UNIT PULSAR H2

Release 0.0.1

Cesar Bautista

Oct 14, 2025

CONTENTS

1 PULSAR H2 Development Board **3**
- 1.1 Introduction . . . 3
- 1.2 Features . . . 3

2 Desktop Environment **9**
- 2.1 Install the required software . . . 9
- 2.2 Python 3.7 or later . . . 9
- 2.3 Git . . . 10
- 2.4 MinGW . . . 11
- 2.5 Visual Studio Code . . . 15
- 2.6 Arduino IDE Installation . . . 16
- 2.7 Thonny IDE Installation . . . 16

3 Installing packages - Micropython **17**
- 3.1 Installation Guide Using MIP Library . . . 17
- 3.2 DualMCU Library . . . 18
- 3.3 Libraries available . . . 20

4 ESP-IDF Getting Started **21**
- 4.1 Installation Steps . . . 21
- 4.2 Customizing the Installation Path . . . 22
- 4.3 First Steps with ESP-IDF . . . 22

5 Development board **25**
- 5.1 Squematic Diagram . . . 25
- 5.2 Pinout distribution . . . 25

6 General Purpose Input/Output (GPIO) Pins **27**
- 6.1 Working with LEDs on ESP32-H2 . . . 27

7 Analog to Digital Conversion **29**
- 7.1 ADC Definition . . . 29
- 7.2 ESP32-H2 ADC Channels (Official Documentation) . . . 29
- 7.3 ADC Pin Status on PULSAR H2 . . . 30
- 7.4 Summary Table: Usable ADC Pins . . . 30
- 7.5 Class ADC . . . 31
- 7.6 ADC Pin Usage Examples . . . 31
- 7.7 Reading Values . . . 31
- 7.8 Example Code . . . 32

8 I2C (Inter-Integrated Circuit) **35**

8.1 I2C Overview 35
8.2 Pinout Details 35
8.3 I2C Features on ESP32-H2 36
8.4 Scanning for I2C Devices 36
8.5 SSD1306 Display 38

9 SPI (Serial Peripheral Interface) 43
9.1 SPI Overview 43
9.2 SDCard SPI 43

10 WS2812 Control 51
10.1 Code Example 51

11 Communication 55
11.1 Bluetooth 55
11.2 Serial 56

12 How to Generate an Error Report 57
12.1 Steps to Create an Error Report 57
12.2 Review and Follow-Up 57

13 Indices and tables 59

Index 61

Note: You are reading the most recent version available.

Documentation for the UNIT PULSAR H2 Development Board. This board is a versatile ultra-low-power microcontroller with advanced connectivity and peripheral features, making it suitable for a wide range of IoT and embedded applications.

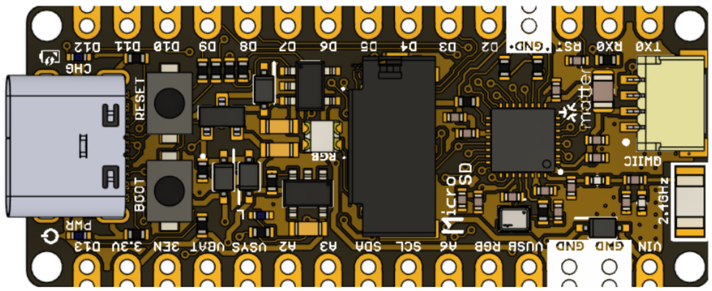


Fig. 1: PULSAR H2 Development Board

CHAPTER

ONE

PULSAR H2 DEVELOPMENT BOARD

1.1 Introduction

This guide will help you get started with the **PULSAR H2** development board. The **PULSAR H2** is a development board based on the ESP32H2 microcontroller. It is designed for prototyping and developing IoT applications. The board features a variety of interfaces, including GPIO, I2C, SPI, UART, and more. It also has built-in support for Zigbee, Thread (802.15.4), and Bluetooth connectivity.

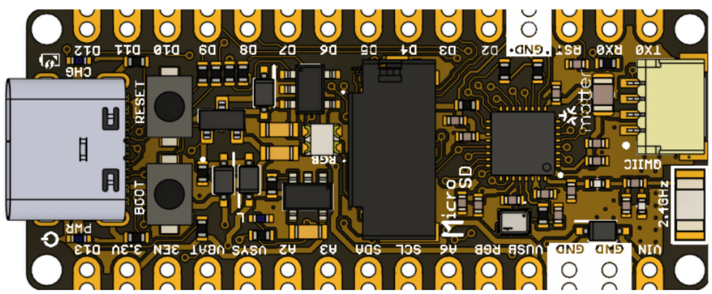


Fig. 1.1: **PULSAR H2** Development Board

1.2 Features

1.2.1 CPU

Espressif ESP32-H2FH4S

- Single-core 32-bit RISC-V processor
- Up to 96 MHz operating frequency
- Four-stage pipeline

1.2.2 Internal Memory

- **320 KB** of internal SRAM
- **128 KB** of ROM (for boot and system functions)
- **4 MB** of integrated SPI Flash (in the ESP32-H2FH4 module)
- **4 KB** LP Memory
- **16 KB** cache

1.2.3 Wireless Connectivity

- **Bluetooth® 5.0 LE**, supports LE 2M, LE Coded PHY, Extended Advertising, and Advertising Extensions
- **IEEE 802.15.4** for Zigbee and Thread, with support for Matter over Thread

1.2.4 Peripheral Interfaces

- **19 programmable GPIOs** (including GPIO8, GPIO9, and GPIO25 as strapping pins)
- **12-bit SAR ADC** (up to 5 channels)
- Temperature sensor
- SPI, UART, I^{2}C, I^{2}S, PWM, RMT
- **USB 2.0 Full-Speed** (with integrated Serial/JTAG controller and PHY)
- General-purpose SPI, UART (×2), and I^{2}C (×2) interfaces
- RMT with up to 2 transmit channels and 2 receive channels
- **LED PWM controller** (up to 6 channels)
- **Motor Control PWM (MCPWM)**
- Pulse Count Controller
- **General DMA controller** (3 TX channels, 3 RX channels)
- Parallel I/O (PARLIO) controller
- SoC Event Task Matrix (ETM)
- **Two TWAI® Controllers** (compatible with ISO 11898-1 / CAN 2.0)
- Two 54-bit general-purpose timers
- 52-bit system timer
- Three watchdog timers
- SDIO, JTAG, GPIO

