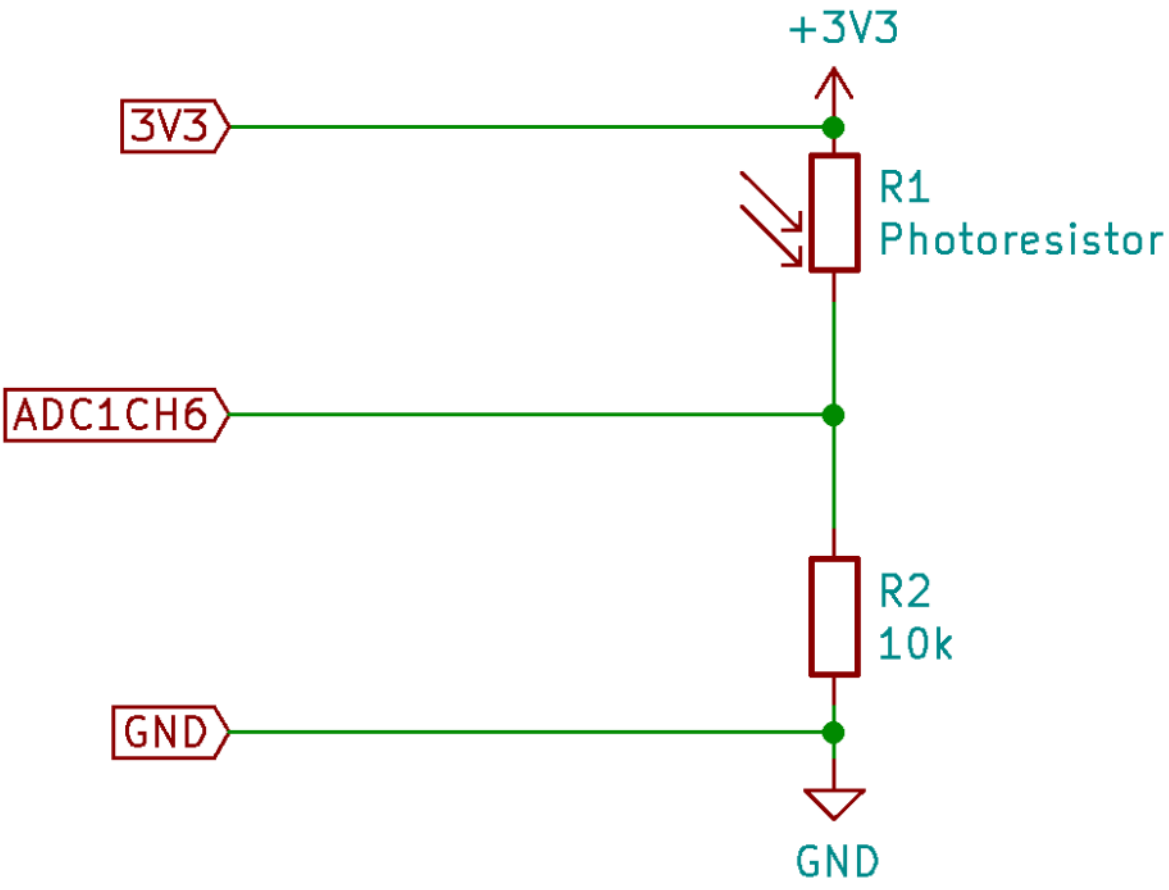
Part 12
-
ESP32 meets Raspberry Pi

Version: 2021-01-14

ESP32 meets Raspberry Pi and vice versa

Often Raspberry Pi computers get unfairly compared with microcontrollers such as ESP32. In fact, they are very different beasts, despite their diminutive similarities. Raspberry Pi has the power of its operating system and better connectivity while ESP32 has better support for analogue devices and faster reaction times. So what happens when you need both in a project? You use both, as it's not too complicated to get a Raspberry Pi computer and ESP32 talking! To demonstrate, we’re going to read the analogue input from a simple photocell using an ESP32 and then send that information to a Raspberry Pi.

