MCU

Interrupts and PWM

{Learn, Create, Innovate};

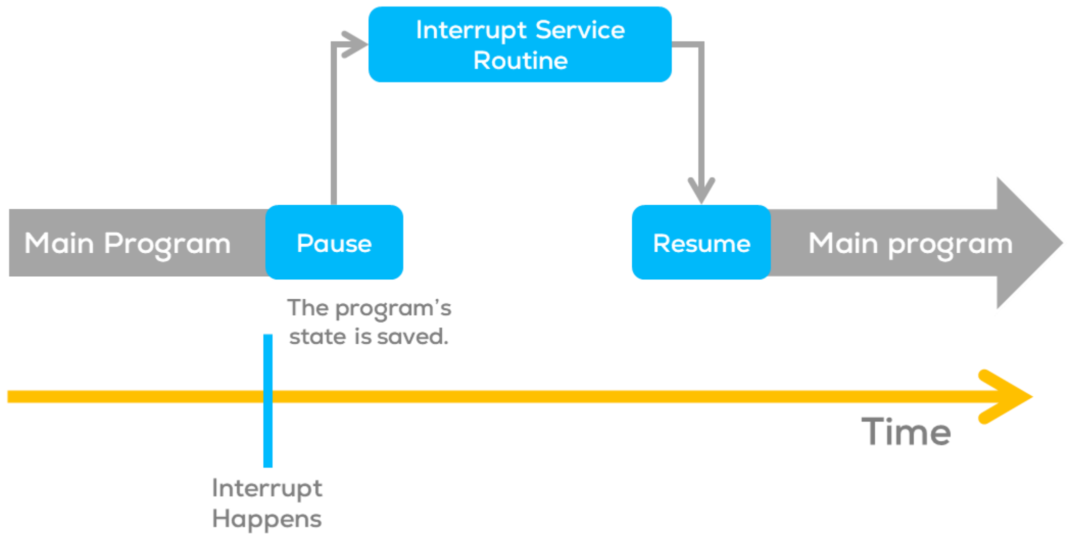
Interrupts

Hackerboard / ESP32

Interrupts

Interrupts

- Interrupts are signals that temporarily pause the main program to handle specific events.
- Such signals can come from external hardware or internal processes to the MCU.
- These signals are typically called triggers.
- Interrupts allow the microcontroller to respond immediately when an event occurs.



Interrupts

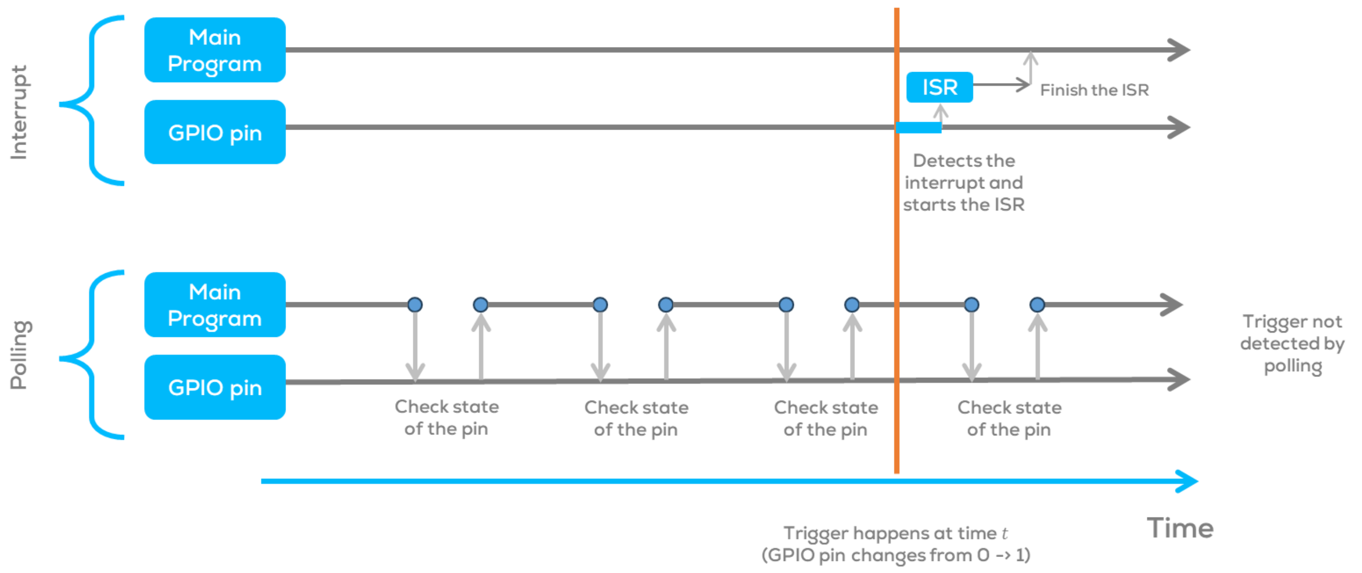
Advantages of Interrupts

- Multitasking capability: Handle multiple tasks simultaneously.
- Faster response time: Immediate action upon event detection.
- Lower power consumption: The microcontroller can sleep between interrupts.
- Precise timing: Ideal for time-critical tasks.
- Simplified code: No need for constant event checking.
- Compared with polling, where the user must continuously monitor the state of a variable or pin, interrupts respond to an event upon detection of the trigger signal.

Uses

- Interrupts are useful when solving timing problems.
- Some application examples may include reading a rotary encoder or monitoring user input (button, etc.).

