Embedded Systems CSEN701

Dr. Catherine Elias

Eng. Abdalla Mohamed

Office: C1.213
Mail : abdalla.abdalla@guc.edu.eg

Eng. Youssef Abdelshafy

Office: C2.110
Mail : youssef.abdelshafy@guc.edu.eg

Eng. Mina Wasfy

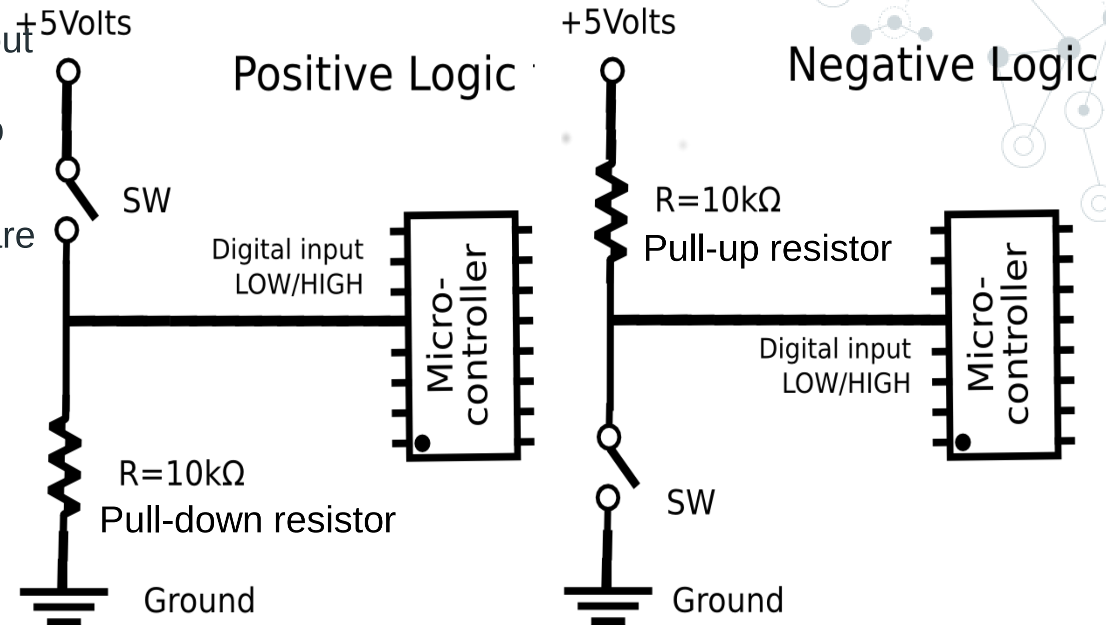
Office: C4.229
Mail : mina.wasfy@guc.edu.eg

Outline :

- **Recap.**
- Sensors
- Infra-red Implementation
- ADC
- Potentiometer ADC Implementation
- Why Drivers ?
- Modular Infra-red Driver

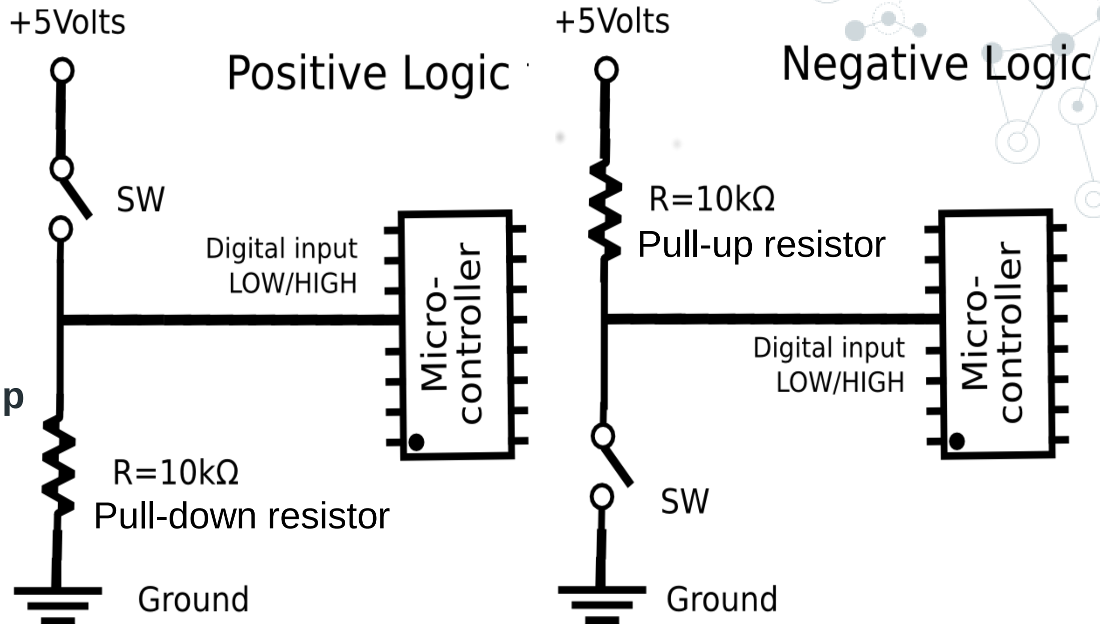
First we have to understand different digital logics …

- Positive logic is the **default logic** , the input is initially low until the button is ON /activated so it deliver High voltage (5v) to the Microcontroller/LED when pressed.
- Sensors/Pins working with positive logic are called Active-HIGH.
- Pull-down resistors are associated with positive logic to **pull** the initial pin value **down** to GND thus preventing the floating of the input value.
- Button Is OFF/open -- Input = GND (0V)
- Button is ON/Closed -- Input = VCC (5V)



First we have to understand different digital logics …

- Negative Logic connection operates in an opposite manner, the input is initially high until the button is On/activated it delivers GND (0V) to the Microcontroller/LED.
- Sensors/Pins working with negative logic are called Active-Low.
- Pull-up resistors are associated with negative logic to **pull** the initial pin value **up** to High voltage (5V) thus preventing the floating of the input value .
- Button Is OFF/open -- Input = VCC (5V)
- Button is ON/Closed -- Input = GND (0V)



EX1

Implement an embedded C code to :

1. Connect push button A to pin 5 in PORT C (Positive Logic)
2. Connect push button B to pin 3 in PORT B (Negative Logic)
3. Apply the Internal pullup resistor to pin3 PORT B
4. Configure PIN 2 in PORTD as output
5. Connect Pin 2 to RED LED Pin5-- RED (Hardware step)
6. Turn on Red LED when A is pressed
7. Turn off the RED LED when B is pressed .

```
#include <avr/io.h>
int main (void){
 DDRB = 0x00 ; DDRD = 0x00 ; DDRC = 0x00 ; PORTC = 0x00 ; PORTB=0x00; PORTD=0x00 ;    // initialize the registers
 DDRC &=~(1<<5)  ;  // configure pin5 as input  in PORTA ( pushbutton A is connected to PIN 5 in positive Logic )
 DDRB &=~(1<<3)  ;  // configure pin3 as input in PORTB ( pushbutton B is connected to PIN 3 in negative Logic )
 PORTB |= (1<<3) ; // set bit 3 to HIGH to activate the internal pull-up resistor at pin 3
 DDRD   |= (1<<2) ; // configure pin 2 as an output pin  at PORTD
 while (1 ) {
 if ( PINC & ( 1<<5) ) {   //  HINT : (PINC & 0b00100000) is only true when bit 5 at PINC is 1 (pushbutton A is pressed +ve L)
 PORTD |= (1<<2) ;  // set the output to HIGH  to TURN ON the LED
 }
 if ( !(PINB & (1<<3) ) {   // (PINB & (1<<3)) is true when Bit 3 is ON ( not pressed ) so it will be false (!) if  pressed ( -ve L)
      PORTD &= ~(1<<2)  ; // set the output to LOW  by clearing bit 2        // pushbutton B is connected in negative logic
   } }                                          }
```

Note : Extra information Don't Memorize

The RP2040 GPIO Hardware Registers :

- There are 28 programmable GPIO pins on the Pico. There are 40 pins, but the others are ground, power and a couple of specialized pins .
- The registers shown in the diagram are used to control the input/output specifications in the DIO peripheral.
- DIO peripheral contains 32-bit hardware register which is mapped to 32-bits of memory in the RP2040's address space
- Check the datasheet for each register description.

Register	Address
gpio_in	0xd0000004
gpio_hi_in	0xd0000008
gpio_out	0xd0000010
gpio_set	0xd0000014
gpio_clr	0xd0000018
gpio_togl	0xd000001c
gpio_oe	0xd0000020
gpio_oe_set	0xd0000024
gpio_oe_clr	0xd0000028
gpio_togl	0xd000002c
gpio_hi_out	0xd0000030
gpio_hi_set	0xd0000034
gpio_hi_clr	0xd0000038
gpio_hi_togl	0xd000003c
gpio_hi_oe	0xd0000040
gpio_hi_oe_set	0xd0000044
gpio_hi_oe_clr	0xd0000048
gpio_hi_oe_togl	0xd000004c

Note : Extra information Don't Memorize

- **GPIO_OE**: Output enable register (1 for output, 0 for input). 32-bit register , each bit maps to a corresponding GPIO PIN resembling the **DDRX** .

- **GPIO_CTRL**: There is a 32-bit GPIO control register for each pin , separated by 8 bytes in the memory space , it is used to configure the function of the pins.

- **GPIO_OUT**: Output value register. Resembles **PORTX** in output pins, used to enter the output needed in the pin in its corresponding bit .

- **GPIO_IN**: Input value register. Used to read the Value of the input pin using its corresponding bit resembling PINX.

gpio.c

gpio.c > ...

Can be included in a header file GPIO.h so we can include it directly as #include "GPIO.h"

```c
#include <stdint.h>

#define GPIO_BASE       0x40014000  // Base address for GPIO registers
#define GPIO_OE         (*(volatile uint32_t *)(GPIO_BASE + 0x20))  // Output Enable Register
#define GPIO_OUT        (*(volatile uint32_t *)(GPIO_BASE + 0x10))  // Output Register
#define GPIO_IN         (*(volatile uint32_t *)(GPIO_BASE + 0x14))  // Input Register

// Macro to calculate control register address for a specific pin
#define GPIO_CTRL(pin)  (*(volatile uint32_t *)(GPIO_BASE + 0x04 + (pin * 8)))
int main() {
    // Set pin 0 as output
    GPIO_OE |= (1 << 0);        // Set bit 0 of the OE register to 1 (output enable)
    GPIO_CTRL(0) = 5;           // Set function select to SIO ( single input / output -- normal gpio operation ) for pin 0

    // Set pin 1 as input
    GPIO_OE &= ~(1 << 1);       // Set bit 1 of the OE register to 0 (input enable)
    GPIO_CTRL(1) = 5;           // Set function select to SIO (( single input / output -- normal gpio operation )) for pin 1

    // Set pin 0 high (output)
    GPIO_OUT |= (1 << 0);       // Set pin 0 to high

    // Read pin 1 (input)
    uint32_t pin1_value = (GPIO_IN & (1 << 1)) != 0;  // Read the state of pin 1

    // Set pin 0 low (output)
    GPIO_OUT &= ~(1 << 0);      // Set pin 0 to low

    // Continue running (this is just an example, so infinite loop)
    while (1);

    return 0;
}
```

Note : Extra information Don't Memorize

•**GPIO_OE**: Output enable register (1 for output, 0 for input).
32-bit register , each bit maps to a corresponding GPIO PIN resembling the **DDRX** .