Note: I used a Raspberry Pi 4 with Raspbian Buster OS and a DOIT ESP32 DevKit V1 with 36 pins

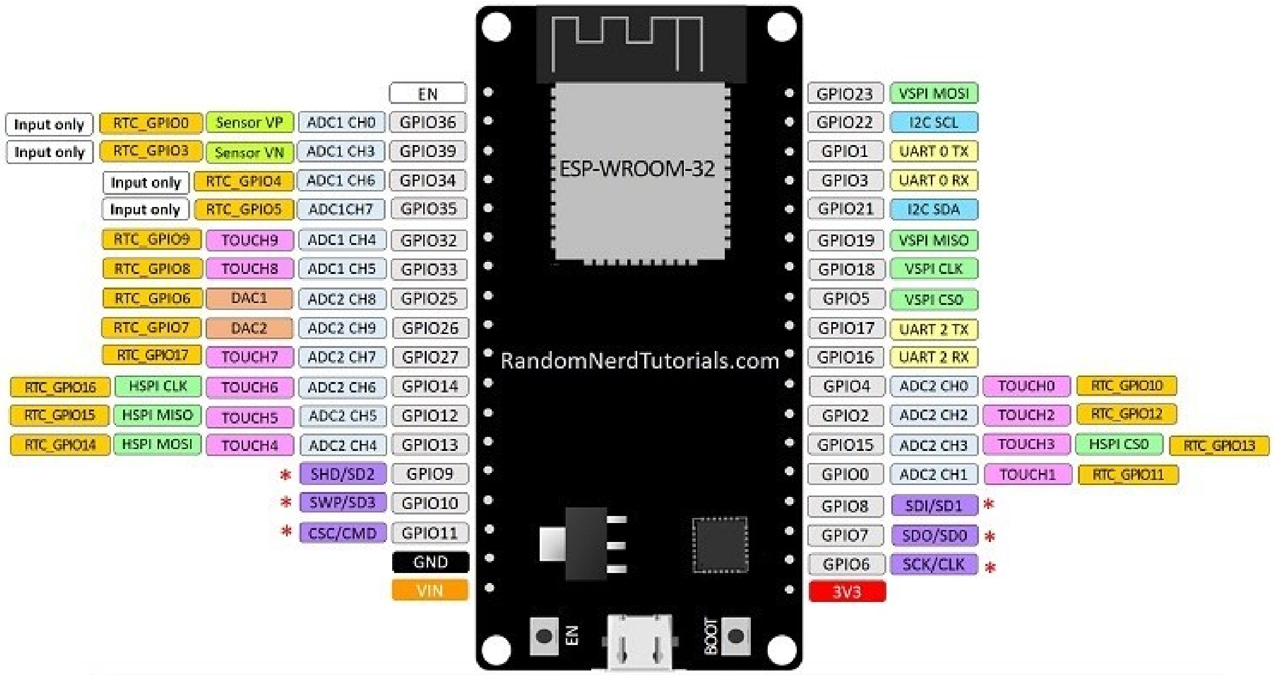


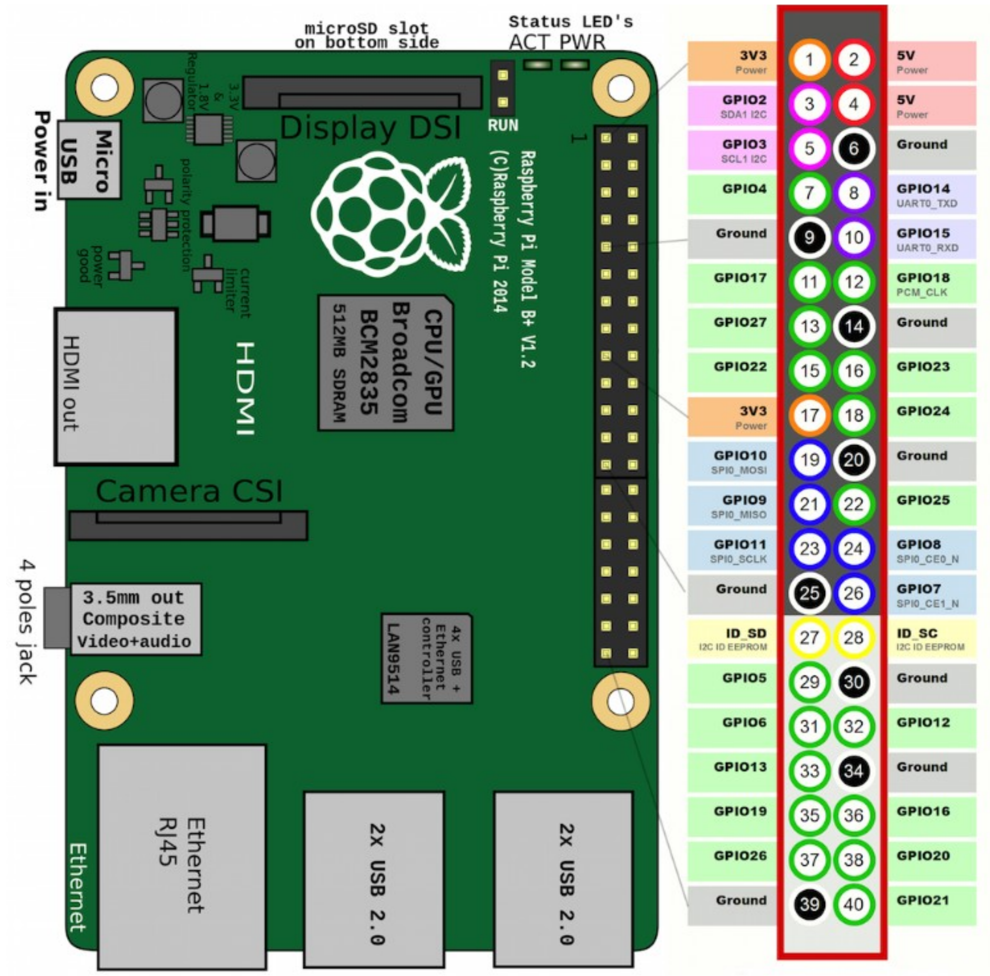
The circuit is powered from the ESP32 3.3V.

ADC1CH6 = GPIO34 on the ESP32.

ESP32 DEVKIT V1 – DOIT

version with 36 GPIOs


EN
Input only RTC_GPIO0 Sensor VP ADC1 CH0 GPIO36
Input only RTC_GPIO3 Sensor VN ADC1 CH3 GPIO39
Input only RTC_GPIO4 ADC1 CH6 GPIO34
Input only RTC_GPIO5 ADC1CH7 GPIO35
RTC_GPIO9 TOUCH9 ADC1 CH4 GPIO32
RTC_GPIO8 TOUCH8 ADC1 CH5 GPIO33
RTC_GPIO6 DAC1 ADC2 CH8 GPIO25
RTC_GPIO7 DAC2 ADC2 CH9 GPIO26
RTC_GPIO17 TOUCH7 ADC2 CH7 GPIO27
RTC_GPIO16 HSPI CLK TOUCH6 ADC2 CH6 GPIO14
RTC_GPIO15 HSPI MISO TOUCH5 ADC2 CH5 GPIO12
RTC_GPIO14 HSPI MOSI TOUCH4 ADC2 CH4 GPIO13
* SHD/SD2 GPIO9
* SWP/SD3 GPIO10
* CSC/CMD GPIO11
GND
VIN