Polling vs. Interrupt routines when checking a GPIO



Interrupts

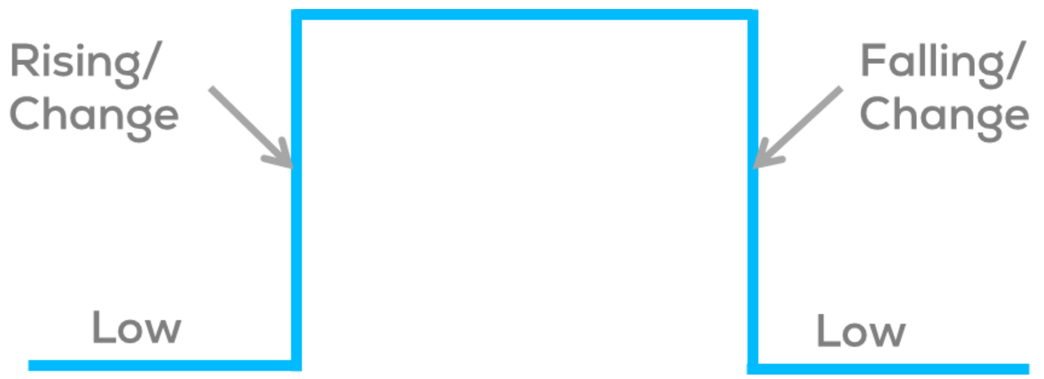
Types of interrupts

- **External Interrupts**
  - Triggered by external events, falling or rising edges; such as button presses or changes in sensor signals.
  - Ideal for real-time event-driven applications such as button presses, sensor readings, and rotary encoders.
- **Timer Interrupts**
  - Generated by internal timers at specific intervals.
  - Useful for precise timing, generating PWM signals, real-time clock, controlling stepper motors, and time-dependent tasks.
- **Pin Change Interrupts**
  - Triggered when the logic level changes on certain pins.
  - Suitable for scenarios with multiple pins changing simultaneously, like keypad inputs.
- **Special Interrupts**
  - Reset and watchdog timer interrupts.
  - Essential for system recovery and managing system stability.

Interrupts

Interrupt Modes (Arduino and ESP32)

- **LOW** to trigger the interrupt whenever the pin is low,
- **CHANGE** to trigger the interrupt whenever the pin changes value
- **RISING** to trigger when the pin goes from low to high,
- **FALLING** for when the pin goes from high to low



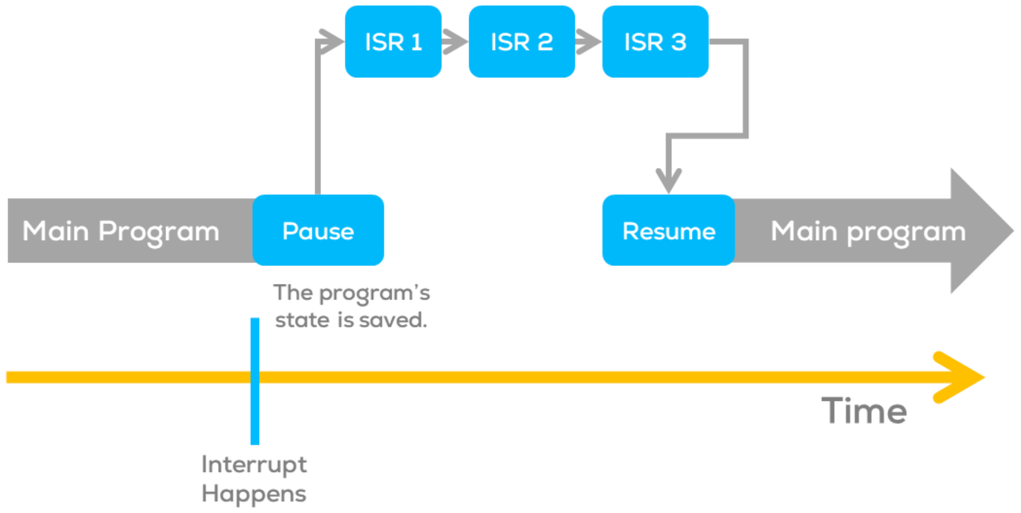
Interrupts

ISR (Interrupt Service Routines)

- Special functions that handle interrupts, allowing the microcontroller to respond immediately to events.
- ISR cannot have any parameters and shouldn't return anything.
- When an interrupt occurs, the microcontroller pauses the main program and jumps to the corresponding ISR.
- ISR executes, performs the necessary actions, and then returns to the main loop.

Warning:

- In Arduino, the functions *millis()*, *delay()* and *micros()* rely on interrupts to count, so they will never increment inside an ISR.
- When multiple ISRs are used, only one can run at a time. After the current one finishes, the rest of the interrupts will be executed in an order that depends on their priority. More information about priority can be found here.



Interrupts

Flags

- Interrupts use flags for different purposes; in Arduino and ESP the most common flags used are
  - **interrupts():** Re-enables interrupts (after they've been disabled.
    - Interrupts allow specific tasks to happen in the background and are enabled by default.
  - **nointerrupts():** Disables interrupts (you can re-enable them with interrupts()).
    - Some functions will not work while interrupts are disabled, and incoming communication may be ignored.
    - Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical code sections.

Variables

- Typically, global variables are used to pass data between an ISR and the main program.
- The variables used in an ISR must be declared as *volatile.*
  - Declaring variables as volatile, tells the compiler that such variables might change at any time, and thus the compiler must reload the variable whenever you reference it, instead of copying the value.

    volatile int var;

    volatile boolean flag;

ESP32 Interrupts

Configuring & Handling ESP32 GPIO Interrupts In Arduino IDE

- The following procedure mustpin-by-pined to configure Interrupts on the ESP32.

1. Attach Intterupt to a pin

```
attachInterrupt(GPIOPin, ISR, Mode);
```

- attachInterrupt() set an interrupt on a pin.
- GPIOPin –GPIO pin for the external interrupt.
- ISR – is the name of the Interrupt service routine (callback function) it can be any name as long as the function is called the same.
- Mode – defines interrupt mode: LOW,CHANGE, RISING,FALLING.

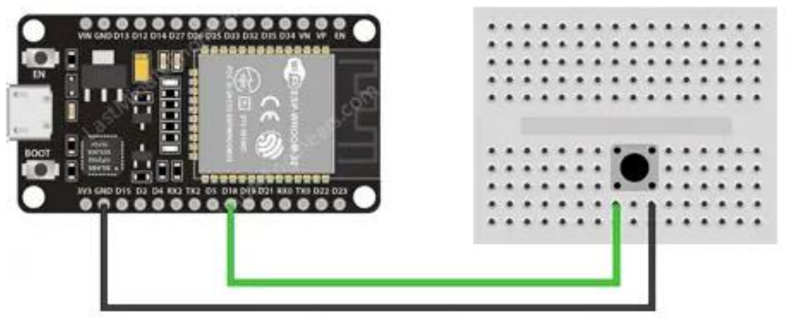
2. Set up an ISR (Interrupt service routine)

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

- Change the "ISR" to another name as long as it's the same as when declared on "attachInterrupt()" function.
- An ISR cannot have any parameters, and they should not return anything.
- ISRs should be as short and fast as possible as they block normal program execution.
- They should have the IRAM_ATTR attribute, according to the ESP32 documentation.

