Ch. 3 IoT Programmin

Section 2: Pull Resistors & Interrupt

COMPSCI 147

Internet-of-Things; Software and Systems

Example - Hooking up a pushbutton on Arduino

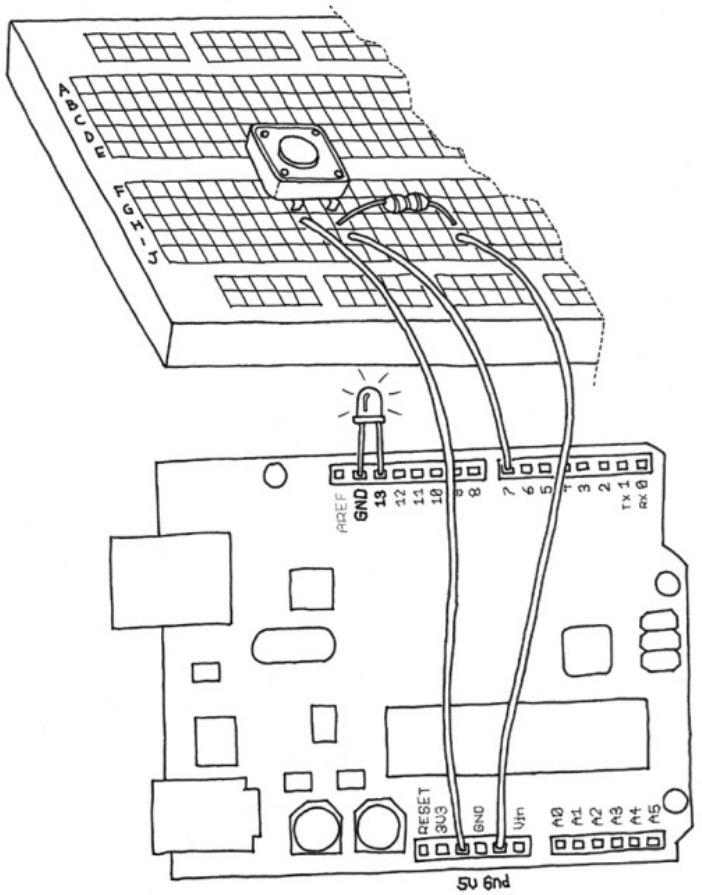
```
// Example 02: Turn on LED while the button is pressed

const int LED = 13;   // the pin for the LED
const int BUTTON = 7; // the input pin where the
                      // pushbutton is connected
int val = 0;          // val will be used to store the state
                      // of the input pin

void setup() {
  pinMode(LED, OUTPUT);   // tell Arduino LED is an output
  pinMode(BUTTON, INPUT); // and BUTTON is an input
}

void loop(){
  val = digitalRead(BUTTON); // read input value and store it

  // check whether the input is HIGH (button pressed)
  if (val == HIGH) {
    digitalWrite(LED, HIGH); // turn LED ON
  } else {
    digitalWrite(LED, LOW);
  }
}
```