1.2.5 Built-in Security

- **Secure Boot** – Ensures firmware integrity during startup
- **Flash Encryption** – Provides secure memory encryption and decryption
- **4096-bit OTP** (One-Time Programmable memory), with up to 1792 bits available for user data

1.2.6 Cryptographic Hardware Acceleration

- **AES-128/256** (FIPS PUB 197) – Supports ECB, CBC, CFB, OFB, and CTR modes (FIPS PUB 800-38A)
- **SHA Accelerator** (FIPS PUB 180-4)
- **RSA Accelerator**
- **ECC Accelerator**
- **ECDSA** (Elliptic Curve Digital Signature Algorithm)
- **HMAC** (Hash-based Message Authentication Code)
- **Digital Signature Engine**
- Access Permission Management (APM)
- Random Number Generator (RNG)
- Power Glitch Detector

1.2.7 Power Management and Operating Voltage

- **I/O Operating Voltage**: 3.3 V
- **Ultra-Low Power Consumption** – Designed for energy-efficient applications requiring extended battery life
- Fine-resolution power control through adjustable clock frequency, duty cycle, RF operating modes, and individual power control of internal components
- **Four Power Modes** optimized for different operation scenarios:
  - Active, Modem-sleep, Light-sleep, and Deep-sleep
- **Power Consumption in Deep-sleep Mode**: 7 µA
- Independent RTC (Real-Time Clock) for data and event retention during Deep-sleep mode
- LP (Low-Power) memory remains powered on in Deep-sleep mode

1.2.8 Antenna

- **Integrated PCB antenna** (no external antenna required)

1.2.9 Storage

- Integrated **microSD card slot** via SPI for data logging, multimedia storage, and firmware updates
- Connected to GPIO0, GPIO4, GPIO5, and GPIO25

1.2.10 Power Management

- **Vin**: Up to 6V via pin header
- **USB-C powered** (5V input)
- **VUSB Output**: Available
- **3.3V AP2112K 3.3V LDO Regulator** (max input 6V): 350 mA nominal current, up to 600 mA peak with thermal protection
- Supports **LiPo battery charging** with an onboard power management circuit. Charging current: 200 mA

1.2.11 Interfaces and Connectors

- **1 × I2C JST-SH (1.0 mm pitch)**: Qwiic-compatible connector wired to GPIO12 and GPIO22 for Low-Power I2C
- **1 × microSD Card Holder**
- **1 × Auxiliary Battery Connector** (optional): Supports both 2.0 mm and 1.25 mm pitch options
- **1 × USB Type-C Connector**
- **2 × 15-pin Header Connectors**: With castellated holes for easy surface mounting

1.2.12 Communication and Connectivity

- **USB-C connector** for programming and power
- **Reset button** and **Flash/Boot button** for manually entering flash mode

1.2.13 LED Indicators

- **Green or Red PWR LED** (0603) – Power indication
- **Orange CHG LED** (0603) – LiPo charging status
- **Pink BLINK LED** (0603, GPIO4) – User-programmable
- **WS2812 RGB LED** (2020) – Fully addressable for status or visual feedback connected to GPIO8

1.2.14 Software Support

- **Arduino IDE** (official Uelectronics-ESP32 Arduino Package)
- **ESP-IDF** for advanced native development
- **MicroPython** and **CircuitPython** support
- **PlatformIO / VS Code** for professional development

1.2.15 Applications

- **Smart Home** (Matter, Thread)
- **Home and Industrial Automation** (including CAN Bus and low-power systems)
- **IoT Prototyping and Embedded Development**
- **Multi-radio Devices and Mesh Communication**
- **Robotics and Sensor Networks**

CHAPTER

TWO

DESKTOP ENVIRONMENT

The environment setup is the first step to start working with the PULSAR H2 board. The following steps will guide you through the setup process.

1. Install the required software
2. Set up the development environment
3. Install the required libraries
4. Set up the board

2.1 Install the required software

The following software is required to start working with the PULSAR H2 board:

1. **Python 3.7 or later**: Python is required to run the scripts and tools provided by the PULSAR H2 board.
2. **Git**: Git is required to clone the PULSAR H2 board repository.
3. **MinGW**: MinGW is a native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications.
4. **Visual Studio Code**: Visual Studio Code is a code editor that is required to write and compile the code.

This section will guide you through the installation process of the required software.

2.2 Python 3.7 or later

Python is a programming language that is required to run the scripts and tools,

To install Python, follow the instructions below:

1. Download the Python installer from the:
2. Run the installer and follow the instructions.

Attention: Make sure to check the box that says "Add Python to PATH" during the installation process.

Open a terminal and run the following command to verify the installation:

Fig. 2.1: Add python to PATH

```
python --version
```

If the installation was successful, you should see the Python version number.

2.3 Git

Git is a version control system that is required to clone the repositories in general. To install Git, follow the instructions below:

1. Download the Git installer from the

2. Run the installer and follow the instructions.
3. Open a terminal and run the following command to verify the installation:

```
git --version
```

If the installation was successful, you should see the Git version number.

2.4 MinGW

MinGW is a native Windows port of the GNU Compiler Collection (GCC), with freely distributable import libraries and header files for building native Windows applications. MinGW provides a complete Open Source programming toolset that is suitable for the development of native Windows applications, and which do not depend on any 3rd-party C-Runtime DLLs. MinGW, being Minimalist, does not, and never will, attempt to provide a POSIX runtime environment for POSIX application deployment on MS-Windows. If you want POSIX application deployment on this platform, please consider Cygwin instead.