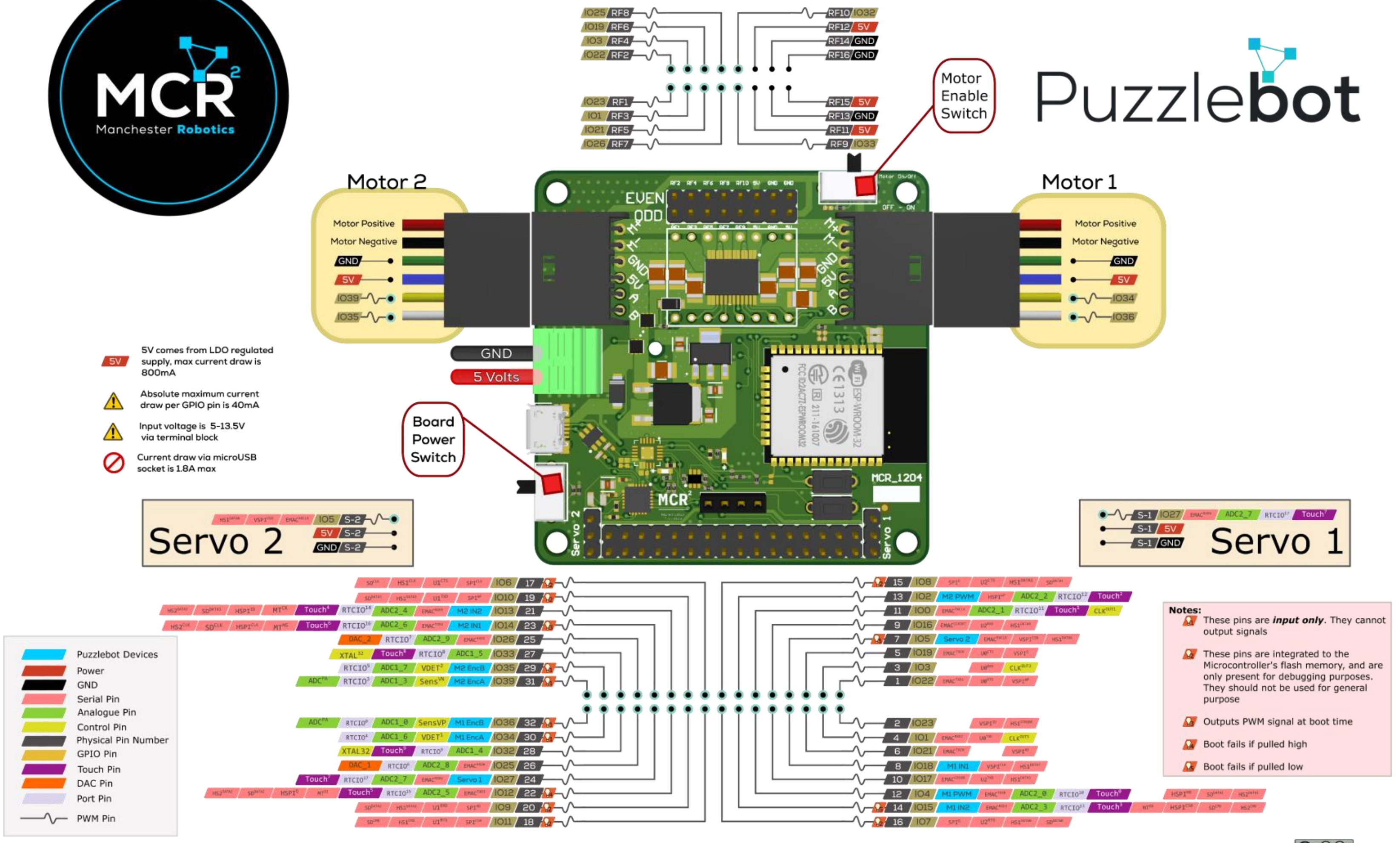
Interrupts: Example

```
struct Button {
  const uint8_t PIN;
  uint32_t numberKeyPresses;
  bool pressed;
};

Button button1 = {18, 0, false};

void IRAM_ATTR isr() {
  button1.numberKeyPresses++;
  button1.pressed = true;
}

void setup() {
  Serial.begin(115200);
  pinMode(button1.PIN, INPUT_PULLUP);
  attachInterrupt(button1.PIN, isr, FALLING);
}

void loop() {
  if (button1.pressed) {
    Serial.printf("Button has been pressed %u times\n",
button1.numberKeyPresses);
    button1.pressed = false;
  }
}
```



- This example expects an external interrupt given by a switch connected to GPIO 18 of the ESP32.
- Normally, you should use digitalPinToInterrupt(pin), rather than place an interrupt number directly into your sketch.

Hackerboard Pinout



5V comes from LDO regulated supply, max current draw is 800mA

⚠ Absolute maximum current draw per GPIO pin is 40mA

⚠ Input voltage is 5-13.5V via terminal block

🚫 Current draw via microUSB socket is 1.8A max

🟦	Puzzlebot Devices
🟥	Power
⬛	GND
🟥	Serial Pin
🟩	Analogue Pin
🟨	Control Pin
⬛	Physical Pin Number
🟨	GPIO Pin
🟪	Touch Pin
🟧	DAC Pin
⬜	Port Pin
─∿─	PWM Pin