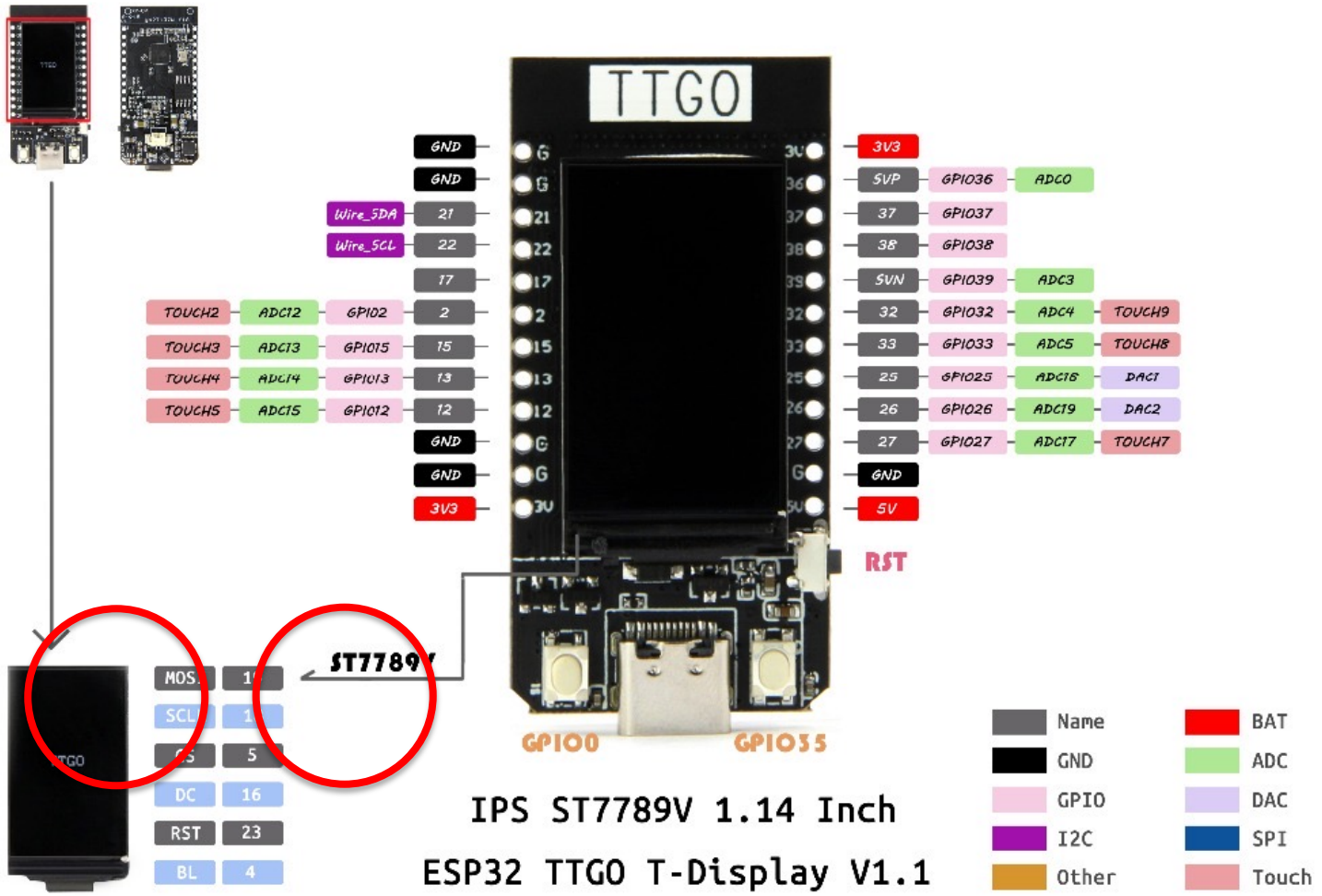


https://www.arduino.cc/en/tutorial/button

Buttons!

- Seems most simple, but has a lot of nuances !