•**GPIO_OUT**: Output value register. Resembles **PORTX** in output pins, used to enter the output needed in the pin in its corresponding bit .

•**GPIO_IN**: Input value register. Used to read the Value of the input pin using its corresponding bit resembling PINX.

SIO: GPIO_OE Register

Offset: 0x020

Description

GPIO output enable

Table 24. GPIO_OE Register

Bits	Description	Type	Reset
31:30	Reserved.	-	-

2.3. Processor subsystem 46

RP2040 Datasheet

Bits	Description	Type	Reset
29:0	Set output enable (1/0 → output/input) for GPIO0...29. Reading back gives the last value written. If core 0 and core 1 both write to GPIO_OE simultaneously (or to a SET/CLR/XOR alias), the result is as though the write from core 0 took place first, and the write from core 1 was then applied to that intermediate result.	RW	0x00000000

Note : Extra information
Don't Memorize

GPIO_CTRL Register Overview :

There is a 32-bit GPIO control register for each pin , separated by 8 bytes in the memory address space.

1.Purpose: The GPIO_CTRL register allows you to configure various attributes of GPIO pins, such as their modes, functions. **Function Selection**: Determines the function of the GPIO pin (e.g., GPIO, UART, SPI).

IO_BANK0: GPIO0_CTRL, GPIO1_CTRL, …, GPIO28_CTRL, GPIO29_CTRL Registers

Offsets: 0x004, 0x00c, …, 0x0e4, 0x0ec

Description

GPIO control including function select and overrides.

Table 285. GPIO0_CTRL, GPIO1_CTRL, …, GPIO28_CTRL, GPIO29_CTRL Registers

Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
29:28	IRQOVER	0x0 → don't invert the interrupt 0x1 → invert the interrupt 0x2 → drive interrupt low 0x3 → drive interrupt high	RW	0x0
27:18	Reserved.	-	-	-
17:16	INOVER	0x0 → don't invert the peri input 0x1 → invert the peri input 0x2 → drive peri input low 0x3 → drive peri input high	RW	0x0

2.19. GPIO 247

RP2040 Datasheet

Bits	Name	Description	Type	Reset
15:14	Reserved.	-	-	-
13:12	OEOVER	0x0 → drive output enable from peripheral signal selected by funcsel 0x1 → drive output enable from inverse of peripheral signal selected by funcsel 0x2 → disable output 0x3 → enable output	RW	0x0
11:10	Reserved.	-	-	-
9:8	OUTOVER	0x0 → drive output from peripheral signal selected by funcsel 0x1 → drive output from inverse of peripheral signal selected by funcsel 0x2 → drive output low 0x3 → drive output high	RW	0x0
7:5	Reserved.	-	-	-
4:0	FUNCSEL	Function select. 31 == NULL. See GPIO function table for available functions.	RW	0x1f

Note : Extra information Don't Memorize

However writing Low level code is extremely tiring and complicated !

gpio.c

gpio.c > ...

```c
#include <stdint.h>

#define GPIO_BASE       0x40014000  // Base address for GPIO registers
#define GPIO_OE         (*(volatile uint32_t *)(GPIO_BASE + 0x20))  // Output Enable Register
#define GPIO_OUT        (*(volatile uint32_t *)(GPIO_BASE + 0x10))  // Output Register
#define GPIO_IN         (*(volatile uint32_t *)(GPIO_BASE + 0x14))  // Input Register

// Macro to calculate control register address for a specific pin
#define GPIO_CTRL(pin)  (*(volatile uint32_t *)(GPIO_BASE + 0x04 + (pin * 8)))
int main() {
    // Set pin 0 as output
    GPIO_OE |= (1 << 0);        // Set bit 0 of the OE register to 1 (output enable)
    GPIO_CTRL(0) = 5;           // Set function select to SIO ( single input / output -- normal gpio operation ) for pin 0

    // Set pin 1 as input
    GPIO_OE &= ~(1 << 1);       // Set bit 1 of the OE register to 0 (input enable)
    GPIO_CTRL(1) = 5;           // Set function select to SIO (( single input / output -- normal gpio operation )) for pin 1

    // Set pin 0 high (output)
    GPIO_OUT |= (1 << 0);       // Set pin 0 to high

    // Read pin 1 (input)
    uint32_t pin1_value = (GPIO_IN & (1 << 1)) != 0;  // Read the state of pin 1

    // Set pin 0 low (output)
    GPIO_OUT &= ~(1 << 0);      // Set pin 0 to low

    // Continue running (this is just an example, so infinite loop)
    while (1);

    return 0;
}
```

Pico SDK C library provides a high-level API for the hardware peripherals

Link : https://www.raspberrypi.com/documentation/pico-sdk/

GPIO Functions in Pico SDK

1.Initialization and Direction

- **gpio_init(uint gpio):** Initializes the specified GPIO pin.
- **gpio_set_dir(uint gpio, bool out):** Sets the direction of the GPIO pin (input or output).
  - out = true for output, out = false for input.

2.Setting and Reading GPIO States

- **gpio_put(uint gpio, bool value):** Sets the state of an output GPIO pin (HIGH or LOW).
- **gpio_get(uint gpio):** Reads the current state of an input GPIO pin.

3.Pull-up/Pull-down Control

- **gpio_pull_up(uint gpio):** Enables the internal pull-up resistor on the specified pin.
- **gpio_pull_down(uint gpio):** Enables the internal pull-down resistor on the specified pin.
- **gpio_disable_pulls(uint gpio):** Disables both pull-up and pull-down resistors on the pin.

Utilized the GPIO SDK API to initialize pins , set their directions , set output and receive input .

gpio.c

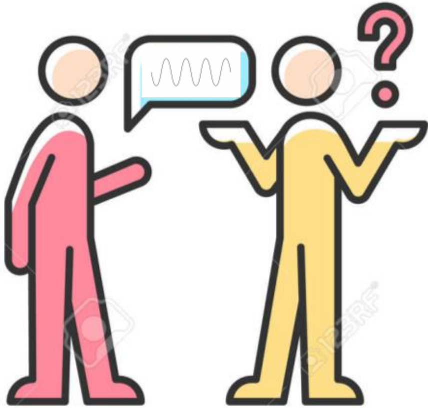
gpio.c > main()

```c

int main() {
    // Define pins
    const uint LED_PIN = 2;                // Output pin (for LED)
    const uint INPUT_PIN_A = 5;            // Input pin for button A (positive logic)
    const uint INPUT_PIN_B = 3;            // Input pin for button B (negative logic with pull-up)
    // Initialize output pin (LED)
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);  // Set as output
    // Initialize input pins
    gpio_init(INPUT_PIN_A);
    gpio_set_dir(INPUT_PIN_A, GPIO_IN);    // Set as input

    gpio_init(INPUT_PIN_B);
    gpio_set_dir(INPUT_PIN_B, GPIO_IN);    // Set as input
    gpio_pull_up(INPUT_PIN_B);             // Enable pull-up resistor for negative logic

    // Infinite loop to check button states and control the LED
    while (1) {
        // Read the state of button A (positive logic)
        bool button_a_state = gpio_get(INPUT_PIN_A);  // HIGH when not pressed, LOW when pressed
        // Read the state of button B (negative logic, pull-up enabled)
        bool button_b_state = gpio_get(INPUT_PIN_B);  // LOW when not pressed, HIGH when pressed
        if (button_a_state) {
            // If button A is pressed (positive logic), turn on the LED
            gpio_put(LED_PIN, 1);  // Set the LED pin to HIGH (turn on)
        }

        if (!button_b_state) {
            // If button B is pressed (negative logic), turn off the LED
            gpio_put(LED_PIN, 0);  // Set the LED pin to LOW (turn off)
        }

        sleep_ms(100);  // Small delay to debounce buttons
    }
    return 0;
}
```

Outline :

- Recap.
- **Sensors**
- Example
- ADC
- Example

Different physical changes happen in the environment around our system, and the system needs to measure these changes to respond to them.
BUT …
The system cannot understand the language the environment speaks



The environment *The system*

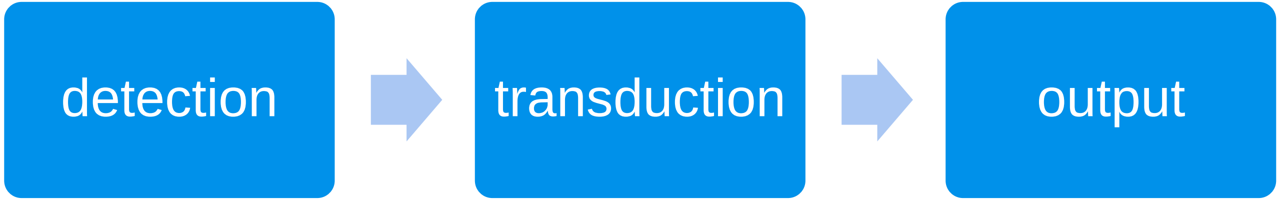
Sensors

That is why embedded system need sensors.
They are devices that detect or measure physical changes in the environment and convert them into electrical Signals or readable inputs to the system, to enable the system to respond to changes.