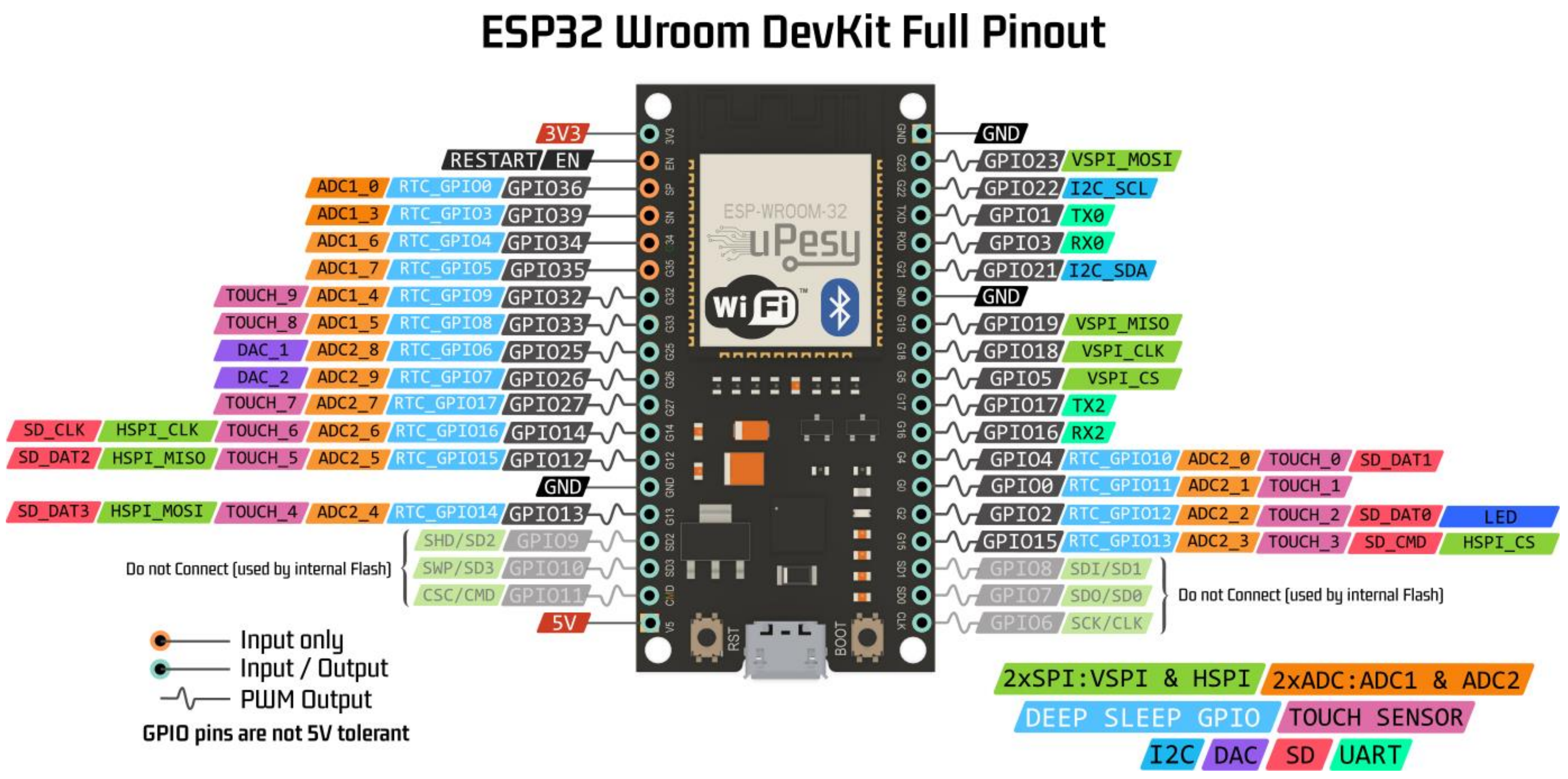
Notes:

1. These pins are ***input only***. They cannot output signals
2. These pins are integrated to the Microcontroller's flash memory, and are only present for debugging purposes. They should not be used for general purpose
3. Outputs PWM signal at boot time
4. Boot fails if pulled high
5. Boot fails if pulled low