To install MinGW, follow the instructions below:

1. Download the MinGW installer from the

2. Run the installer and follow the instructions.

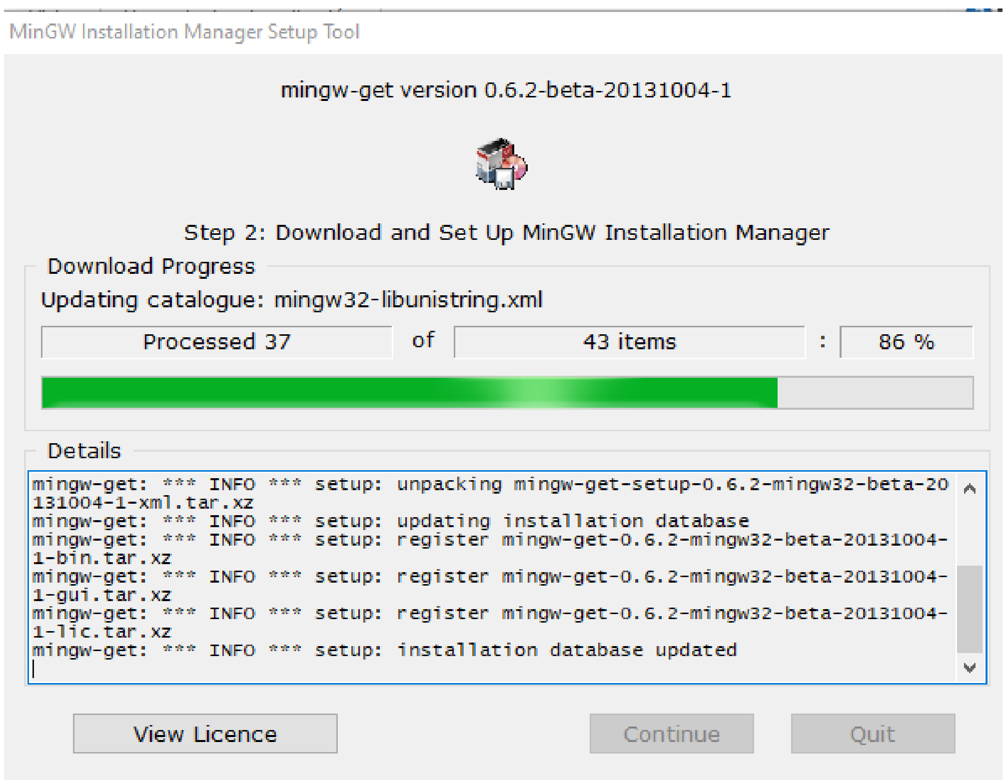
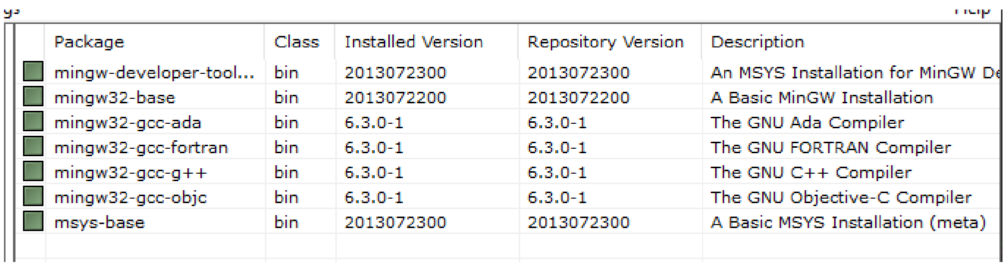


Fig. 2.2: MinGW installer

Note: During the installation process, make sure to select the following packages:

- mingw32-base
- mingw32-gcc-g++
- msys-base

3. Open a terminal and run the following command to verify the installation:



Package	Class	Installed Version	Repository Version	Description
mingw-developer-tool...	bin	2013072300	2013072300	An MSYS Installation for MinGW De
mingw32-base	bin	2013072200	2013072200	A Basic MinGW Installation
mingw32-gcc-ada	bin	6.3.0-1	6.3.0-1	The GNU Ada Compiler
mingw32-gcc-fortran	bin	6.3.0-1	6.3.0-1	The GNU FORTRAN Compiler
mingw32-gcc-g++	bin	6.3.0-1	6.3.0-1	The GNU C++ Compiler
mingw32-gcc-objc	bin	6.3.0-1	6.3.0-1	The GNU Objective-C Compiler
msys-base	bin	2013072300	2013072300	A Basic MSYS Installation (meta)

Fig. 2.3: MinGW installation

```
mingw --version
```

If the installation was successful, you should see the MinGW version number.

2.4.1 Environment Variable Configuration

Remember that for Windows operating systems, an extra step is necessary, which is to open the environment variable -> Edit environment variable:

```
C:\MinGW\bin
```

2.4.2 Locate the file

After installing MinGW, you will need to locate the *mingw32-make.exe* file. This file is typically found in the *C:/MinGW/bin* directory. Once located, rename the file to *make.exe*.

2.4.3 Rename it

After locating *mingw32-make.exe*, rename it to *make.exe*. This change is necessary for compatibility with many build scripts that expect the command to be named *make*.

> **Warning:** If you encounter any issues, create a copy of the file and then rename the copy to *make.exe*.

2.4.4 Add the path to the environment variable

Next, you need to add the path to the MinGW bin directory to your system's environment variables. This allows the *make* command to be recognized from any command prompt.