The environment Sensor *The system*

Sensors

The process undergoes 3 stages

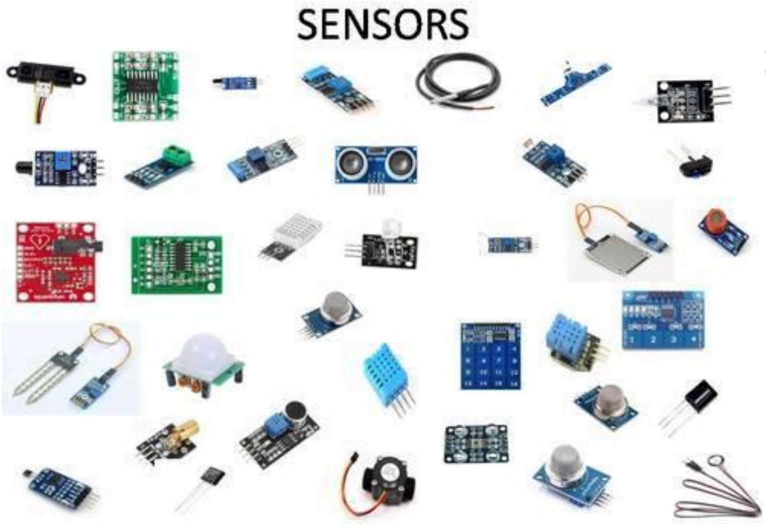


Detection

Sensors are designed to detect a physical phenomenon or a property

Examples

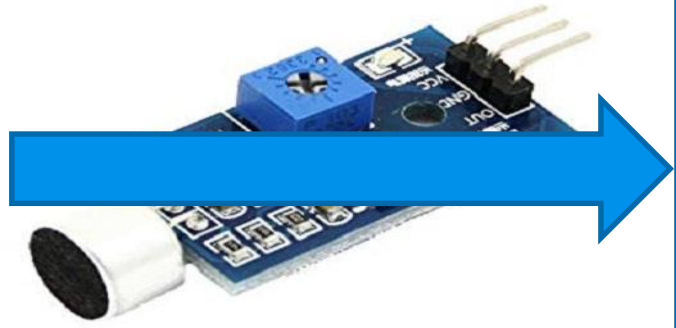
- Proximity sensor
- Temperature Sensor
- Infra-red sensor
- Light intensity sensor
- Microphone
- Pressure sensor
- Color sensor



Transduction

Conversion of the physical phenomenon into a measurable signal.
The measurable signal can be Vibrational (sound) , Thermal , optical, mechanical or any type of form / energy which can be eventually converted to **Electrical Output Signal** .

Sound vibrations



Electrical signal

Examples

Sensor	functionality	from	to
Digital IR (infra-red sensor)	Detect presence of an object within a distance based on infra-red radiation	IR radiation	Digital electrical signal
Temperature	Measures amount of heat energy	Heat energy	Analogue electrical Signal
Ultrasonic	Measures distance/presence of target object	vibrations	Analogue/digital electrical signals
Light Intensity	Measures Light intensity	Light energy	Analogue electrical signals
Sound / Microphone	Measures sound level	Sound vibrations	Analogue electrical signals

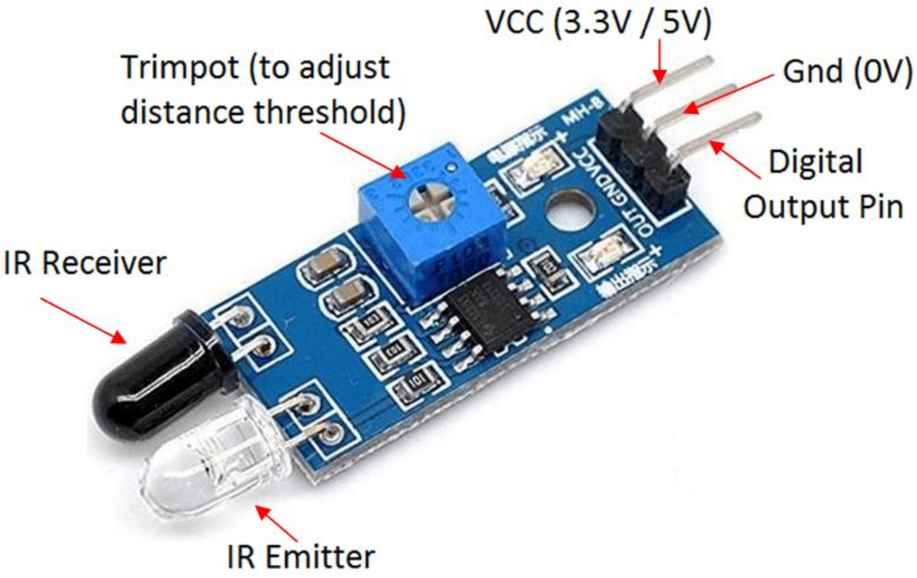
Outline :

- Recap.
- Sensors
- **Infra-red Implementation**
- ADC
- Potentiometer ADC Implementation
- Why Drivers ?

Let’s test a digital IR sensor, if the IR sensor is activated , toggle the built in LED

NOTE : Check the Sensor datasheet for the sensor Aspects :

- **Active-High / Active-Low ?**
- **Operating Voltage VCC ?**
- **How to adjust sensor threshold ?**



NOTE : Check the Sensor datasheet for the sensor Aspects :

- **Active-High / Active-Low ?**
- **Operating Voltage VCC → 3-5?**
- **How to adjust sensor threshold ?**

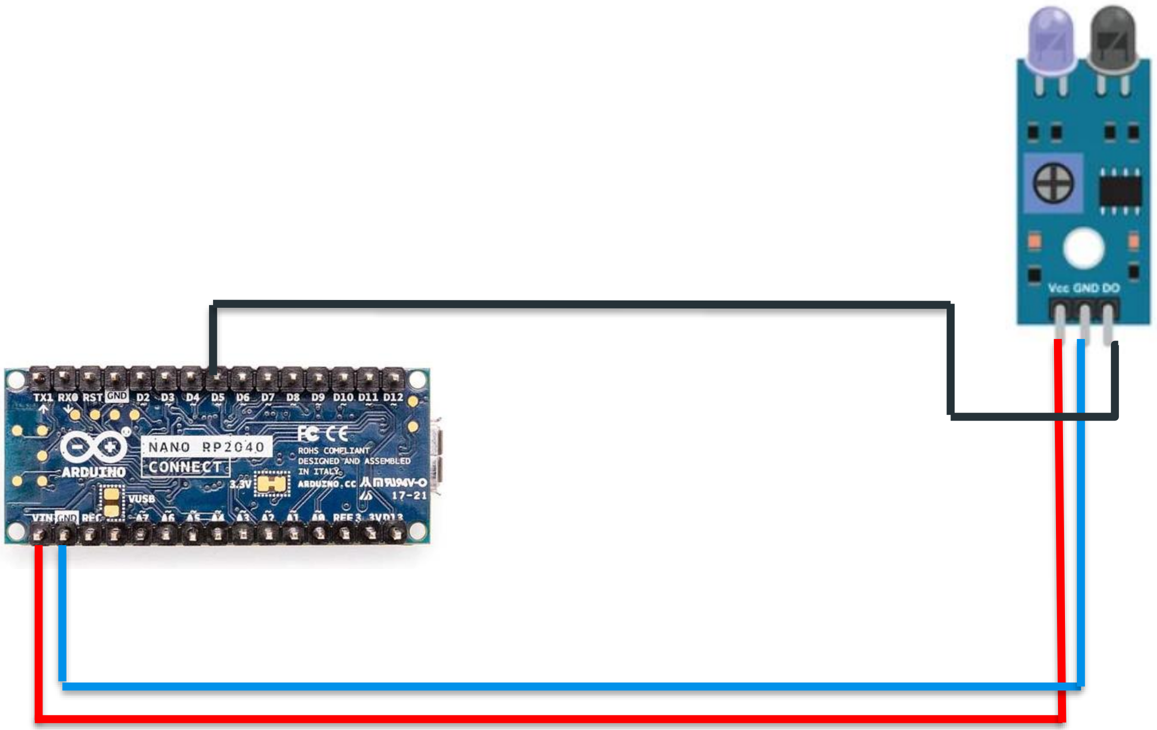
↑ | ↓ Rotating the knob CW

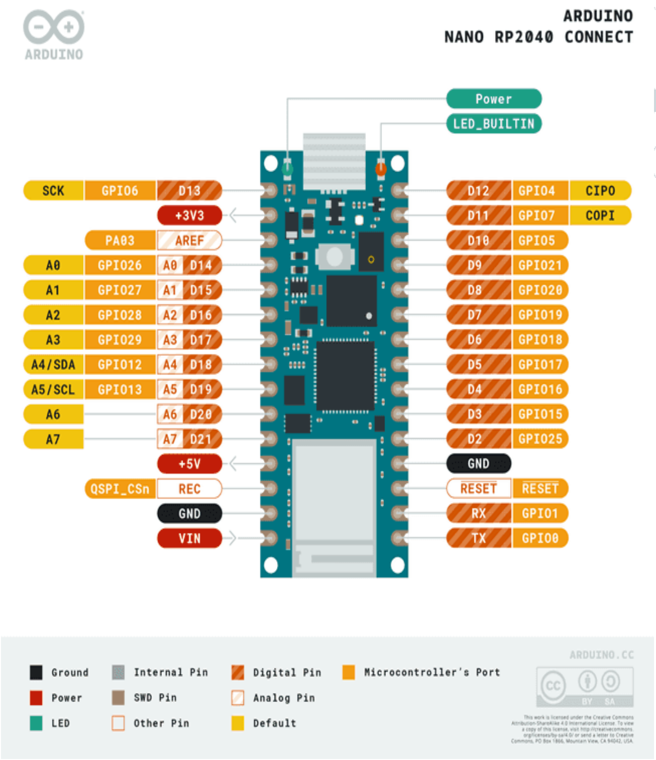
Features: IR Sensor Module With pot (ADIY)

- Onboard detection indication
- The effective distance range of 2cm to 10cm
- A preset knob to fine-tune the distance range
- There is an obstacle, the green indicator light on the circuit board
- TTL output is high whenever it senses an obstacle
- Detection angle: 35 °
- Comparator chip: LM393
- 3mm screw holes for easy mounting
- Dimensions: 48 x 14 x 8 mm (L x W x H)
- Weight: 3gm

The module features a 3-wire interface with VCC, GND, and an OUT pin on its tail. It works fine with 3v3 to 5V levels. Upon hindrance/reflectance, the output pin gives out a digital signal (a low-level signal). The onboard preset helps to fine-tune the range of operation, effective distance range is 2cm to 10cm.