01 FEB 2022
v1

https://www.manchester-robotics.com

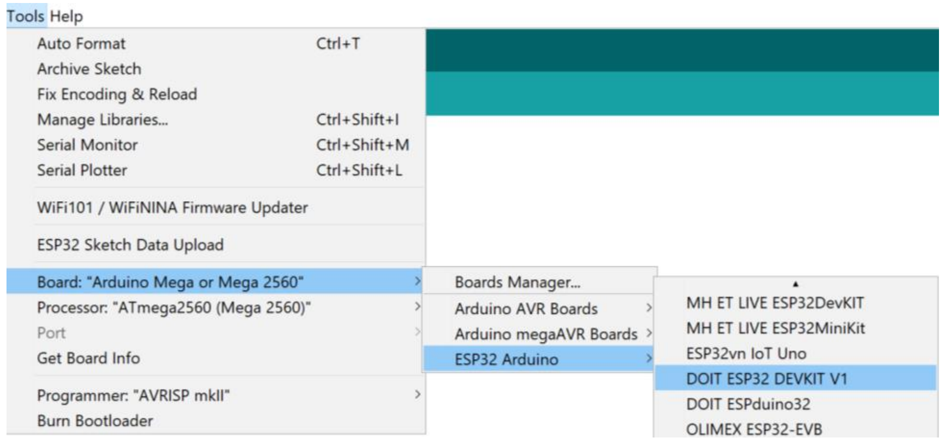
ESP32 Pinout


ESP32 Wroom DevKit Full Pinout
3V3
RESTART / EN
ADC1_0 RTC_GPIO0 GPIO36
ADC1_3 RTC_GPIO3 GPIO39
ADC1_6 RTC_GPIO4 GPIO34
ADC1_7 RTC_GPIO5 GPIO35
TOUCH_9 ADC1_4 RTC_GPIO9 GPIO32
TOUCH_8 ADC1_5 RTC_GPIO8 GPIO33
DAC_1 ADC2_8 RTC_GPIO6 GPIO25
DAC_2 ADC2_9 RTC_GPIO7 GPIO26
TOUCH_7 ADC2_7 RTC_GPIO17 GPIO27
SD_CLK HSPI_CLK TOUCH_6 ADC2_6 RTC_GPIO16 GPIO14
SD_DAT2 HSPI_MISO TOUCH_5 ADC2_5 RTC_GPIO15 GPIO12
GND
SD_DAT3 HSPI_MOSI TOUCH_4 ADC2_4 RTC_GPIO14 GPIO13
SHD/SD2 GPIO9
SWP/SD3 GPIO10
CSC/CMD GPIO11
Do not Connect (used by internal Flash)
5V
GND
GPIO23 VSPI_MOSI
GPIO22 I2C_SCL
GPIO1 TX0
GPIO3 RX0
GPIO21 I2C_SDA
GND
GPIO19 VSPI_MISO
GPIO18 VSPI_CLK
GPIO5 VSPI_CS
GPIO17 TX2
GPIO16 RX2
GPIO4 RTC_GPIO10 ADC2_0 TOUCH_0 SD_DAT1
GPIO0 RTC_GPIO11 ADC2_1 TOUCH_1
GPIO2 RTC_GPIO12 ADC2_2 TOUCH_2 SD_DAT0 LED
GPIO15 RTC_GPIO13 ADC2_3 TOUCH_3 SD_CMD HSPI_CS
GPIO8 SDI/SD1
GPIO7 SDO/SD0
GPIO6 SCK/CLK
Do not Connect (used by internal Flash)
3V3 EN SP SN 34 G35 G32 G33 G25 G26 G27 G14 G12 GND G13 SD2 SD3 CMD V5
GND G23 G22 TXD RXD G21 GND G19 G18 G5 G17 G16 G4 G0 G2 G15 SD1 SD0 CLK
ESP-WROOM-32
uPesy
WiFi
RST
BOOT
Input only
Input / Output
PWM Output
GPIO pins are not 5V tolerant
2xSPI:VSPI & HSPI
2xADC:ADC1 & ADC2
DEEP SLEEP GPIO
TOUCH SENSOR
I2C
DAC
SD
UART

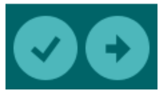
Activity

Uploading (Arduino IDE)

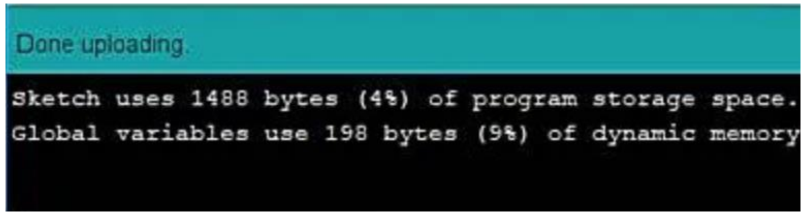
- Connect the Hackerboard or the ESP32 board
- Select the port to be used Tools›Port
- Select the board to be used Tools -> ESP32 Arduino › DOIT ESP32 DEVKIT V1 (for Hackerboard is the same)



- Upload the code using the arrow on the top left corner of the IDE.



- The following message should be displayed:



```
Done uploading.
Sketch uses 1488 bytes (4%) of program storage space.
Global variables use 198 bytes (9%) of dynamic memory
```

- Open the terminal

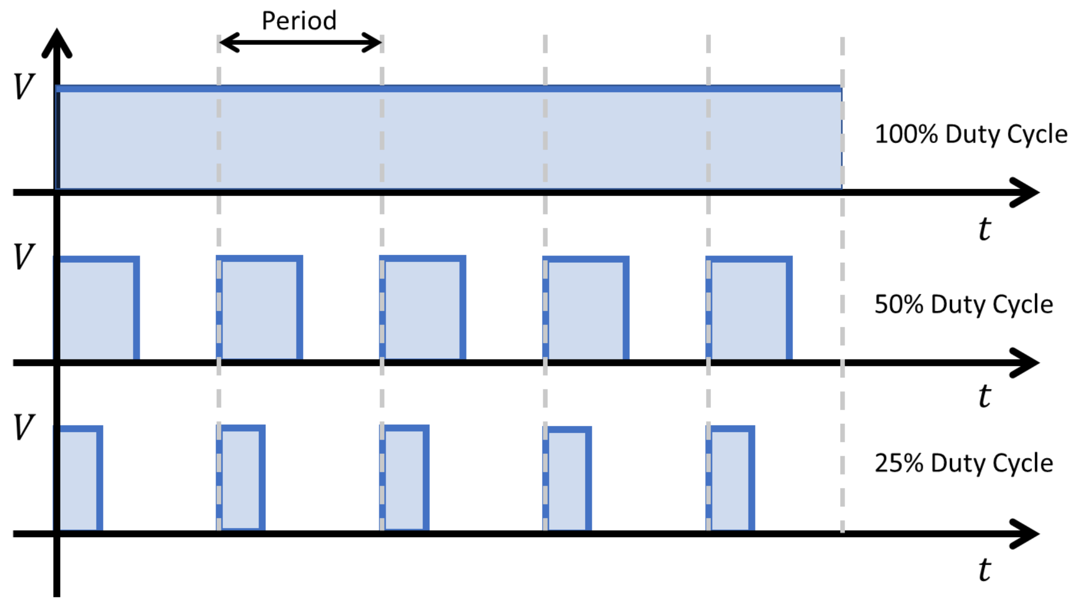
PWM Signals

Hackerboard/ESP32

PWM

What is PWM?