- TTGO has 2 onboard Buttons



Pull up / Pull down

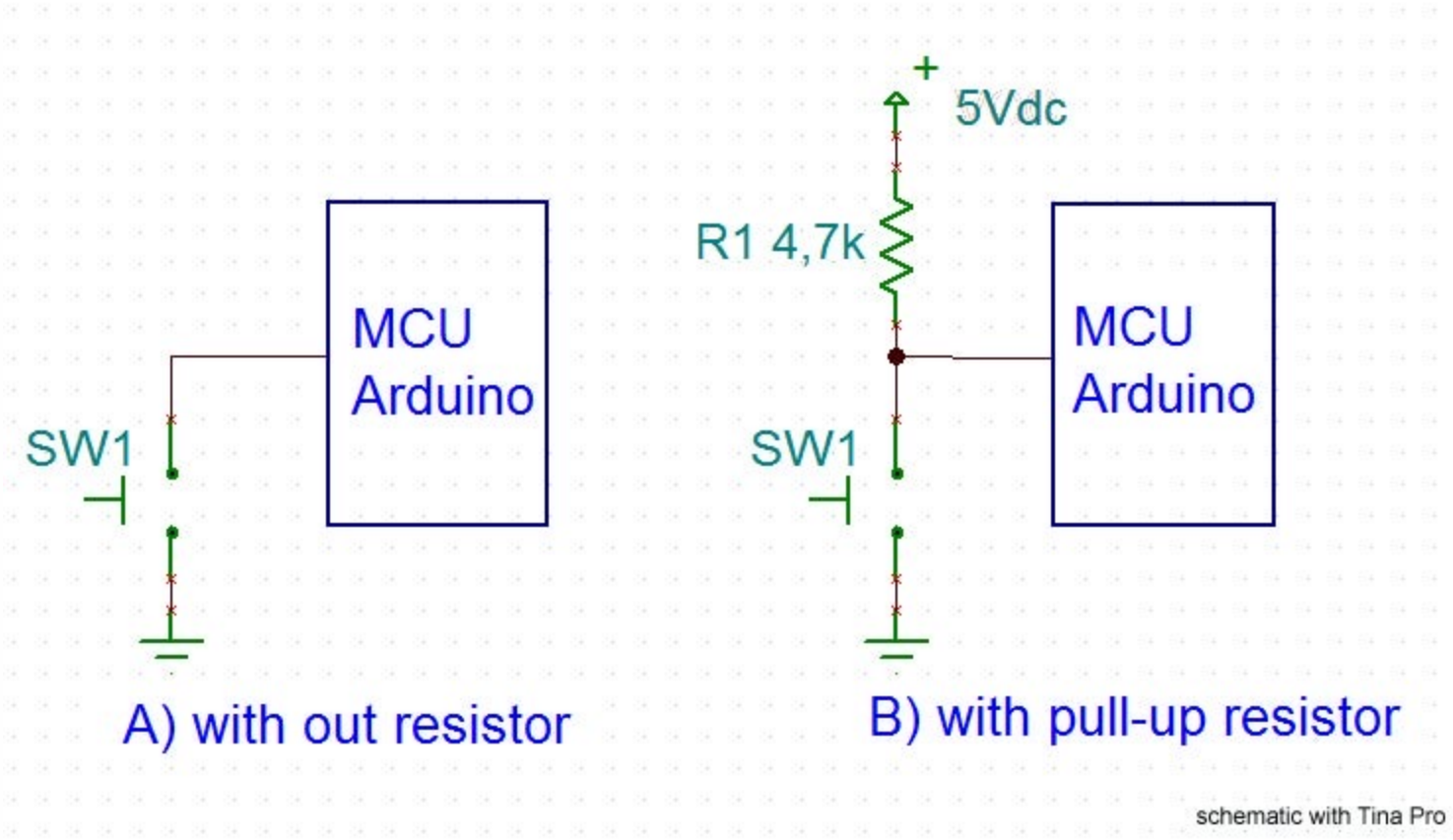
- Open switch:
  Floating value so it could be either a '0' or a '1".