Wiring On Arduino Nano RP2040 (ARM based)





Note : Extra information
Don't Memorize

Low-level code

Header File for definition and function Prototypes .h file .

```
#include <stdint.h>

#define GPIO_BASE      0x40014000  // GPIO base address
#define GPIO_OUT       (GPIO_BASE + 0x10)  // GPIO output register
#define GPIO_OUT_SET    (GPIO_BASE + 0x14)  // GPIO output set register
#define GPIO_OUT_CLR    (GPIO_BASE + 0x18)  // GPIO output clear register
#define GPIO_OE        (GPIO_BASE + 0x20)  // GPIO output enable register
#define GPIO_OE_SET     (GPIO_BASE + 0x24)  // GPIO output enable set
#define GPIO_OE_CLR     (GPIO_BASE + 0x28)  // GPIO output enable clear
#define GPIO_IN        (GPIO_BASE + 0x4C) // GPIO input register
```

Paste them in a header file and name it “GPIO.h” Then : #include”GPIO.h”

```
// Define pin numbers
#define LED_PIN        25  // LED (GPIO 25)
#define SENSOR_PIN      16  // IR sensor (GPIO 16)

// Bit manipulation macros for low-level register operations
#define BIT_SET(reg, pin)    (*(volatile uint32_t *)(reg) |= (1 << (pin)))
#define BIT_CLEAR(reg, pin)  (*(volatile uint32_t *)(reg) &= ~(1 << (pin)))
#define BIT_TOGGLE(reg, pin) (*(volatile uint32_t *)(reg) ^= (1 << (pin)))
#define BIT_READ(reg, pin)   (*(volatile uint32_t *)(reg) & (1 << (pin)))
```

Low-level code

.C implementation file

```
// Initialize GPIO pins
void gpio_init() {
    // Set LED_PIN as output by setting its bit in GPIO_OE_SET
    BIT_SET(GPIO_OE_SET, LED_PIN);

    // Set SENSOR_PIN as input by clearing its bit in GPIO_OE_CLR
    BIT_CLEAR(GPIO_OE_SET, SENSOR_PIN);  // Ensure input is
cleared
}

// Toggle the LED state
void toggle_led() {
    BIT_TOGGLE(GPIO_OUT, LED_PIN);  // Toggle the LED pin
}

// Read the sensor input state
uint32_t read_sensor() {
    return BIT_READ(GPIO_IN, SENSOR_PIN);  // Return the state of
the IR sensor
}
```

main.C file (application file)

```
int main() {
    gpio_init();  // Initialize the GPIO pins

    while (1) {
        // Read the current state of the sensor
        uint32_t sensor_state = read_sensor();

        // Check if the sensor is HIGH
        if (sensor_state) {
            // Toggle the LED
            toggle_led();
        }

        while(read_sensor()); // wait till sensor is not HIGH
    }

    return 0;
}
```

Pico SDK C library provides a high-level API for the hardware peripherals

Link : https://www.raspberrypi.com/documentation/pico-sdk/

RECALLING

GPIO Functions in Pico SDK

1. **Initialization and Direction**
   - **gpio_init(uint gpio):** Initializes the specified GPIO pin.
   - **gpio_set_dir(uint gpio, bool out):** Sets the direction of the GPIO pin (input or output).
     - out = true for output, out = false for input.
2. **Setting and Reading GPIO States**
   - **gpio_put(uint gpio, bool value):** Sets the state of an output GPIO pin (HIGH or LOW).
   - **gpio_get(uint gpio):** Reads the current state of an input GPIO pin.
3. **Pull-up/Pull-down Control**
   - **gpio_pull_up(uint gpio):** Enables the internal pull-up resistor on the specified pin.
   - **gpio_pull_down(uint gpio):** Enables the internal pull-down resistor on the specified pin.
   - **gpio_disable_pulls(uint gpio):** Disables both pull-up and pull-down resistors on the pin.

C/C++ SDK Code (HIGH Level)

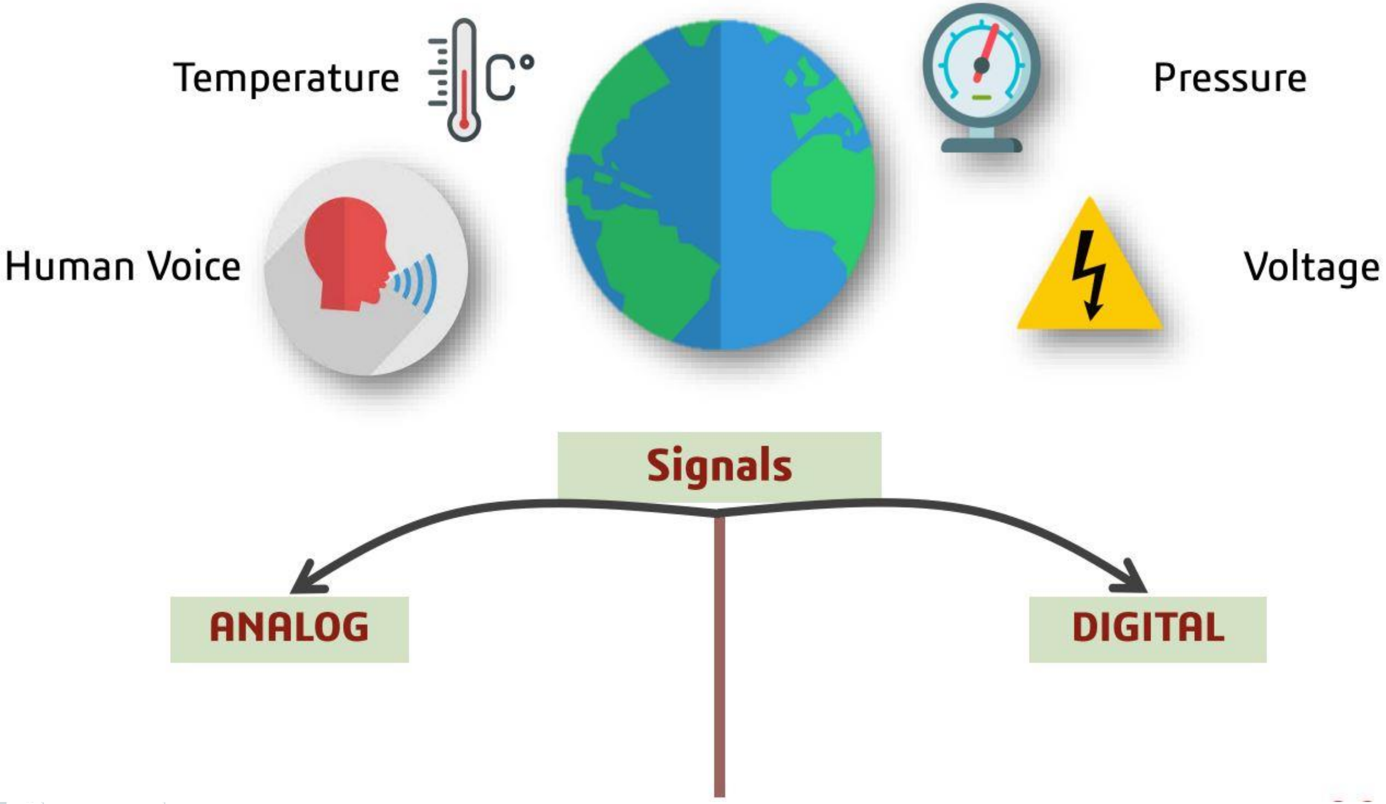
```
#include "pico/stdlib.h"
#include "hardware/gpio.h"    // including the gpio header file

#define LED_PIN         25 // Built-in LED pin
#define SENSOR_PIN      16 // Pin connected to IR sensor

// Toggle the LED state
void toggle_led() {
    gpio_put(LED_PIN, !gpio_get(LED_PIN));
}

// Set up GPIO for the LED and sensor
void gpio_init_custom() {
    // Set the LED pin as output
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);

    // Set the sensor pin as input
    gpio_init(SENSOR_PIN);
    gpio_set_dir(SENSOR_PIN, GPIO_IN);
}
```

```
// Main loop to check the sensor signal and toggle the
LED
int main() {
    // Initialize the GPIO
    gpio_init_custom();

    while (true) {
        // Read the current state of the sensor
        bool state= gpio_get(SENSOR_PIN);

        // Check if the sensor state has changed (crossing
event)
        if (state) {
            // Toggle the LED if the sensor signal changes
            toggle_led();
        }

        // wait for the IR Sensor to deactivate
        while(state);
    }

    return 0;
}
```

Outline :

- Recap.
- Sensors
- Infra-red Implementation
- **ADC**
- Potentiometer ADC Implementation
- Why Drivers ?

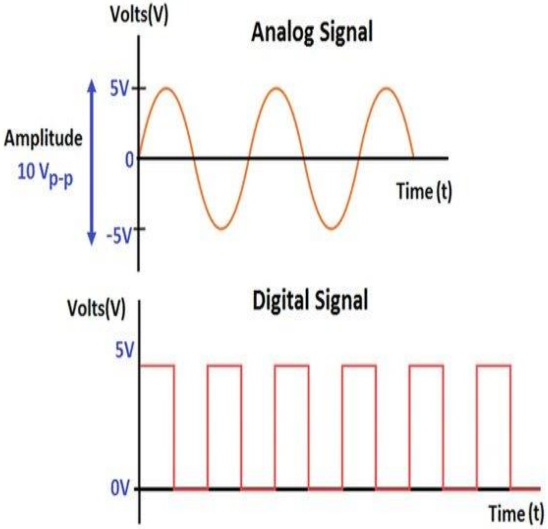
Real World Data



Output Signal

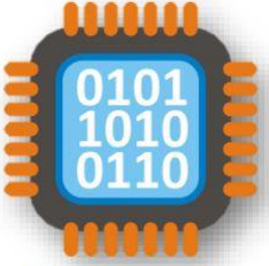
<u>The electrical output signal can be either Analogue or digital signals depending on the sensor :</u>

- An analog signal is a continuous representation of a physical quantity that can vary smoothly over time. It is characterized by an infinite number of possible values within a given range . An electrical analogue signal can have any voltage value from (0 to 5 V) for example .
- A digital signal is a discrete representation either 0 (low) or 1 (HIGH) perfect as an input for binary systems as Microcontrollers . An electrical digital signal output from an Active-High digital sensor can 5V (On state) or 0 (Off state) V for example.