1. Open the Start Search, type in “env”, and select “Edit the system environment variables”.
2. In the System Properties window, click on the “Environment Variables” button.
3. In the Environment Variables window, under “System variables”, select the “Path” variable and click “Edit”.
4. In the Edit Environment Variable window, click “New” and add the path:

   ```
   C:\MinGW\bin
   ```

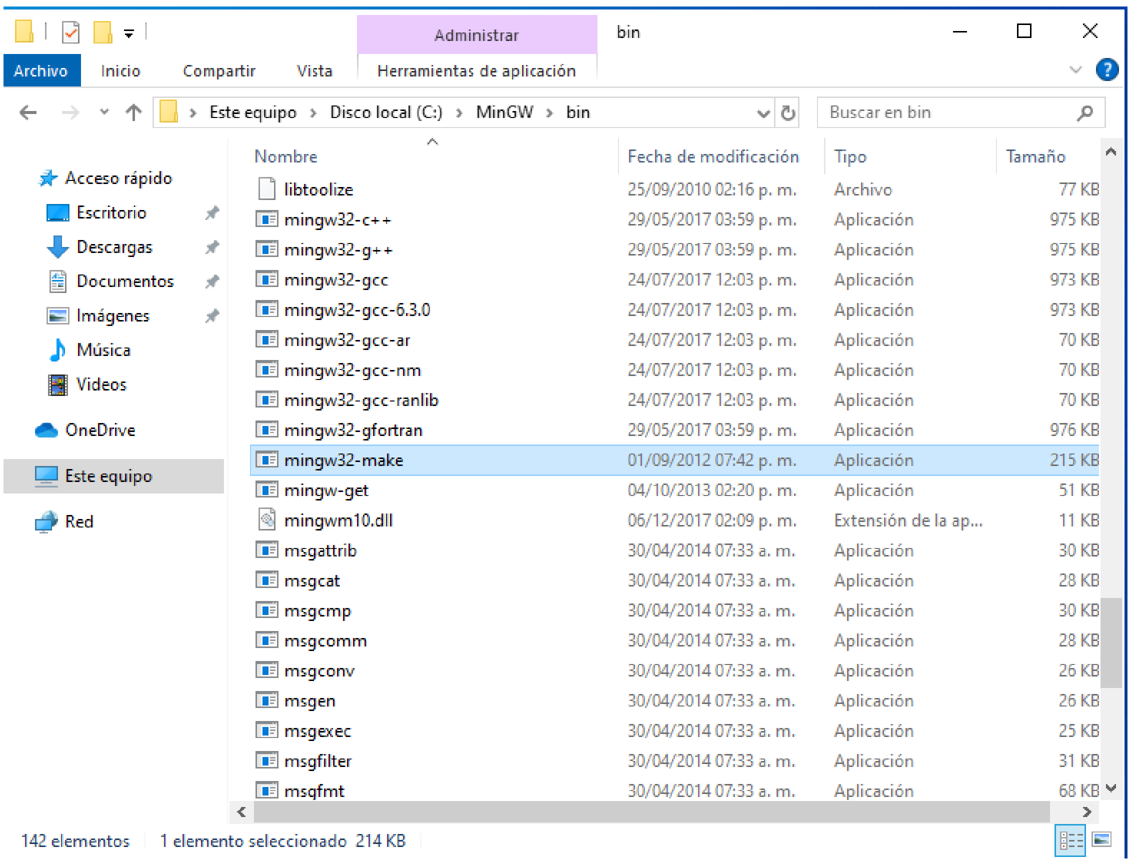


Fig. 2.4: Locating the *mingw32-make.exe* file

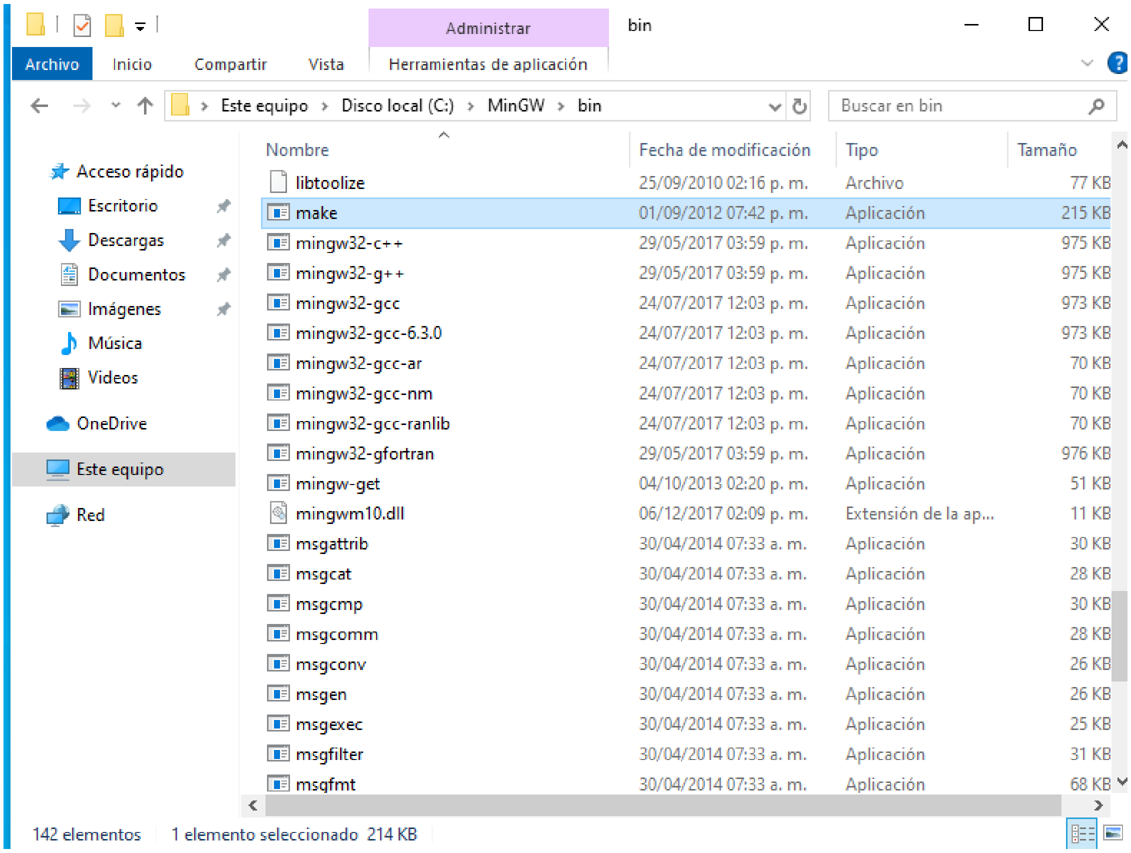


Fig. 2.5: Renaming *mingw32-make.exe* to *make.exe*

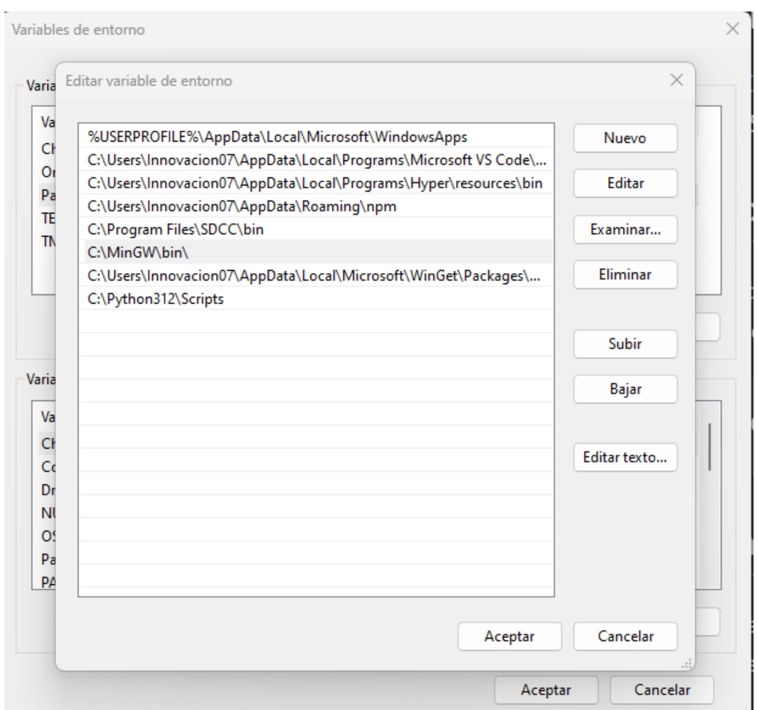


Fig. 2.6: Adding MinGW bin directory to environment variables

2.5 Visual Studio Code

Visual Studio Code is a code editor that is required to write and compile the code.