- To prevent unknown state, “default value” should be assigned to buttons

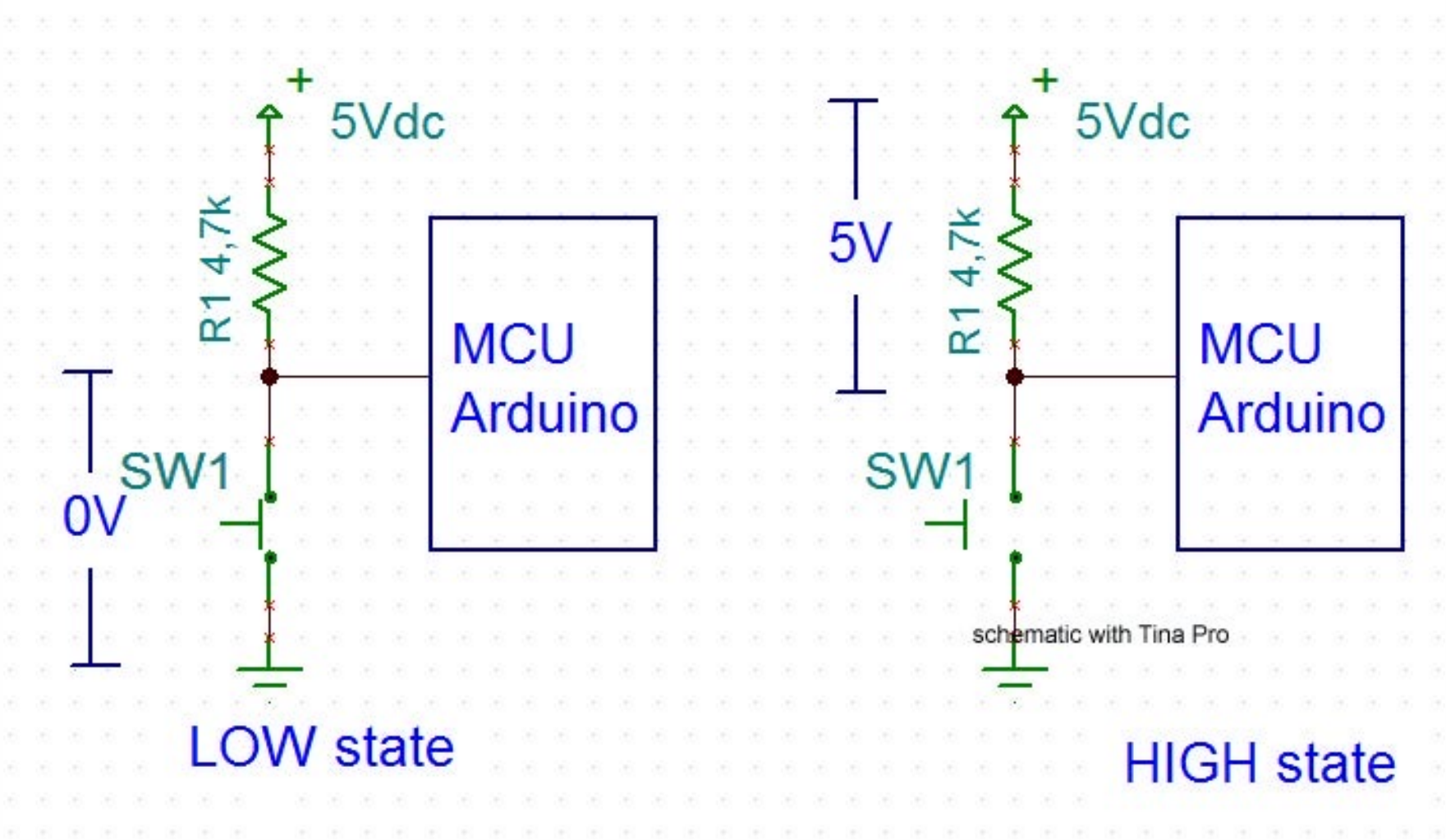
- This is done with a small circuit (resistor)

- Pull up - default high
- Pull down – default low

Pull up (Default: High)



Pull up (Default: High)



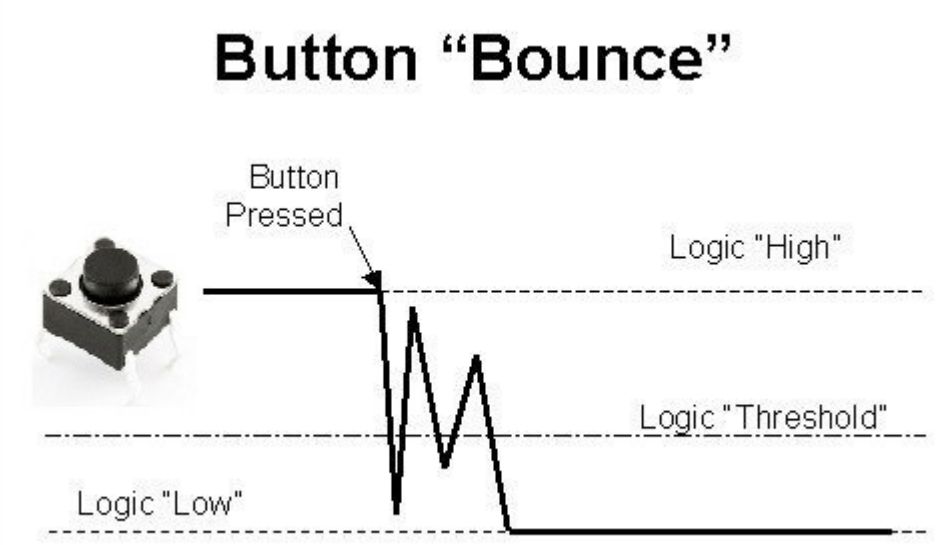
How to use pull ups in code

```
pinMode(inPin, INPUT_PULLUP);
pinMode(inPin, INPUT_PULLDOWN);
```

Debouncing

- When you press (or release) a button it will often change level a couple of times before it settles at the new level.
- This behavior is not wanted.
- A software technique to handle this is to check (poll) the button(s) periodically and only decide that a button is pressed if it has been in the pressed state for a couple of subsequent polls.