Analog Signals	Digital Signals
Continuous and smooth waveform	Discrete and stepped waveform
Represents real-world data as a continuously varying voltage or current .Range of values (0 to 5 V).	Represents data as discrete values (0s and 1s). Either 0 (LOW) or 1 (HIGH) .
Output is an analog voltage or current directly proportional to the measured quantity	Output is a binary representation (0 or 1) of the measured quantity
Infinite resolution, theoretically	Limited by the number of bits in the ADC (e.g., 8-bit, 10-bit, 12-bit)
Requires specialized analog processing circuitry (filters, amplifiers)	Easily processed using digital logic
Prone to signal degradation during transmission and storage	Less susceptible to degradation; easier to transmit and store

Microcontroller will receive **analog signals**



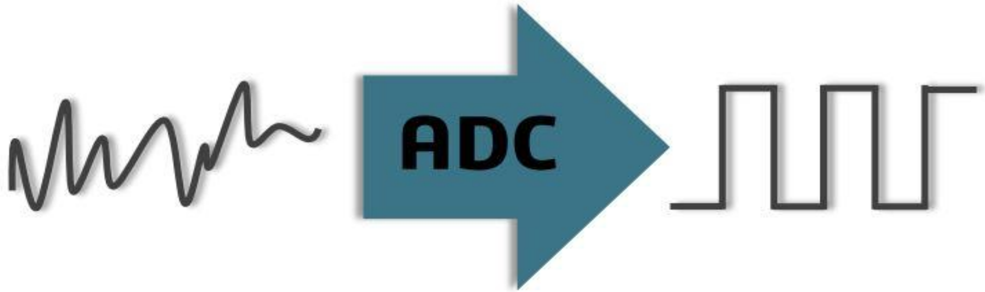
SO

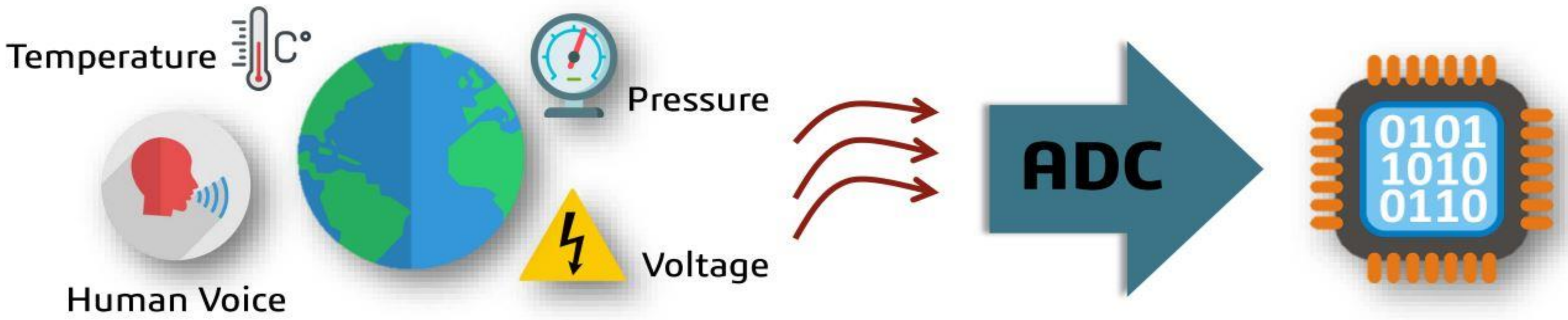
What if we need to get some non-digital data in the microcontroller ?

but

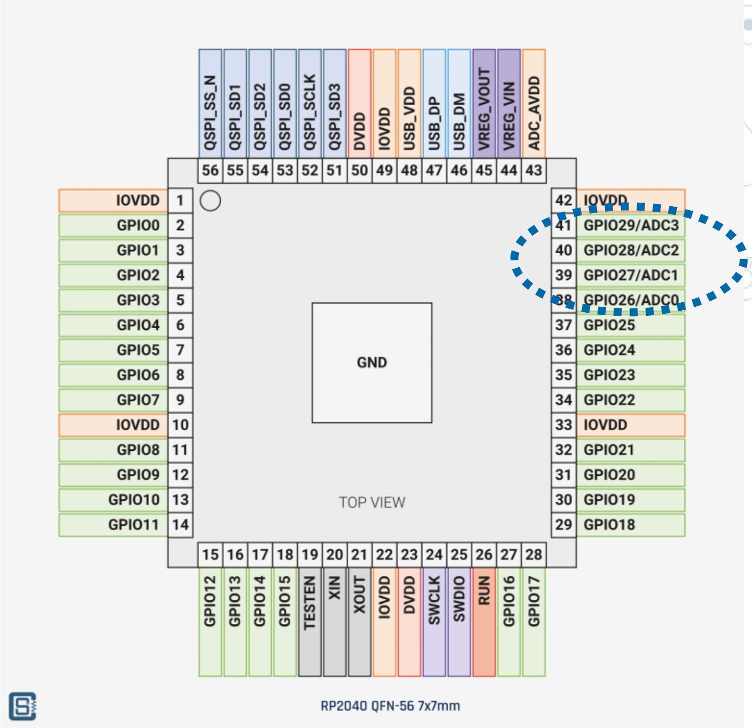
The Microcontroller have to deal with digital information “They only understand ‘0’ or ‘1’ values