To install Visual Studio Code, follow the instructions below:

1. Download the Visual Studio Code installer from the

2. Run the installer and follow the instructions.

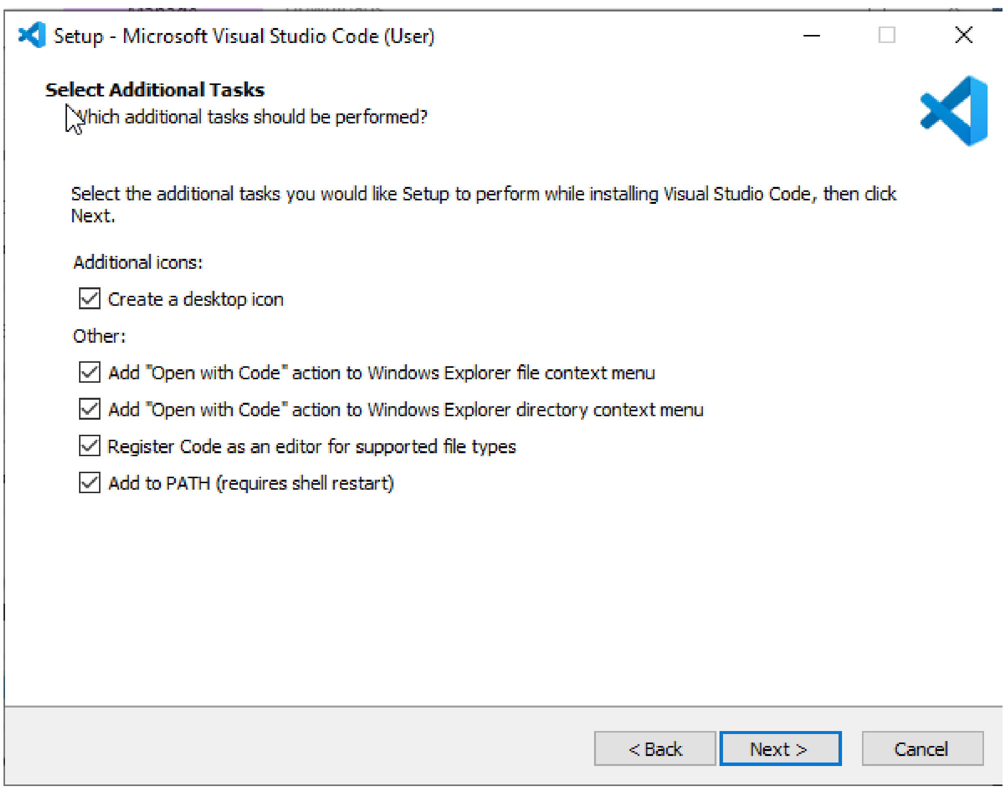


Fig. 2.7: Visual Studio Code installer

Note: During the installation process, make sure to check the box that says "Open with Code".

3. Open a terminal and run the following command to verify the installation:

```
code --version
```

4. Install extensions for Visual Studio Code:

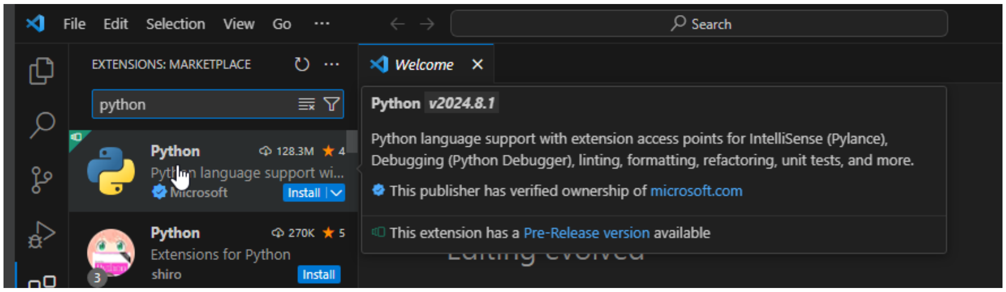


Fig. 2.8: Visual Studio Code extensions

2.6 Arduino IDE Installation

The Arduino IDE is a popular open-source platform for building and programming microcontroller-based projects. It provides a user-friendly interface and a wide range of libraries to simplify the development process.