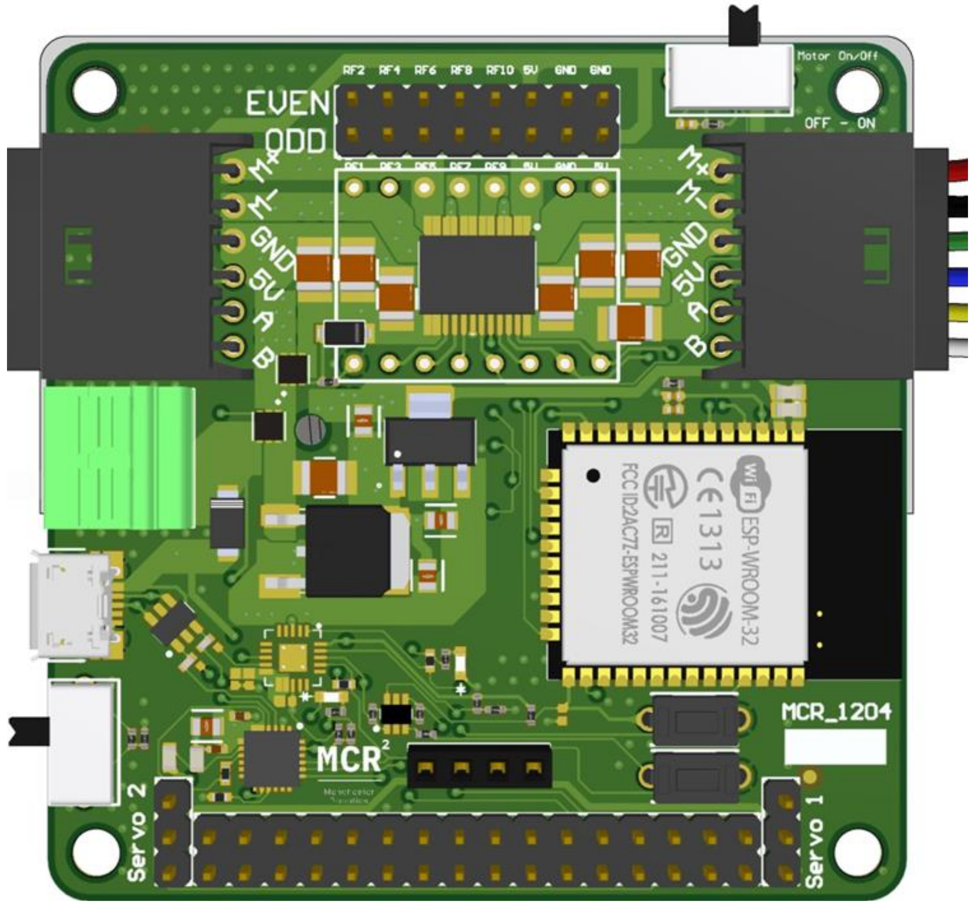
- PWM stands for Pulse Width Modulation.
- It's a technique used in electronics to control the average voltage or current delivered to a device by rapidly switching between full power and no power (On/Off) over a fixed period of time.
- This creates an average voltage or current somewhere in between, effectively controlling the output (power delivered to the system).
- The duration of "on time" is called the pulse width.
- To get varying analog voltage values, the pulse width can be changed "modulated".



PWM

Why Use PWM on ESP32?

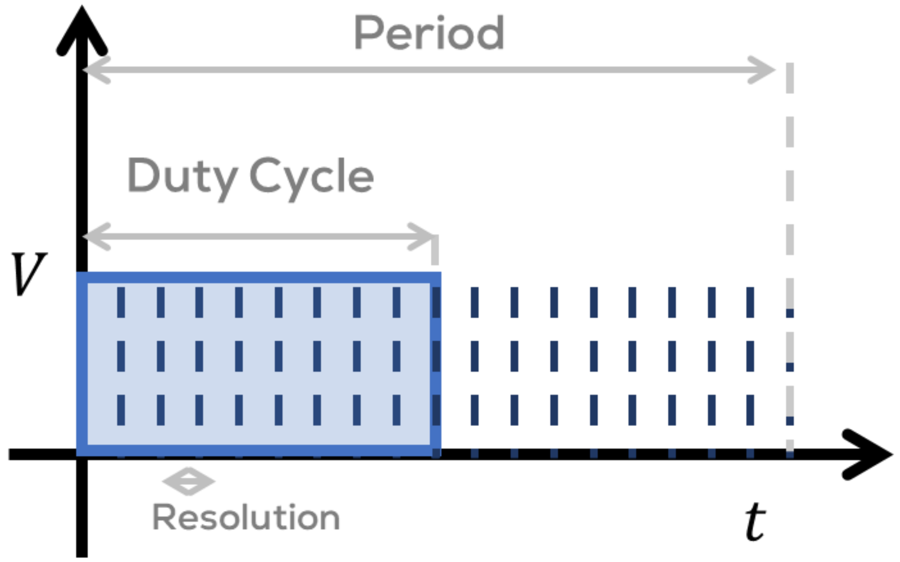
- PWM is commonly used to control things like motor speed, LED brightness, and even audio signals.
- It's an efficient way to simulate varying levels of output using digital control, like from a microcontroller such as the ESP32.
- The ESP32 has a dedicated LEDC (LED Controller) peripheral for PWM.
- It supports up to 16 PWM channels with configurable frequencies and resolutions.
- Unlike Arduino, ESP32 allows more precise PWM control.



ESP32 PWM

ESP32 PWM

- The ESP32 LEDC library provides hardware-accelerated PWM.
- You can configure for each PWM signal:
  - Frequency (Hz)
  - Resolution (bits, up to 16-bit precision)
  - Duty cycle (0 - max resolution value)
- PWM resolution is the number of distinct levels or steps that a PWM signal can have within its duty cycle range. In other words, is the granularity with which the duty cycle can be modulated.
- Each PWM signal on the ESP32 is assigned to a channel. The channel is used to configure parameters previously mentioned.
- To generate a PWM signal on a specific pin, you "attach" that pin to a channel.
- This tells the ESP32 to output the PWM waveform generated by the channel on the specified pin. Multiple pins can be attached to the same channel, which means they can all output the same PWM signal.