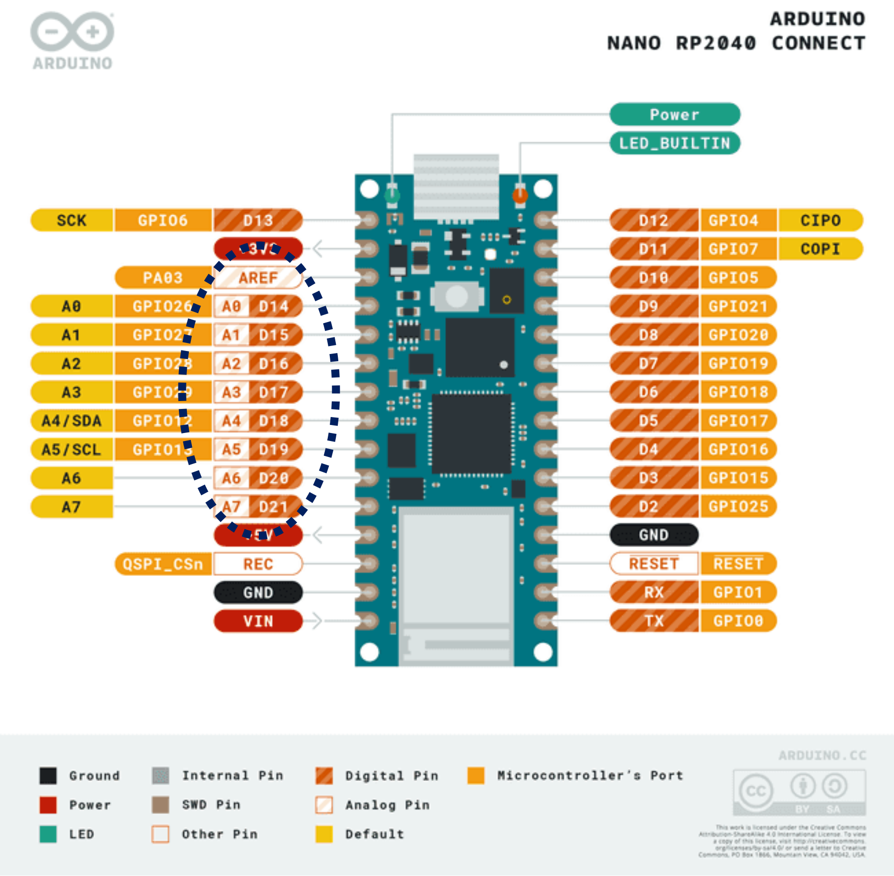
ANALOG TO DIGITAL CONVERTER

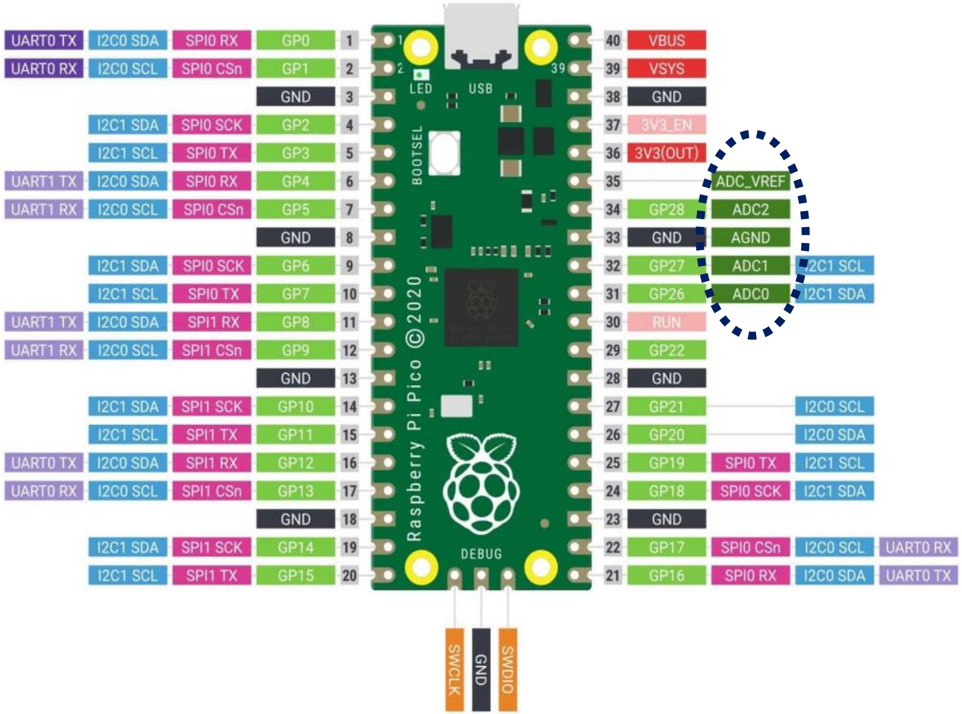


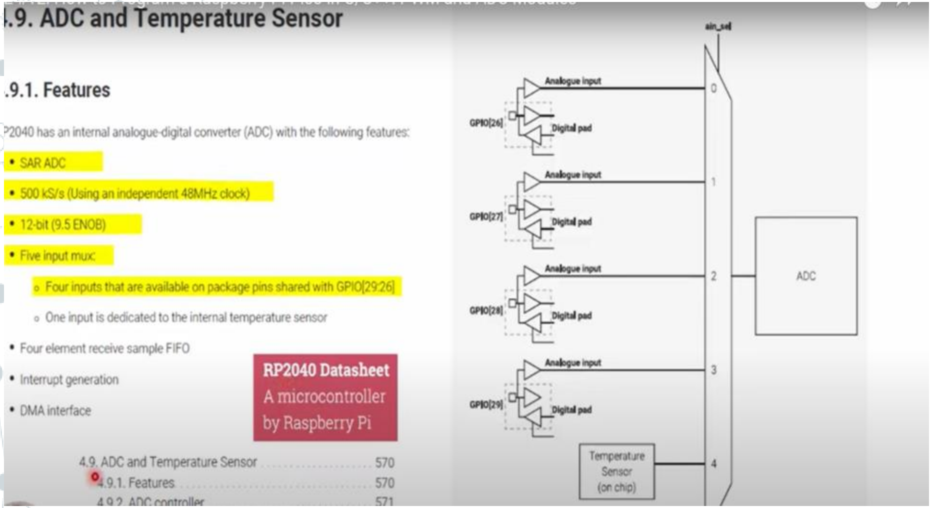


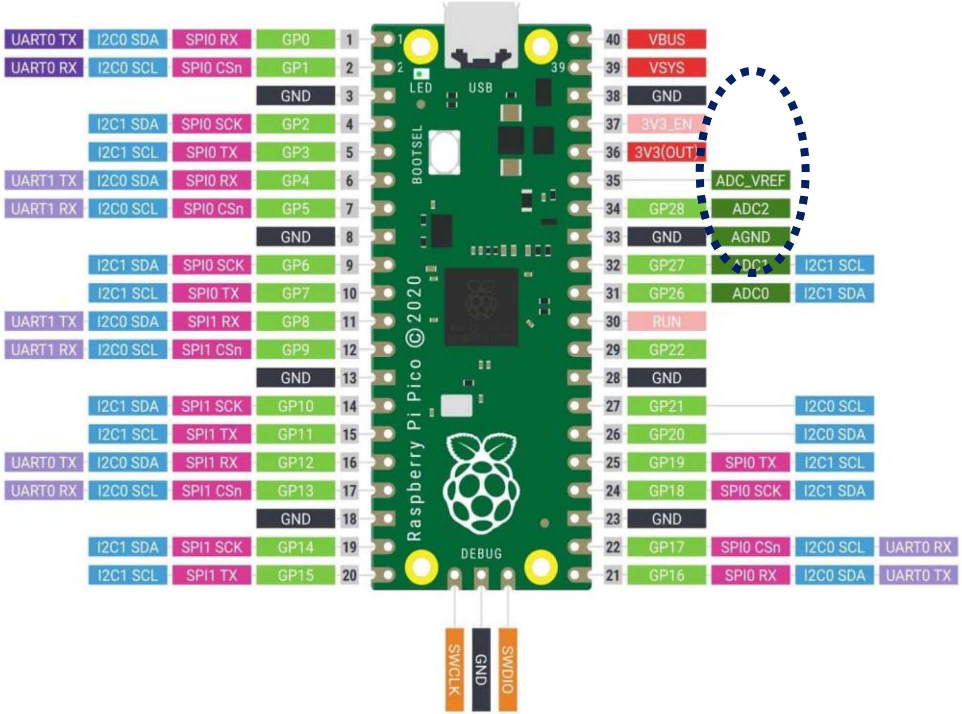
- **RP2040 has 4 ADC channels** which can be used to read analog signal in the range 0-5V.
- It has 12-bit ADC means it will give digital value in the **range of 0 – 4095 ((2^12)-1).** This is called as a resolution which indicates the number of discrete values it can produce over the range of analog values.
- ADC channels works at 48MHz, each one conversion takes **(96 x 1 / 48MHz)** = $2\mu s$ per sample **(500kSample/s)** (sampling frequency)




ARDUINO
ARDUINO
NANO RP2040 CONNECT
Power
LED_BUILTIN
SCK GPIO6 D13
3V3
PA03 AREF
A0 GPIO26 A0 D14
A1 GPIO27 A1 D15
A2 GPIO28 A2 D16
A3 GPIO29 A3 D17
A4/SDA GPIO12 A4 D18
A5/SCL GPIO13 A5 D19
A6 A6 D20
A7 A7 D21
+5V
QSPI_CSn REC
GND
VIN
D12 GPIO4 CIPO
D11 GPIO7 COPI
D10 GPIO5
D9 GPIO21
D8 GPIO20
D7 GPIO19
D6 GPIO18
D5 GPIO17
D4 GPIO16
D3 GPIO15
D2 GPIO25
GND
RESET RESET
RX GPIO1
TX GPIO0
Ground
Power
LED
Internal Pin
SWD Pin
Other Pin
Digital Pin
Analog Pin
Default
Microcontroller's Port
ARDUINO.CC


UART0 TX I2C0 SDA SPI0 RX GP0 1
UART0 RX I2C0 SCL SPI0 CSn GP1 2
GND 3
I2C1 SDA SPI0 SCK GP2 4
I2C1 SCL SPI0 TX GP3 5
UART1 TX I2C0 SDA SPI0 RX GP4 6
UART1 RX I2C0 SCL SPI0 CSn GP5 7
GND 8
I2C1 SDA SPI0 SCK GP6 9
I2C1 SCL SPI0 TX GP7 10
UART1 TX I2C0 SDA SPI1 RX GP8 11
UART1 RX I2C0 SCL SPI1 CSn GP9 12
GND 13
I2C1 SDA SPI1 SCK GP10 14
I2C1 SCL SPI1 TX GP11 15
UART0 TX I2C0 SDA SPI1 RX GP12 16
UART0 RX I2C0 SCL SPI1 CSn GP13 17
GND 18
I2C1 SDA SPI1 SCK GP14 19
I2C1 SCL SPI1 TX GP15 20
40 VBUS
39 VSYS
38 GND
37 3V3_EN
36 3V3(OUT)
35 ADC_VREF
34 GP28 ADC2
33 GND AGND
32 GP27 ADC1 I2C1 SCL
31 GP26 ADC0 I2C1 SDA
30 RUN
29 GP22
28 GND
27 GP21 I2C0 SCL
26 GP20 I2C0 SDA
25 GP19 SPI0 TX I2C1 SCL
24 GP18 SPI0 SCK I2C1 SDA
23 GND
22 GP17 SPI0 CSn I2C0 SCL UART0 RX
21 GP16 SPI0 RX I2C0 SDA UART0 TX
LED
USB
BOOTSEL
Raspberry Pi Pico ©2020
DEBUG
SWCLK
GND
SWDIO


.9. ADC and Temperature Sensor
.9.1. Features
P2040 has an internal analogue-digital converter (ADC) with the following features:
• SAR ADC
• 500 kS/s (Using an independent 48MHz clock)
• 12-bit (9.5 ENOB)
• Five input mux:
∘ Four inputs that are available on package pins shared with GPIO[29:26]
∘ One input is dedicated to the internal temperature sensor
• Four element receive sample FIFO
• Interrupt generation
• DMA interface
RP2040 Datasheet
A microcontroller by Raspberry Pi
4.9. ADC and Temperature Sensor 570
4.9.1. Features 570
ain_sel
Analogue input
Digital pad
GPIO[26]
GPIO[27]
GPIO[28]
GPIO[29]
ADC
Temperature Sensor (on chip)


UART0 TX I2C0 SDA SPI0 RX GP0 1
UART0 RX I2C0 SCL SPI0 CSn GP1 2
GND 3
I2C1 SDA SPI0 SCK GP2 4
I2C1 SCL SPI0 TX GP3 5
UART1 TX I2C0 SDA SPI0 RX GP4 6
UART1 RX I2C0 SCL SPI0 CSn GP5 7
GND 8
I2C1 SDA SPI0 SCK GP6 9
I2C1 SCL SPI0 TX GP7 10
UART1 TX I2C0 SDA SPI1 RX GP8 11
UART1 RX I2C0 SCL SPI1 CSn GP9 12
GND 13
I2C1 SDA SPI1 SCK GP10 14
I2C1 SCL SPI1 TX GP11 15
UART0 TX I2C0 SDA SPI1 RX GP12 16
UART0 RX I2C0 SCL SPI1 CSn GP13 17
GND 18
I2C1 SDA SPI1 SCK GP14 19
I2C1 SCL SPI1 TX GP15 20
40 VBUS
39 VSYS
38 GND
37 3V3_EN
36 3V3(OUT)
35 ADC_VREF
34 GP28 ADC2
33 GND AGND
32 GP27 ADC1 I2C1 SCL
31 GP26 ADC0 I2C1 SDA
30 RUN
29 GP22
28 GND
27 GP21 I2C0 SCL
26 GP20 I2C0 SDA
25 GP19 SPI0 TX I2C1 SCL
24 GP18 SPI0 SCK I2C1 SDA
23 GND
22 GP17 SPI0 CSn I2C0 SCL UART0 RX
21 GP16 SPI0 RX I2C0 SDA UART0 TX
LED
USB
BOOTSEL
Raspberry Pi Pico ©2020
DEBUG
SWCLK
GND
SWDIO

Digital Output value Calculation:

- ADC Resolution = Vref / ((2^n) - 1); n= #ADC bits
- Digital Output = Vin / Resolution.

Vref - The reference voltage is the maximum value that the ADC can convert.
To keep things simple, let us consider that Vref is 5V,

- For 0 Vin, digital o/p value = 0
- For 2.5 Vin, digital o/p value = 2047 (12-bit)
- For 5 Vin, digital o/p value = 4095 (12-bit)

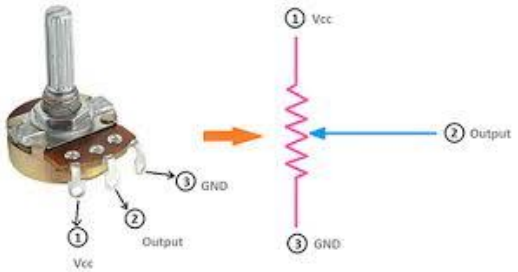
Outline :

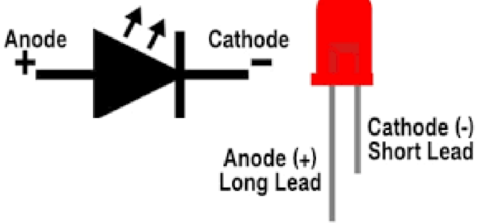
- **Recap.**
- Sensors
- Infra-red Implementation
- ADC
- **Potentiometer ADC Implementation**
- Why Drivers ?

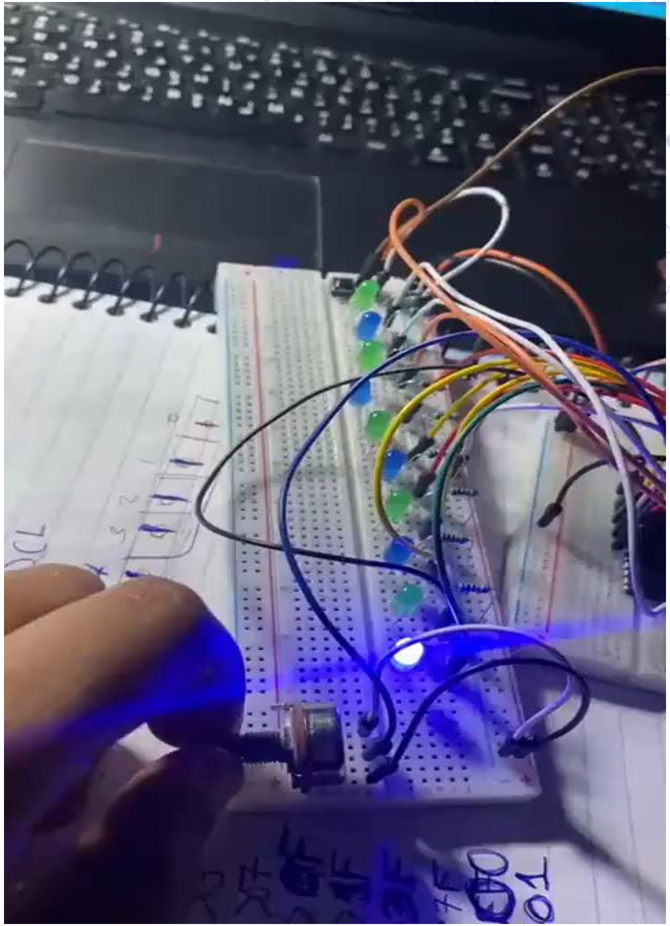
ADC Example

Using a potentiometer and 10 LEDs

- Convert the signal of the Potentiometer to turn on 10 consecutive LEDs using the ADC registers