To install the Arduino IDE, follow the instructions for your operating system in the

2.7 Thonny IDE Installation

Thonny is a Python IDE that is designed for beginners. It provides a simple interface and built-in support for MicroPython, making it an excellent choice for programming the PULSAR H2 board.

Follow the instructions for your operating system in the

CHAPTER

THREE

INSTALLING PACKAGES - MICROPYTHON

This section will guide you through the installation process of the required libraries using the pip package manager.

3.1 Installation Guide Using MIP Library

Note: The *mip* library is utilized to install other libraries on the PULSAR H2 board.

Important: The ESP32-H2 does not support Wi-Fi. For library installation, you'll need to use alternative methods or a gateway device with Wi-Fi connectivity.

3.1.1 Requirements

- ESP32H2 device (PULSAR H2)
- Thonny IDE
- Computer with internet access for library download
- USB connection for file transfer

3.1.2 Installation Instructions

Follow the steps below to install the *max1704x.py* library:

3.1.3 Alternative Installation Methods

Since the ESP32-H2 doesn't support Wi-Fi, consider these alternatives for library installation:

1. **Pre-download libraries**: Download libraries on a computer with internet access and transfer them to the ESP32-H2 via USB.
2. **Use a gateway device**: Use another microcontroller with Wi-Fi capability as a bridge to download libraries.
3. **Manual installation**: Download the library files manually and copy them to the ESP32-H2's filesystem.

```
# Example: Manual library installation
# Download these libraries manually and copy to your ESP32-H2:
# - max1704x.py from UNIT-Electronics/MAX1704X_lib
# - oled.py from compatible micropython libraries
# - sdcard.py for SD card support

# After copying files manually:
import max1704x
import oled
import sdcard
```

3.2 DualMCU Library

Firstly, you need install Thonny IDE. You can download it from the Thonny website.

1. Open Thonny.
2. Navigate to **Tools** -> **Manage Packages**.
3. Search for `dualmcu` and click **Install**.

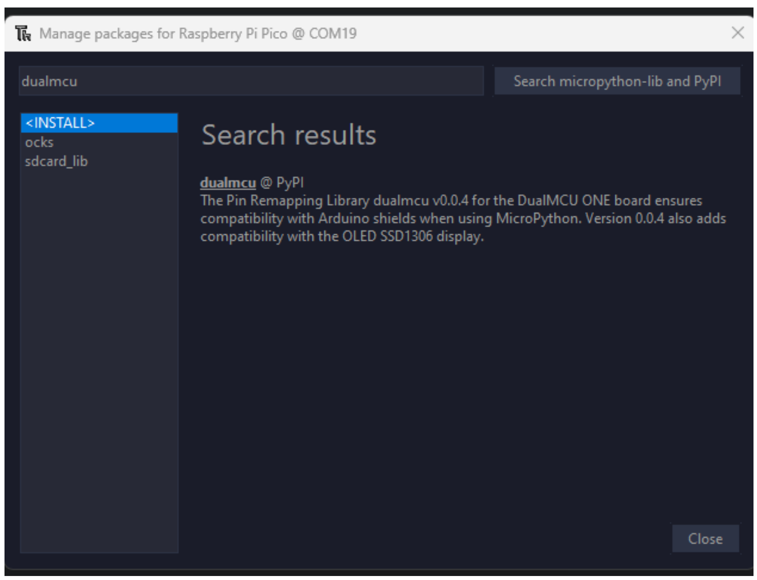


Fig. 3.1: DualMCU Library

4. Successfully installed the library.

Alternatively, download the library from dualmcu.py.

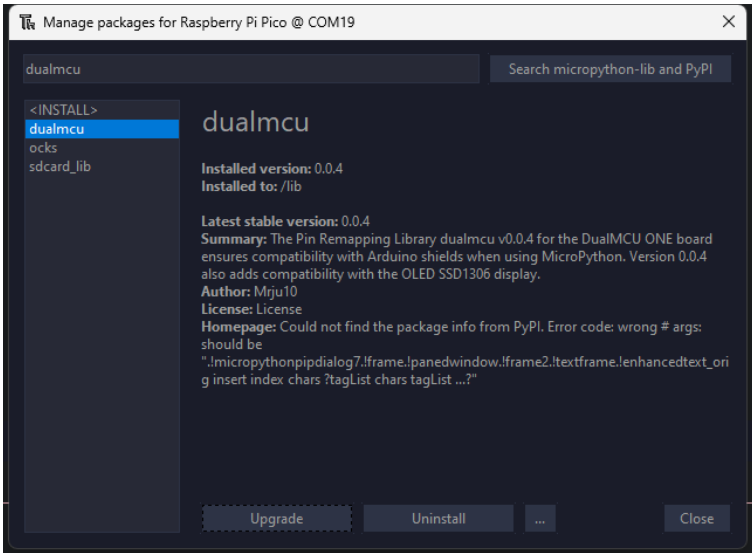


Fig. 3.2: DualMCU Library Successfully Installed

3.2.1 Usage

The library provides a set of tools to help developers work with the DualMCU ONE board. The following are the main features of the library:

- **I2C Support**: The library provides support for I2C communication protocol, making it easy to interface with a wide range of sensors and devices.
- **Arduino Shields Compatibility**: The library is compatible with Arduino Shields, making it easy to use a wide range of shields and accessories with the DualMCU ONE board.
- **SDcard Support**: The library provides support for SD cards, allowing developers to easily read and write data to SD cards.