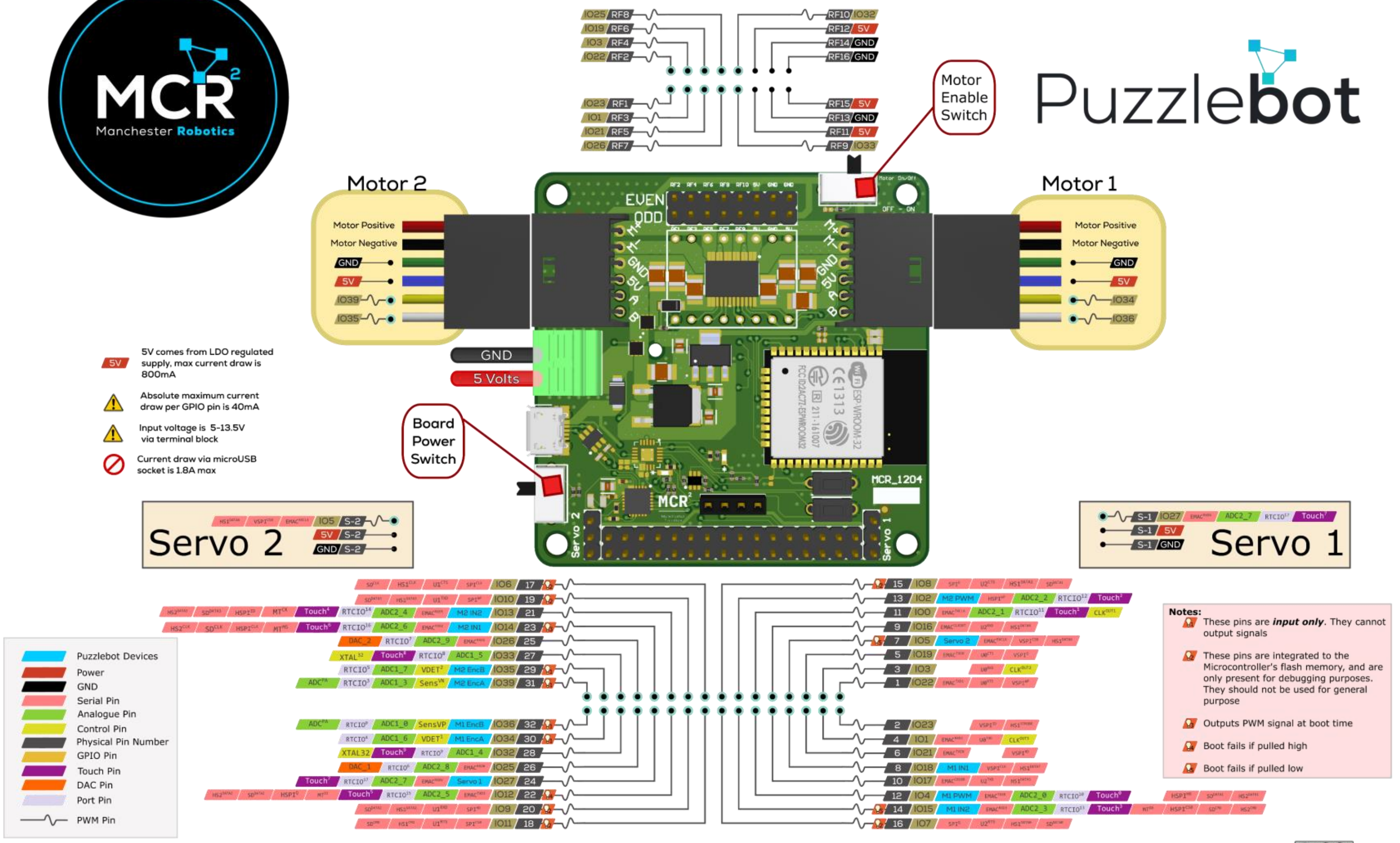
ESP32 PWM

Setting Up a PWM Channel

- To use PWM with LEDC on ESP32, follow these steps:
    1. Select a channel (0 to 15)
    2. Set the frequency (e.g., 5000 Hz for LED dimming or motor control)
    3. Set the resolution (typically 8-bit, 10-bit, or 12-bit)
    4. Attach a GPIO pin or pins to the channel
    5. Write duty cycle values (0 to max resolution value)

```
const int PWM_CHANNEL = 0;    // ESP32 has 16 channels which can generate 16 independent waveforms
const int PWM_FREQ = 500;     // Recall that Arduino Uno is ~490 Hz. Official ESP32 example uses 5,000Hz
const int PWM_RESOLUTION = 8; // We'll use same resolution as Uno (8 bits, 0-255) but ESP32 can go up to 16 bits
// The max duty cycle value based on PWM resolution (will be 255 if resolution is 8 bits)
const int MAX_DUTY_CYCLE = (int)(pow(2, PWM_RESOLUTION) - 1);
const int LED_OUTPUT_PIN = 22;
const int DELAY_MS = 4;  // delay between fade increments

void setup() {
// Sets up a channel (0-15), a PWM duty cycle frequency, and a PWM resolution (1 - 16 bits)
  // ledcSetup(uint8_t channel, double freq, uint8_t resolution_bits);
  ledcSetup(PWM_CHANNEL, PWM_FREQ, PWM_RESOLUTION);

  // ledcAttachPin(uint8_t pin, uint8_t channel);
  ledcAttachPin(LED_OUTPUT_PIN, PWM_CHANNEL);
}
void loop() {
  // fade up PWM on given channel
  for(int dutyCycle = 0; dutyCycle <= MAX_DUTY_CYCLE; dutyCycle++){
    ledcWrite(PWM_CHANNEL, dutyCycle);
    delay(DELAY_MS);
  }
  // fade down PWM on given channel
  for(int dutyCycle = MAX_DUTY_CYCLE; dutyCycle >= 0; dutyCycle--){
    ledcWrite(PWM_CHANNEL, dutyCycle);
    delay(DELAY_MS);
  }
}
```

Hackerboard Pinout



5V	5V comes from LDO regulated supply, max current draw is 800mA
⚠	Absolute maximum current draw per GPIO pin is 40mA
⚠	Input voltage is 5-13.5V via terminal block
🛇	Current draw via microUSB socket is 1.8A max

🟦	Puzzlebot Devices
🟥	Power
⬛	GND
🟥	Serial Pin
🟩	Analogue Pin
🟨	Control Pin
⬛	Physical Pin Number
🟨	GPIO Pin
🟪	Touch Pin
🟧	DAC Pin
⬜	Port Pin
─∿─	PWM Pin