ESP-WROOM-32
RandomNerdTutorials.com
EN
BOOT
GPIO23 VSPI MOSI
GPIO22 I2C SCL
GPIO1 UART 0 TX
GPIO3 UART 0 RX
GPIO21 I2C SDA
GPIO19 VSPI MISO
GPIO18 VSPI CLK
GPIO5 VSPI CS0
GPIO17 UART 2 TX
GPIO16 UART 2 RX
GPIO4 ADC2 CH0 TOUCH0 RTC_GPIO10
GPIO2 ADC2 CH2 TOUCH2 RTC_GPIO12
GPIO15 ADC2 CH3 TOUCH3 HSPI CS0 RTC_GPIO13
GPIO0 ADC2 CH1 TOUCH1 RTC_GPIO11
GPIO8 SDI/SD1 *
GPIO7 SDO/SD0 *
GPIO6 SCK/CLK *
3V3



Just USB

We'll start with the easiest, simplest way of getting our two friends talking. In the case of an ESP32, simply take a USB-2 cable (USB-A ←→ USB-B Micro) and connect the two devices together. The ESP32's serial console is now available to Raspberry Pi OS as `/dev/ttyACM0`. Hence the ESP32 will be powered from the Raspberry Pi's USB port. Build the circuit, then enter the code below and send it to the ESP32.