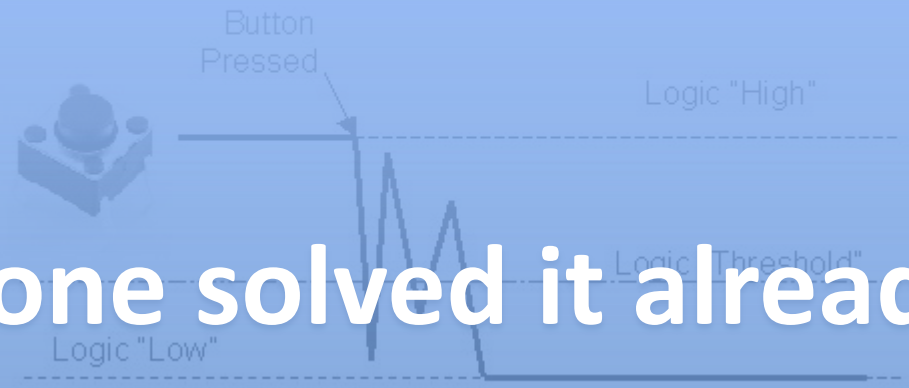
What about double clicks ??
What about triple clicks ??



Debouncing

- When you press (or release) a button it will often change level a couple of times before it settles at the new level.
- This behavior is not wanted.
- A software technique to handle this is to check (poll) the button(s) periodically and only decide that a button is pressed if it has been in the pressed state for a couple of subsequent polls.

What about double clicks ??
What about triple clicks ??



Wouldn't it be nice..?

If someone solved it already?

Buttons require careful coding!

- https://github.com/LennartHennigs/Button2

- Arduino Library to simplify working with buttons.

- It allows you to use callback functions to track single, double, triple and long clicks.

- **It also takes care of debouncing**.

Polling vs interrupts

- During polling, software must repeatedly check and frequently check the state of the button.
- Even if most of the time it is exactly the same as before !
- There is a more efficient way to get notified when state changes: **interrupts**.
- They are taken care of with hardware flags at the low-level, and frees up resources for doing more interesting things (unlike checking button state).

Interrupt

- With an ESP32 board, all the GPIO pins can be configured to function as interrupt request inputs. (compared to only pins # 2 and 3 on Arduino Uno)
  - Used to 'interrupt' the regular process to do some higher priority action on a specific event.

```
void attachInterrupt(uint8_t pin, std::function<void(void)> intRoutine, int mode)
```

- `pin`: Sets the GPIO pin as an interrupt pin, which tells the ESP32 which pin to monitor.
- `intRoutine`: Is the name of the function that will be called every time the interrupt is triggered..
- `mode`: defines when the interrupt should be triggered.
  - LOW whenever pin state is low
  - HIGH whenever pin state is low
  - CHANGE whenever pin changes value
  - RISING whenever pin goes from low to high
  - FALLING whenever pin goes from high to low

    * Don't forget to capitalize.

Interrupt Service Routine

- `void detachInterrupt(uint8_t pin);`

You can optionally call detachInterrupt() function when you no longer want ESP32 to monitor a pin.

- **Interrupt Service Routine** is invoked when an interrupt occurs on any GPIO pin. Its syntax looks like below.

```
void IRAM_ATTR ISR() {
    Statements;
}
```

– Keep ISR SMALL!

What is IRAM_ATTR?

By flagging a piece of code with the `IRAM_ATTR` attribute we are declaring that the compiled code will be placed in the Internal RAM (IRAM) of the ESP32.

Otherwise the code is placed in the Flash. And flash on the ESP32 is much slower than internal RAM.

If the code we want to run is an interrupt service routine (ISR), we generally want to execute it as quickly as possible. If we had to **'wait'** for an ISR to load from flash, things would go horribly wrong.