Examples of the library usage:

```
import machine
from dualmcu import *

i2c = machine.SoftI2C( scl=machine.Pin(22), sda=machine.Pin(21))

oled = SSD1306_I2C(128, 64, i2c)

oled.fill(1)
oled.show()

oled.fill(0)
oled.show()
oled.text('UNIT', 50, 10)
oled.text('ELECTRONICS', 25, 20)

oled.show()
```

3.3 Libraries available

- Dualmcu : The library provides a set of tools to help developers work with the DualMCU ONE board. The library is actively maintained and updated to provide the best experience for developers working with the DualMCU ONE board. For more information and updates, visit the *dualmcu GitHub repository*`
- Ocks : The library provides support for I2C communication protocol.
- SDcard-lib : The library provides support for SD cards, allowing developers to easily read and write data to SD cards; all rights remain with the original author.

The library is actively maintained and updated to provide the best experience for developers working with the DualMCU ONE board.

CHAPTER

FOUR

ESP-IDF GETTING STARTED

The ESP-IDF (Espressif IoT Development Framework) is the official development framework for ESP32 series chips. It provides a comprehensive suite of tools, libraries, and APIs to facilitate application development for ESP32 devices.

This section offers a step-by-step guide to setting up the ESP-IDF environment for the ESP32-H2 chip, including installation instructions and basic usage examples. While the focus is on the ESP32-H2, the guidelines are generally applicable to other ESP32 chips.

Supported Environment: Ubuntu 20.04 or later.

For users on other operating systems, please consult the official ESP-IDF documentation for platform-specific instructions.

Note: **ESP-IDF** is compatible with Windows and macOS, but the installation process may differ. Refer to the official documentation for detailed instructions.

Attention: A stable internet connection is required during installation, as some steps involve downloading necessary files.

4.1 Installation Steps

1. **Install Prerequisites** Ensure all required dependencies are installed. Execute the following commands in a terminal:

```
sudo apt-get update
sudo apt-get install git wget flex bison gperf python3 python3-pip python3-
↪setuptools python3-venv cmake ninja-build ccache libffi-dev libssl-dev dfu-util␣
↪device-tree-compiler
```

2. **Clone the ESP-IDF Repository** Clone the ESP-IDF repository from GitHub. Optionally, specify a particular version or branch.

```
git clone https://github.com/espressif/esp-idf.git
```

3. **Set Up the Environment** Navigate to the cloned ESP-IDF directory and execute the setup script to configure environment variables.

```
cd esp-idf
./install.sh
. ./export.sh
```

Note: To install tools for all supported chips, use the following command:

```
./install.sh --all
```

4. **Install Additional Tools** For ESP32-H2-specific tools, run:

```
./install.sh --esp32h2
```

Note: The *install.sh* script downloads and installs the required tools and dependencies for the ESP32-H2 chip. The duration depends on your internet speed.

5. **Verify Installation** Confirm the installation by checking the ESP-IDF version:

```
idf.py --version
```

4.2 Customizing the Installation Path

To customize the installation path of ESP-IDF, set the *IDF_PATH* environment variable. For example:

```
export IDF_PATH=/path/to/your/esp-idf
. $IDF_PATH/export.sh
. $IDF_PATH/install.sh
```

Note: Replace */path/to/your/esp-idf* with the desired installation directory. This ensures the *IDF_PATH* variable points to the correct location, and the *export.sh* and *install.sh* scripts are executed from there.

4.3 First Steps with ESP-IDF

1. **Create a New Project** Create a directory for your ESP-IDF project and navigate to it:

```
mkdir my_project
cd my_project
```

2. **Generate a Basic Application** Use the *idf.py* tool to create a basic application template:

```
idf.py create-project my_app
```

3. **Build the Project** Navigate to the project directory and build the application:

```
cd my_app
idf.py build
```

4. **Flash the Application** Connect your ESP32-H2 board to your computer and flash the application:

```
idf.py -p /dev/ttyUSB0 flash
```

5. **Monitor the Output** Monitor the output from the ESP32-H2 board:

```
idf.py -p /dev/ttyUSB0 monitor
```

6. **Modify the Code** Edit the code in the *main* directory of your project. The main application file is typically named *main.c* or *main.cpp*. After making changes, rebuild and flash the project.

7. **Clean the Project** To remove all build artifacts, run:

```
idf.py fullclean
```

8. **Update ESP-IDF** To update ESP-IDF to the latest version, navigate to the ESP-IDF directory and execute:

```
git pull
./install.sh
. ./export.sh
```

9. **Uninstall ESP-IDF** To uninstall ESP-IDF, delete the cloned repository and unset related environment variables:

```
rm -rf esp-idf
unset IDF_PATH
unset PATH
unset LD_LIBRARY_PATH
unset PYTHONPATH
unset CMAKE_PREFIX_PATH
```

10. **Explore ESP-IDF Examples** The ESP-IDF repository includes numerous example projects demonstrating various features. These can be found in the *examples* directory. Copy and modify any example project as needed.

11. **Refer to ESP-IDF Documentation** For comprehensive information, including API references and guides, visit the official ESP-IDF documentation: ESP-IDF Documentation.

12. **Join the ESP-IDF Community** For assistance or discussions, join the ESP-IDF community on GitHub or the Espressif Community Forum. The community is active and provides support for various ESP32 development topics.

CHAPTER

FIVE

DEVELOPMENT BOARD

5.1 Squematic Diagram

5.2 Pinout distribution

The following table provides the pinout details for the **PULSAR H2** board.