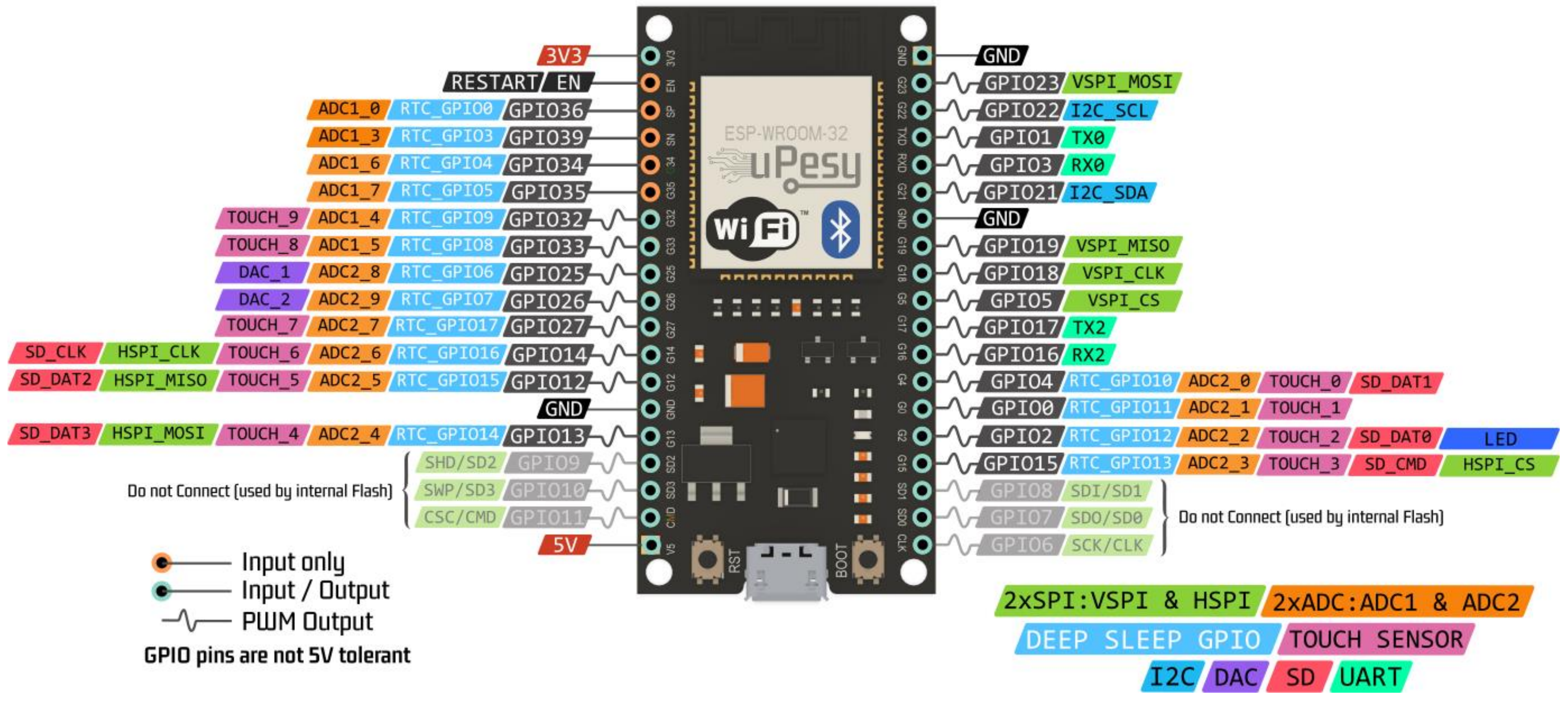
Notes:
- 1 These pins are ***input only***. They cannot output signals
- 2 These pins are integrated to the Microcontroller's flash memory, and are only present for debugging purposes. They should not be used for general purpose
- 3 Outputs PWM signal at boot time
- 4 Boot fails if pulled high
- 5 Boot fails if pulled low

01 FEB 2022
v1

https://www.manchester-robotics.com

ESP32 Pinout

ESP32 Wroom DevKit Full Pinout


3V3
RESTART / EN
ADC1_0 RTC_GPIO0 GPIO36
ADC1_3 RTC_GPIO3 GPIO39
ADC1_6 RTC_GPIO4 GPIO34
ADC1_7 RTC_GPIO5 GPIO35
TOUCH_9 ADC1_4 RTC_GPIO9 GPIO32
TOUCH_8 ADC1_5 RTC_GPIO8 GPIO33
DAC_1 ADC2_8 RTC_GPIO6 GPIO25
DAC_2 ADC2_9 RTC_GPIO7 GPIO26
TOUCH_7 ADC2_7 RTC_GPIO17 GPIO27
SD_CLK HSPI_CLK TOUCH_6 ADC2_6 RTC_GPIO16 GPIO14
SD_DAT2 HSPI_MISO TOUCH_5 ADC2_5 RTC_GPIO15 GPIO12
GND
SD_DAT3 HSPI_MOSI TOUCH_4 ADC2_4 RTC_GPIO14 GPIO13
SHD/SD2 GPIO9
SWP/SD3 GPIO10
CSC/CMD GPIO11
Do not Connect (used by internal Flash)
5V
GND
GPIO23 VSPI_MOSI
GPIO22 I2C_SCL
GPIO1 TX0
GPIO3 RX0
GPIO21 I2C_SDA
GND
GPIO19 VSPI_MISO
GPIO18 VSPI_CLK
GPIO5 VSPI_CS
GPIO17 TX2
GPIO16 RX2
GPIO4 RTC_GPIO10 ADC2_0 TOUCH_0 SD_DAT1
GPIO0 RTC_GPIO11 ADC2_1 TOUCH_1
GPIO2 RTC_GPIO12 ADC2_2 TOUCH_2 SD_DAT0 LED
GPIO15 RTC_GPIO13 ADC2_3 TOUCH_3 SD_CMD HSPI_CS
GPIO8 SDI/SD1
GPIO7 SDO/SD0
GPIO6 SCK/CLK
Do not Connect (used by internal Flash)
3V3 EN SP SN 34 G35 G32 G33 G25 G26 G27 G14 G12 GND G13 SD2 SD3 CMD V5
GND G23 G22 TXD RXD G21 GND G19 G18 G5 G17 G16 G4 G0 G2 G15 SD1 SD0 CLK
ESP-WROOM-32
uPesy
WiFi
RST
BOOT
Input only
Input / Output
PWM Output
GPIO pins are not 5V tolerant
2xSPI:VSPI & HSPI
2xADC:ADC1 & ADC2
DEEP SLEEP GPIO
TOUCH SENSOR
I2C
DAC
SD
UART