Note: You could use the Arduino IDE in Raspberry Pi OS Desktop to reprogram the ESP32.

```
//PhotoResistor GPIO34
int lightPin = 34;

void setup()
{
  // Set up serial port
  Serial.begin(115200);
}

void loop()
{
  // Get the current reading
  int lightLevel = analogRead(lightPin);

  // Send it to the serial port
  Serial.println(lightLevel);

  // Wait a bit then repeat
  delay(500);
}
```

Now on the command line (terminal or SSH terminal) from your Raspberry Pi

Make sure pip3 is installed

```
sudo apt -y install python3-pip
```

Install Python serial library and tools

```
sudo pip3 install pyserial
```

Run the miniterm on ttyUSB0 at 115200 baud

```
python3 -m serial.tools.miniterm /dev/ttyUSB0 115200
```

You should see a list of numbers coming in. That's the readings from the photocell. Cover it with your finger and see it change! To quit, press CTRL+] (or Ctrl+$ on AZERTY keyboard).

And here is a small demo Ptyhon program to capture the same input data

```
#!/usr/bin/python
import serial

ser = serial.Serial('/dev/ttyUSB0',115200)

while True:
    readedText = ser.readline()
    print(readedText)

ser.close()
```

Direct UART connection

A USB cable is simple but bulky. If space is an issue, we can do the same job by connecting Raspberry Pi and the ESP32 directly using their serial ports.
Know that the ESP32 outputs operate between 0V and 3.3V. Also the Raspberry Pi does. So you can interconnect directly without any problem.

Note: The ESP32 will be powered from the Raspberry Pi's 5V

So interconnect as follows

Raspberry Pi			ESP32	
Name	***GPIO***	***Pin***	***Name***	***GPIO***
GND		6	GND	
5V		2	VIN	
UART0_RXD	15	10	UART 0 TX	1
UART0_TXD	14	8	UART 0 RX	3

First, run `sudo raspi-config` and select `Interfacing Options`, then enable the serial port but not the serial login/console. Then reboot the Pi.
Build the circuit and then, on Raspberry Pi, run the screen command as before but now pointing to the primary UART which is serial0

```
  python3 -m serial.tools.miniterm /dev/serial0 115200
```

Multple ESP32’s with I^{2}C

I^{2}C is a standard interface available on both Raspberry Pi 4 and ESP32 that allows a direct-wire connection to multiple devices. Follow the wiring below.

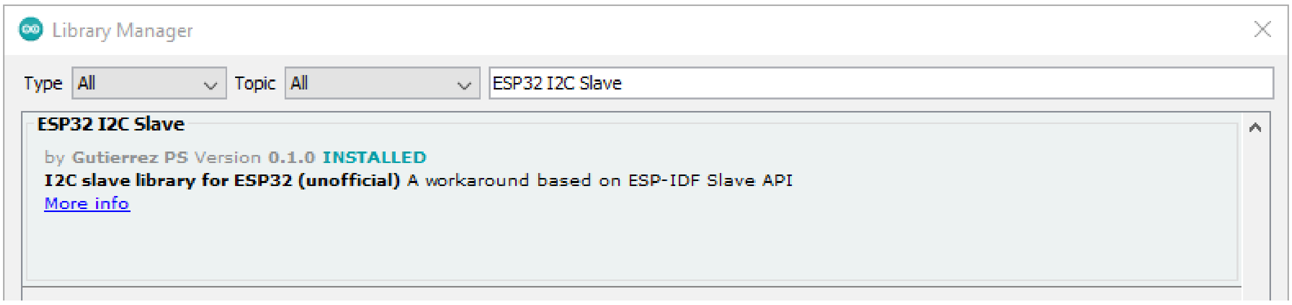
Important to know: I have been testing this with several ESP32’s from 2 different (chinese vendors). With ESP32’s from one vendor, it never worked. I always got a error `121 Remote I/O` but with the ESP32’s from the other vendor (and brand ‘DOIT’), it worked like a charm. So, if it ain’t working for you, you might want to change vendor and/or brand for your ESP32.