Note : Extra information
Don't Memorize

Low-level code

```c
#include <stdint.h>

// Define base addresses for GPIO and ADC registers
#define SIO_BASE        0xD0000000
#define GPIO_OUT        (SIO_BASE + 0x10)
#define GPIO_OUT_SET    (SIO_BASE + 0x14)
#define GPIO_OUT_CLR    (SIO_BASE + 0x18)

#define GPIO_BASE       0x40014000
#define GPIO_CTRL(n)    (GPIO_BASE + (n)*8)

#define ADC_BASE        0x4004C000
#define ADC_CS          (ADC_BASE + 0x00)
#define ADC_RESULT      (ADC_BASE + 0x04)
#define ADC_FCS         (ADC_BASE + 0x08)
#define ADC_DIV         (ADC_BASE + 0x0C)
#define ADC_INTS        (ADC_BASE + 0x10)

// Define the pins and constants
#define LED_BASE_PIN    0
#define LED_COUNT       10
#define ADC_CHANNEL     0   // Using ADC channel 0 (GPIO 26)
#define ADC_MAX_VALUE   4095 // 12-bit ADC
#define ADC_SCALE       (ADC_MAX_VALUE / LED_COUNT)
```

4.9.6. List of Registers

The ADC registers start at a base address of 0x4004c000 (defined as ADC_BASE in SDK).

Table 567. List of ADC registers

Offset	Name	Info
0x00	CS	ADC Control and Status
0x04	RESULT	Result of most recent ADC conversion
0x08	FCS	FIFO control and status
0x0c	FIFO	Conversion result FIFO
0x10	DIV	Clock divider. If non-zero, CS_START_MANY will start conversions at regular intervals rather than back-to-back. The divider is reset when either of these fields are written. Total period is 1 + INT + FRAC / 256
0x14	INTR	Raw Interrupts
0x18	INTE	Interrupt Enable
0x1c	INTF	Interrupt Force
0x20	INTS	Interrupt status after masking & forcing

Note : Extra information Don't Memorize

Low-level code

```
void delay(int cycles) {    // Simple delay function
   for (int i = 0; i < cycles; i++) {
      asm volatile ("nop");
   }
}

void gpio_init(uint32_t pin) { // Function to initialize GPIO pins for LEDs
   uint32_t *gpio_ctrl_reg = (uint32_t *)(GPIO_CTRL(pin));
   *gpio_ctrl_reg = 5;  // Select SIO function (5) for GPIO control
}

void gpio_set(uint32_t pin, int value) {  // Function to set a GPIO pin high or low
   if (value) {
      *(volatile uint32_t *)(GPIO_OUT_SET) = (1 << pin);
   } else {
      *(volatile uint32_t *)(GPIO_OUT_CLR) = (1 << pin);
   }
}

void adc_init() {          // Function to initialize the ADC
// Enable the ADC and set the dividers to a reasonable value
   *(volatile uint32_t *)(ADC_CS) = 0;      // Disable ADC
   *(volatile uint32_t *)(ADC_CS) |= 1;     // Enable ADC

   while (*(volatile uint32_t *)(ADC_CS) & (1 << 8));  // Wait for ADC to be ready
}
```

ADC: CS Register

Offset: 0x00

Description

ADC Control and Status

Bits	Name	Description	Type	Reset
31:21	Reserved.	-	-	-
20:16	RROBIN	Round-robin sampling. 1 bit per channel. Set all bits to 0 to disable. Otherwise, the ADC will cycle through each enabled channel in a round-robin fashion. The first channel to be sampled will be the one currently indicated by AINSEL. AINSEL will be updated after each conversion with the newly-selected channel.	RW	0x00
15	Reserved.	-	-	-
14:12	AINSEL	Select analog mux input. Updated automatically in round-robin mode.	RW	0x0
11	Reserved.	-	-	-
10	ERR_STICKY	Some past ADC conversion encountered an error. Write 1 to clear.	WC	0x0
9	ERR	The most recent ADC conversion encountered an error; result is undefined or noisy.	RO	0x0
8	READY	1 if the ADC is ready to start a new conversion. Implies any previous conversion has completed. 0 whilst conversion in progress.	RO	0x0
7:4	Reserved.	-	-	-
3	START_MANY	Continuously perform conversions whilst this bit is 1. A new conversion will start immediately after the previous finishes.	RW	0x0
2	START_ONCE	Start a single conversion. Self-clearing. Ignored if start_many is asserted.	SC	0x0
1	TS_EN	Power on temperature sensor. 1 - enabled. 0 - disabled.	RW	0x0
0	EN	Power on ADC and enable its clock. 1 - enabled. 0 - disabled.	RW	0x0

Low-level code

Note : Extra information Don't Memorize

ADC: RESULT Register

Offset: 0x04

Table 569. RESULT Register

Bits	Description	Type	Reset
31:12	Reserved.	-	-
11:0	Result of most recent ADC conversion	RO	0x000

```
// Function to read the ADC value from the potentiometer
uint16_t adc_read() {
// Select the ADC channel (0 for GPIO26)
   *(volatile uint32_t *)(ADC_CS) |= (ADC_CHANNEL<<12);  // Set channel
   *(volatile uint32_t *)(ADC_CS) |= (1 << 3);           // Start continous conversion

   // Return the 12-bit ADC result
   while (!(*(volatile uint32_t *)(ADC_CS) & (1 << 8)));
   return *(volatile uint32_t *)(ADC_RESULT) & 0xFFF;  Get 12 ADC bits
}

// Function to control LEDs based on the ADC value
void control_leds(int num_leds) {
   for (int i = 0; i < LED_COUNT; i++) {
      gpio_set(LED_BASE_PIN + i, (i < num_leds) ? 1 : 0);
   }
}
```

- **READY (bit 8)** tells you when the conversion is done and it's safe to read the result.
- **ERR (bit 9)** signals if there was an error during conversion.
- **START_ONCE (bit 2)** is used to start a single ADC conversion.
- **START_MANY (bit 3)** allows continuous conversion.
- **TS_EN (bit 1)** is used to enable the internal temperature sensor.
- **EN (bit 0)** powers on the ADC and enables its clock.
- **AINSEL (12:14)** : channel number 0-4

Low-level code

```
int main() {
    // Initialize the LEDs (GPIO pins)
    for (int i = 0; i < LED_COUNT; i++) {
        gpio_init(LED_BASE_PIN + i);
    }

    // Initialize the ADC
    adc_init();

    while (1) {
        // Read the potentiometer value from the ADC
        uint16_t adc_value = adc_read();

        // Calculate the number of LEDs to light up
        int leds_to_light = adc_value / ADC_SCALE;

        // Control the LEDs based on the ADC value
        control_leds(leds_to_light);

        // Delay to prevent rapid changes
        delay(1000000);
    }

    return 0;
}
```

•**Initialization**: Power on the ADC and configure settings.
ADC_CS → **Set** bit 0 to enable ADC.
•**Channel Selection**: Use AINSEL (12:14) bits to select the input channel.
•**Start Conversion**: Trigger the ADC to start sampling.
Set bit 2 for single conversion or bit 3 for continuous conversion .
•**Read Result**: Wait for the conversion to complete and read the result.
Checking the Ready bit (8^{th} bit) .
•**Utilization**: Process the ADC value in your application logic.

Pico C SDK

Introduction

Hardware APIs

- hardware_adc
  - Detailed Description
  - Functions
  - Function Documentation
- hardware_base
  - Detailed Description
  - Functions
  - Function Documentation
- hardware_claim
  - Detailed Description
  - Functions
  - Function Documentation
- hardware_clocks
  - Detailed Description
  - Typedefs
  - Enumerations
  - Functions
  - Typedef Documentation

Functions

`void adc_init (void)`
Initialise the ADC HW.

`static void adc_gpio_init (uint gpio)`
Initialise the gpio for use as an ADC pin.

`static void adc_select_input (uint input)`
ADC input select.

`static uint adc_get_selected_input (void)`
Get the currently selected ADC input channel.

`static void adc_set_round_robin (uint input_mask)`
Round Robin sampling selector.

`static void adc_set_temp_sensor_enabled (bool enable)`
Enable the onboard temperature sensor.

`static uint16_t adc_read (void)`
Perform a single conversion.

`static void adc_run (bool run)`
Enable or disable free-running sampling mode.

`static void adc_set_clkdiv (float clkdiv)`
Set the ADC Clock divisor.

`static void adc_fifo_setup (bool en, bool dreq_en, uint16_t dreq_thresh, bool err_in_fifo, bool byte_shift)`
Setup the ADC FIFO.

`static bool adc_fifo_is_empty (void)`

C/C++ SDK Code

```
#include "pico/stdlib.h"
#include "hardware/adc.h"

#define POTENTIOMETER_PIN        26        // ADC input pin
#define LED_BASE_PIN             0         // Starting pin for LEDs (assuming LEDs are connected to GPIO 0-9)

void setup_adc() {
    // Initialize ADC
    adc_init();

    // Select the input pin for the potentiometer (ADC pin)
    adc_gpio_init(POTENTIOMETER_PIN);
    adc_select_input(0);       // Corresponds to GPIO26/ADC0
}

int read_adc() {
    // Read raw ADC value (12-bit resolution, range 0-4095)
    uint16_t adc_value = adc_read();

    // Scale down to 10 LEDs, so range is 0-10
    return (adc_value * 10) / 4095;
}
```

```
void control_leds(int num_leds) {
    // Turn on LEDs sequentially based on the ADC value
    for (int i = 0; i < 10; i++) {
        if (i < num_leds) {
            gpio_put(LED_BASE_PIN + i, 1); // Turn on the LED
        } else {
            gpio_put(LED_BASE_PIN + i, 0); // Turn off the
LED
        }
    }
}
```

