events** which cause **actions** are usually programmed using the **finite-state machine (FSM)** abstraction/design pattern.

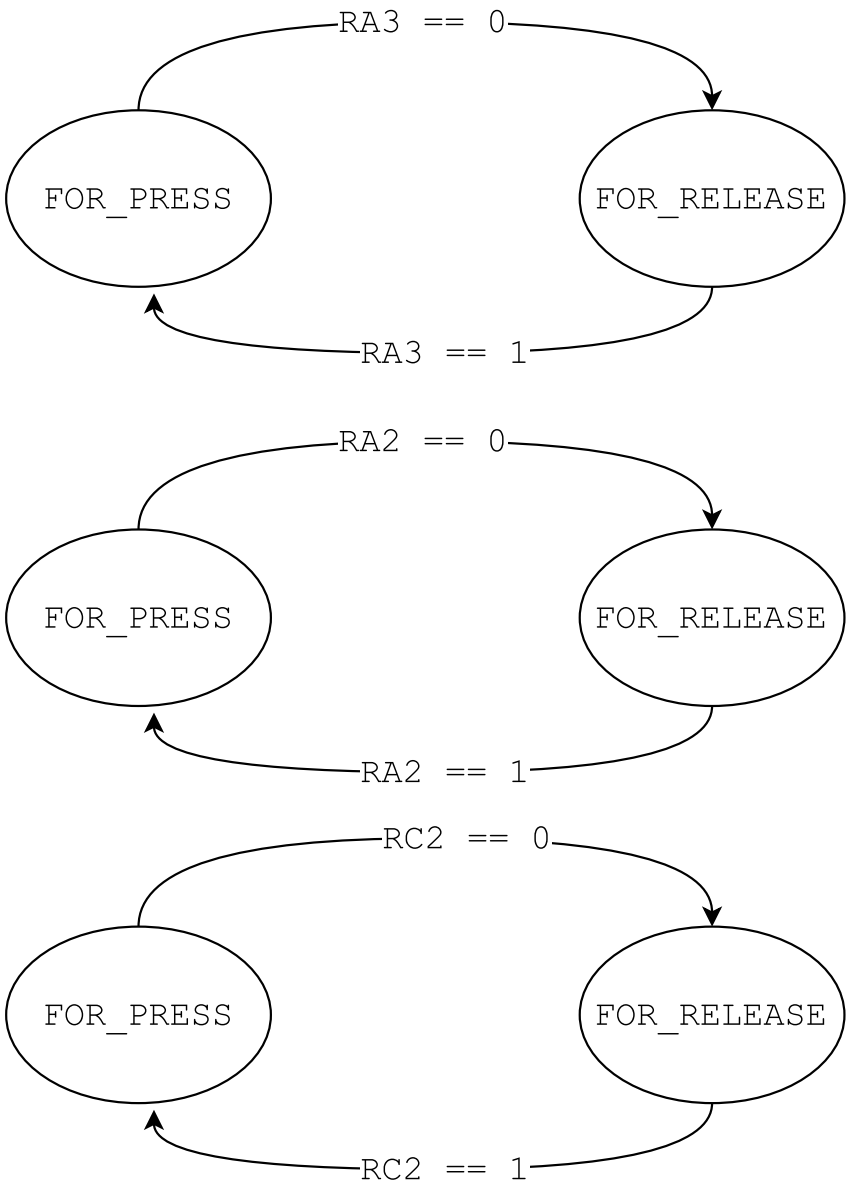
A finite-state machine can be formulated as

- A set of **states** S;
- A set of **events** E;
- A set of **action** A;
- A **function** $f(s, e) \rightarrow (a, s')$ which, given the **current state** $s \in S$ and the **event** occurred $e \in E$, gives the **action** $a \in A$ to be performed and the **new state** $s' \in S$.

A FSM is usually represented graphically by means of a directed graph in which:

- Vertices are **states**
- Edges are **state transitions**

The FSM of our “keyboard”



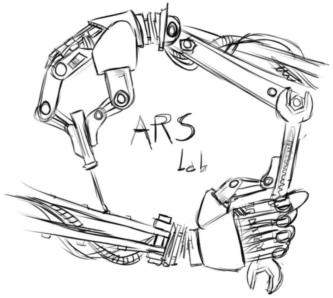
Exercise!!

- Toggle LED in RB0 each time the button in RA3 is pressed.
- Toggle LED in RB1 each time the button in RA2 is pressed.
- Cycle LEDs from RB2 to RB5 each time button RC2 is pressed.
- Turn off all leds after about 4 seconds of inactivity.

Handling Asynchronous Events in MCUs
The Finite-State Machine Design Pattern

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