Note:

- The ESP32 will be powered from the Raspberry Pi’s 5V
- I no longer use the photoresitor to get some data but a random number to ease wiring

Raspberry Pi			ESP32	
Name	***GPIO***	***Pin***	***Name***	***GPIO***
GND		6	GND	
5V		2	VIN	
SDA1 I2C	2	3	I2C SDA	21
SCL1 I2C	3	5	I2C SCL	22

We need to reprogram the ESP32 and need an extra library for this. The ‘ESP32 I2C Slave’ library can be installed through Tools → Manage Libraries. Search for it and install.



Library Manager

Type All Topic All ESP32 I2C Slave

ESP32 I2C Slave

by **Gutierrez PS** Version **0.1.0** **INSTALLED**
I2C slave library for ESP32 (unofficial) A workaround based on ESP-IDF Slave API
More info

Now reprogram the ESP32 with the code

```
#include <WireSlave.h>

#define SDA_PIN 21
#define SCL_PIN 22
#define I2C_SLAVE_ADDR 0x08

void sndData();
void rcvData(int cnt);

void setup()
{
  delay(2000);

  // Set up serial port
  Serial.begin(115200);

  // Start i2c and set my address to 8
  bool res = WireSlave.begin(SDA_PIN, SCL_PIN, I2C_SLAVE_ADDR);
  if (!res)
  {
    Serial.println("I2C slave init failed");
    while(1) delay(100);
  }

  Serial.println("I2C slave init successful");
  // When a request is made, call this function
  WireSlave.onRequest(sndData);
  WireSlave.onReceive(rcvData);

}

void loop()
{
  static unsigned long lastTime;

  // the slave response time is directly related to how often
  // this update() method is called, so avoid using long delays
  // inside loop()
  WireSlave.update();

  if (millis() - lastTime > 2000)
  {
    Serial.println("Still Alive");
    lastTime = millis();
  }

  // let I2C and other ESP32 peripherals interrupts work
  delay(10);
}

// Send the current reading to the requester
void sndData()
{
  int rndNbr = random(999);
  String str = "RND is ";
  Serial.println(" -> Sent: ");
  Serial.print("     ");
  Serial.print(str);
  Serial.println(rndNbr);

  // use .print to send string data
  WireSlave.print(str);
  WireSlave.print(rndNbr);
  // use .write to send ingeter data
  //WireSlave.write(rndNbr);
}

// executes whenever a complete and valid packet is received from master
void rcvData(int cnt)
{
  Serial.print(" -> Received ");
  Serial.print(cnt);
  Serial.println(" bytes");
  Serial.print("     ");
  // loop through all but the last byte
  while (WireSlave.available())
  {
    // receive byte as a character
    char c = WireSlave.read();
    // receive byte as a integerr
    //int c = WireSlave.read();
    // print the data
    Serial.print(c);
  }
  Serial.println("");
}
```

On your PI, `sudo raspi-config` and select `Interfacing Options`, then enable the I^{2}C.
Reboot your Pi.
Then run these commands to install required libraries. They only need to be run/installed once

```
sudo pip3 install smbus
sudo pip3 install adafruit-blinka
sudo pip3 install adafruit-extended-bus
```