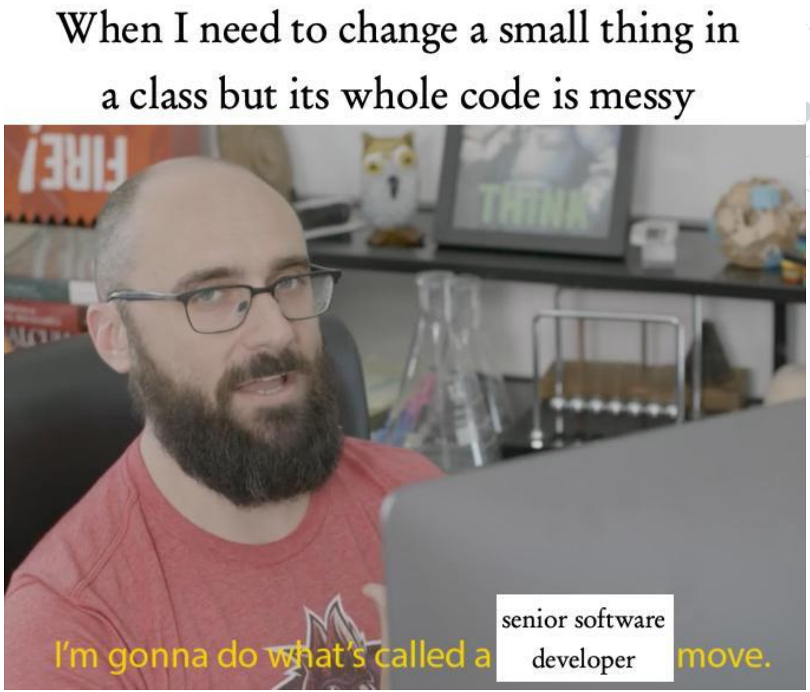
C/C++ SDK Code

```
int main() {
    // Initialize all necessary pins
    stdio_init_all();

    // Initialize LEDs as output
    for (int i = 0; i < 10; i++) {
        gpio_init(LED_BASE_PIN + i);
        gpio_set_dir(LED_BASE_PIN + i, GPIO_OUT);
    }

    // Initialize ADC
    setup_adc();

    while (1) {
        // Read the potentiometer value
        int leds_to_light = read_adc();

        // Control the LEDs based on the ADC value
        control_leds(leds_to_light);

        // Small delay for better visibility of LED changes
        sleep_ms(100);
    }

    return 0;
}
```

Outline :

- Recap.
- Sensors
- Infra-red Implementation
- ADC
- Potentiometer ADC Implementation
- **Why Drivers ?**

The code is a bit messy and long, isn't it??



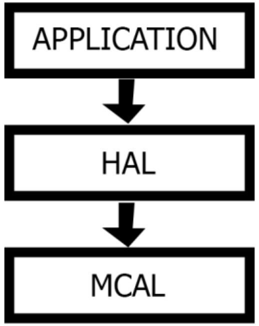
Therefore, we use drivers for each peripheral/sensor/actuator consisting of a header file and a C file for each .

Let's create a driver for the IR sensor

WHY Drivers??

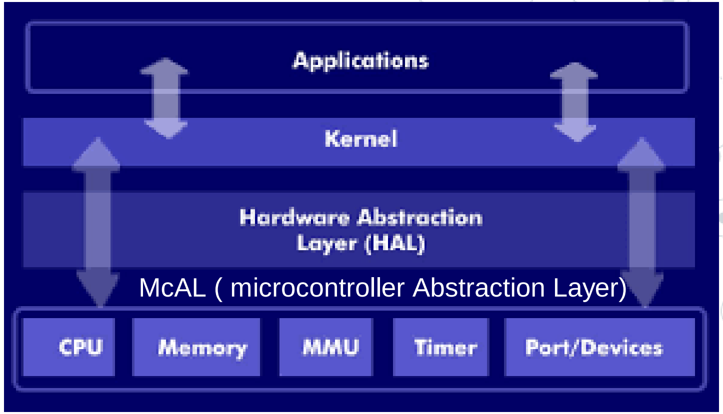
•**Separation of Concerns**: Each driver module typically handles a specific piece of hardware or functionality, isolating concerns to make the code easier to understand.
•**Reusable Code**: Common driver functions can be encapsulated in driver modules, allowing them to be reused across multiple projects, reducing code duplication.
•**Ease of Maintenance**: Changes can be made to a specific module without affecting the rest of the codebase, simplifying maintenance and updates.

Drivers are the building blocks that support the layered Architecture for the embedded system



Embedded System Layers

Layer	Purpose	Example Components
MCAL	Provides low-level access to hardware peripherals	GPIO, ADC, UART drivers
HAL	Provides a higher-level API for hardware interaction	Sensor drivers, communication protocol drivers
Application Layer	Implements application logic and functionality	User interfaces, application-specific tasks



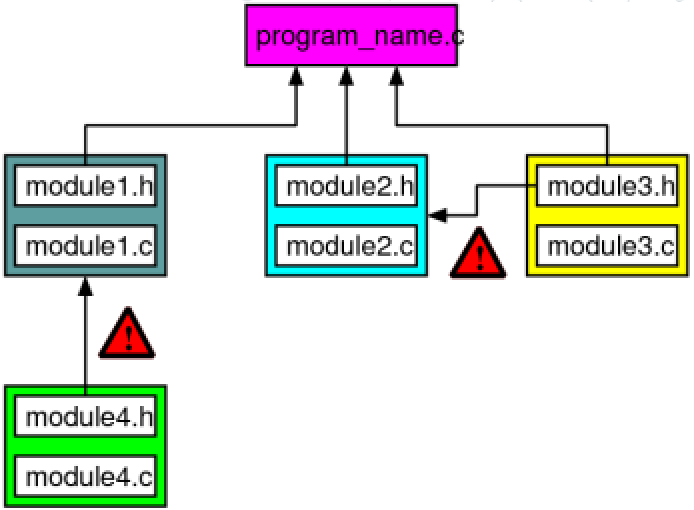
MCAL : Abstracts the hardware details of different microcontrollers. (GPIO.c , GPIO.h)

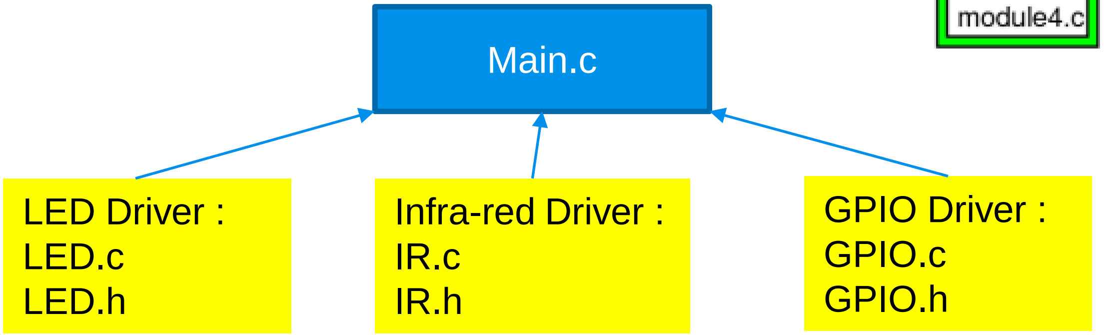
HAL : Sits on top of MCAL , Simplifies hardware interactions and Allows for easy integration of drivers and middleware. (led.c , led.h . IR.c , IR.h) .

Application : Contains application-specific logic and state management. (main.c)

.h file:

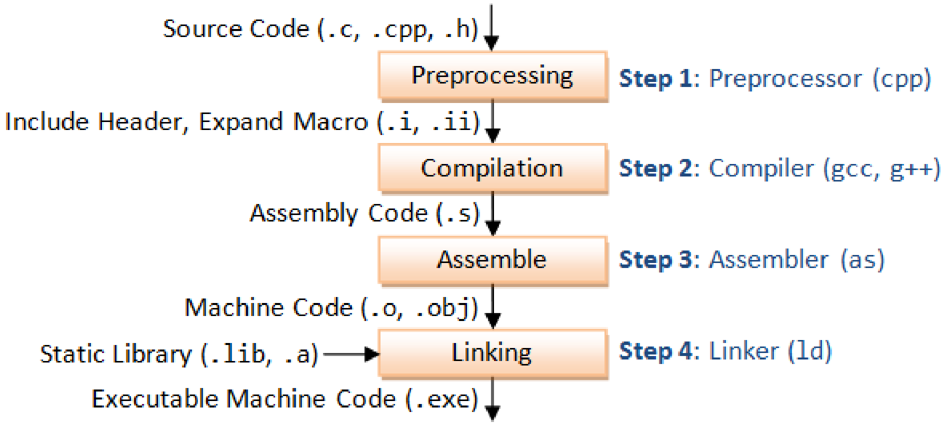
- Declarations of functions and data structures.
- Prevent multiple inclusions using **include guards**.
- Acts as the interface for other .c files.
- Registers names , addresses .
- **.c file**:
- Contains actual code/implementations of the declared functions.
- Links with other .c files during the build process.
- Includes the .h files for function declarations and type definitions.





BACK TO THE COMPILATION PROCESS FLOW

- **Preprocessing**: Handles directives, creates a translation unit.
- **Compilation**: Translates to assembly, checks syntax.
- **Assembly**: Converts to machine code, produces object files.
- **Linking**: Combines object files and libraries into an executable.



infrared_sensor.h

Infrared_sensor.h

HAL layer

```
#ifndef INFRARED_SENSOR_H
#define INFRARED_SENSOR_H
```

Header file guards

```
#include "pico/stdlib.h"
```

It includes GPIO.h and low level microcontroller drivers as ADC , GPIO. (MCAL layer)

```
// Define the default pin for the infrared sensor
#ifndef IR_SENSOR_PIN
#define IR_SENSOR_PIN 15  // Default GPIO pin for infrared sensor
#endif

// Function to initialize the infrared sensor
void infrared_sensor_init(uint sensor_pin);

// Function to read the state of the infrared sensor
bool infrared_sensor_is_triggered(uint sensor_pin);

#endif // INFRARED_SENSOR_H
```

infrared_sensor.c

Infrared_sensor.c

```
#include "infrared_sensor.h"

// Initialize the infrared sensor
void infrared_sensor_init(uint sensor_pin) {
    gpio_init(sensor_pin);          // Initialize the GPIO pin for the sensor
    gpio_set_dir(sensor_pin, GPIO_IN);  // Set the pin direction as input
}

// Check if the infrared sensor is triggered
bool infrared_sensor_is_triggered(uint sensor_pin) {
    return gpio_get(sensor_pin);  // Return the state of the sensor (1 if triggered, 0 otherwise)
}
```

main.c

```c
#include "infrared_sensor.h"

int main() {
    stdio_init_all();

    // Initialize the infrared sensor on GPIO 15
    infrared_sensor_init(IR_SENSOR_PIN);

    while (true) {
        if (infrared_sensor_is_triggered(IR_SENSOR_PIN)) {
            // Do something when the sensor is triggered
            printf("Object detected!\n");
        } else {
            // Do something else when the sensor is not triggered
            printf("No object detected.\n");
        }

        sleep_ms(100);  // Wait for a short delay for debouncing
    }

    return 0;
}
```

Main.c
(Application layer)
Main should contain high level logic

THANK YOU