and use your favourite text editor to create `readi2c.py` on Raspberry Pi

```
sudo nano readi2c.py
```

```
from Adafruit_PureIO.smbus import SMBus
import time

# This is the address we gave in the ESP32 - see code
ESP_I2C_address = 0x08

# open I2C bus 1 = RPi3/4
bus = SMBus(1)

# 2 classes to easy the use of I2C data streams
# encode data to be sent
class I2C_Encoder:
    # because ESP Slave I2C library wait for buffer[128] size
    PACKER_BUFFER_LENGTH = (128)

    def __init__(self):
        self._buffer      = [0] * self.PACKER_BUFFER_LENGTH
        self._frame_start = 0x02
        self._frame_end   = 0x04
        self.reset()

    def reset(self):
        # Reset the packing process.
        self._buffer[0]   = self._frame_start
        # field for total lenght on index 1. data starts on field 2
        self._index       = 2
        self._is_written  = False

    def write(self, data: int):
        # write data in prepared buffer
        # do not allow write after .end()
        if self._is_written:
            raise Exception("ERROR: You need to restart process by
using .reset() method before writing to buffer")
        self._buffer[self._index] = data
        self._index += 1

    def end(self):
        # Closes the packet by adding crc8 and length
        # After that, use read()
        # skip field for CRC byte
        self._index += 1
        # add frame end
        self._buffer[self._index] = self._frame_end
        # calc and write total length
        self._index += 1
        self._total_length = self._index
        self._buffer[1] = self._total_length

        # ignore crc and end byte
        payload_range = self._total_length - 2
        # ignore start and length byte [2:payload_range]
        self._buffer[self._index - 2] = crc8(self._buffer[2:payload_range])
        self._is_written = True
        return self._buffer

    def read(self):
        # Read the packet
        if not self._is_written:
            raise Exception("ERROR: You need to finish process by
using .end() method before read buffer")
        return self._buffer

# decodes data received from I2C data stream
class I2C_Decoder:
    error_codes = {
        "INVALID_CRC":    1,
        "INVALID_LENGTH": 2,
        "INVALID_START":  3,
        "INVALID_END":    4,
    }
    error_decodes = {
        1: "INVALID_CRC",
        2: "INVALID_LENGTH",
        3: "INVALID_START",
        4: "INVALID_END",
    }

    def __init__(self):
        self._debug = False
        self._frame_start = 0x02
        self._frame_end   = 0x04

    def write(self, stream):
        # get the i2c data from slave
        # clear any previous
        self._buffer      = []
        self._last_error  = None
        data = list(stream)
        # check if start and end bytes are correct
        if data[0] != self._frame_start:
            self._last_error = self.error_codes["INVALID_START"]
            raise Exception("ERROR: invalid start byte")
        self._length = data[1]
        if data[self._length-1] != self._frame_end:
            self._last_error = self.error_codes["INVALID_END"]
            raise Exception("ERROR: invalid end byte")

        # check if provided crc8 is good
        # ignore start, length, crc and end byte
        crc = crc8(data[2:self._length-2])
        if crc != data[self._length-2]:
            self._last_error = self.error_codes["INVALID_CRC"]
            raise Exception("ERROR: Unpacker invalid crc8")
        self._data = data[2:self._length-2]
        # create string
        answer = ""
        for c in self._data:
            answer = answer + chr(c)
        return answer

    def get_last_error(self):
        """
        @brief get the last error code and message
        @return list [error_code, error_text]
        """
        return self._last_error, self.error_decodes[self._last_error]

# routine to calculate CRC8
def crc8(data: list):

    crc = 0
    for _byte in data:
        extract = _byte
        for j in range(8, 0, -1):
            _sum = (crc ^ extract) & 0x01
            crc >>= 1
            if _sum:
              crc ^= 0x8C
            extract >>= 1
    return crc

# read data from EPS32
def read_from_esp32(i2caddress: hex, size: int):

    decoder = I2C_Decoder()

    try:
        # get data sent by ESP32, in raw format.
        # read 25 bytes
        stream = bus.read_bytes(i2caddress, size)
        # convert to a list to ease handling
        data = decoder.write(stream)
        return data

    except Exception as e:
        print("ERROR: {}".format(e))

# write data to ESP32
def write_to_esp32(i2caddress: hex, data: str):

    encoder = I2C_Encoder()
    try:
        # only if there is data
        if len(data) > 0:
            for c in data:
                encoder.write(ord(c))
        stream = encoder.end()

        # send stream to slave
        bus.write_bytes(i2caddress, bytearray(stream))

    except Exception as e:
        print("{}".format(e))

if __name__ == "__main__":

    while True:
        # request ESP32 to get data
        write_to_esp32(ESP_I2C_address, "")
        # wait to process the request
        time.sleep(0.1)
        # Get the value from the ESP32
        data = read_from_esp32(ESP_I2C_address, 25)
        print(data)

        time.sleep(2)
```

Once written and saved, exit and run the code

```
  python3 readi2c.py
```

In this scenario, the Python script ‘requests’ the data and the ESP32 responds instantly with the reading.

To interconnect multiple ESP32, just change the `I2C_SLAVE_ADDR` per ESP32. And, of course, create a loop in your Python code the request data from every ESP32.

Remember!
ESP32 runs at 5V and Raspberry Pi at 3.3V (3V3). The ESP32 can destroy your Raspberry Pi if wired together incorrectly.