Arduino Nano Pin	Arduino Nano Description	**PULSAR H2**	ESP32-H2 GPIO
1	D13 (SCK/LED)	D13/SCK/	GPIO4
2	3.3V	3.3V	3.3V
3	AREF	EN_3V3	NC
4	A0 (Analog)/D14	VBAT	NC
5	A1 (Analog)/D15	VSYS	•
6	A2 (Analog)/D16	A2/D16	GPIO2
7	A3 (Analog)/D17	A3/D17	GPIO3
8	A4/(SDA)	(SDA)/D18	GPIO12
9	A5/(SCL)	(SCL)/D19	GPIO22
10	A6 (Analog)	A6	GPIO1
11	A7 (Analog)	NEOP_DO	NC
12	5V	5V	5V
13	RESET	RST	RST
14	GND	GND	GND
15	VIN	VIN	VIN
16	D0 (RX)	D0/RX	GPIO23
17	D1 (TX)	D1/TX	GPIO24
18	RESET	RST	RST
19	GND	GND	GND
20	D2	D2/NEOP	GPIO8
21	D3 (PWM)	D3	GPIO9
22	D4	D4	GPIO10
23	D5 (PWM)	D5	GPIO11
24	D6 (PWM)	D6	GPIO13
25	D7	D7	GPIO14
26	D8	D8	GPIO26
27	D9 (PWM)	D9	GPIO27
28	D10 (PWM/SS)	D10/SS	GPIO25

continues on next page

Table 5.1 – continued from previous page

Arduino Nano Pin	Arduino Nano Description	**PULSAR H2**	ESP32-H2 GPIO
29	D11 (PWM/MOSI)	D11/MOSI/A4	GPIO5
30	D12 (MISO)	D12/MISO	GPIO0

CHAPTER

SIX

GENERAL PURPOSE INPUT/OUTPUT (GPIO) PINS

The General Purpose Input/Output (GPIO) pins on the **PULSAR H2** development board are used to connect external devices to the microcontroller. These pins can be configured as either input or output. In this section, we will explore how to work with GPIO pins on the **PULSAR H2** development board using both MicroPython and C++.

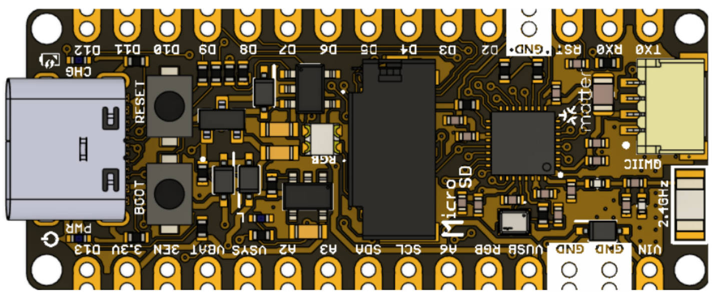


Fig. 6.1: **PULSAR H2** Development Board

Let's begin with a simple example: blinking an LED. This example demonstrates how to control GPIO pins on the **PULSAR H2** development board using both MicroPython and C++.

6.1 Working with LEDs on ESP32-H2

In this section, we will learn how to control a single LED using a microcontroller. The LED will be connected to a GPIO pin, and we will control its on/off states using a simple program.

6.1.1 LED Blinking Example

Tip: The following example demonstrates how to blink an LED connected to GPIO pin 4 on the **PULSAR H2** development board. The LED will turn on for 1 second and then turn off for 1 second, repeating this pattern indefinitely.

MicroPython

```
import machine
import time

led = machine.Pin(4, machine.Pin.OUT)

```

(continues on next page)

(continued from previous page)

```
def loop():
    while True:
        led.on()  # Turn the LED on
        time.sleep(1)  # Wait for 1 second
        led.off()  # Turn the LED off
        time.sleep(1)  # Wait for 1 second

loop()
```

C++

```
#define LED 4

// The setup function runs once when you press reset or power the board
void setup() {
    // Initialize digital pin LED as an output.
    pinMode(LED, OUTPUT);
}

// The loop function runs continuously
void loop() {
    digitalWrite(LED, HIGH);   // Turn the LED on (HIGH is the voltage level)
    delay(1000);              // Wait for 1 second
    digitalWrite(LED, LOW);   // Turn the LED off (LOW is the voltage level)
    delay(1000);              // Wait for 1 second
}
```

esp-idf

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define BLINK_GPIO GPIO_NUM_4  // Puedes cambiarlo según tu hardware

void app_main(void)
{
    // Configura el GPIO como salida
    gpio_reset_pin(BLINK_GPIO);
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);

    while (1) {
        // Enciende el LED
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(pdMS_TO_TICKS(500)); // 500 ms

        // Apaga el LED
        gpio_set_level(BLINK_GPIO, 0);
        vTaskDelay(pdMS_TO_TICKS(500)); // 500 ms
    }
}
```

CHAPTER

SEVEN

ANALOG TO DIGITAL CONVERSION

Learn how to read analog sensor values using the ADC module on the **PULSAR H2** development board with the ESP32-H2. This section will cover the basics of analog input and conversion techniques.

7.1 ADC Definition

Analog-to-digital conversion (ADC) is a process that converts analog signals into digital values. The ESP32-H2, equipped with multiple ADC channels, provides flexible options for reading analog voltages and converting them into digital values. Below, you will find the details on how to utilize these pins for ADC operations.

7.1.1 Quantification and Codification of Analog Signals

Analog signals are continuous signals that can take on any value within a given range. Digital signals, on the other hand, are discrete signals that can only take on specific values. The process of converting an analog signal into a digital signal involves two steps: quantification and codification.

- **Quantification**: This step involves dividing the analog signal into discrete levels. The number of levels determines the resolution of the ADC. For example, a 12-bit ADC can divide the analog signal into 4096 levels.
- **Codification**: This step involves assigning a digital code to each quantization level. The digital code represents the value of the analog signal at that level.

7.2 ESP32-H2 ADC Channels (Official Documentation)

Complete ADC channel mapping according to ESP32-H2 official documentation:

Table 7.1: ESP32-H2 ADC Channels

GPIO Pin	ADC Channel	Description
GPIO0	ADC1_CH0	12-bit SAR ADC, Channel 0
GPIO2	ADC1_CH1	12-bit SAR ADC, Channel 1
GPIO3	ADC1_CH2	12-bit SAR ADC, Channel 2
GPIO4	ADC1_CH3	12-bit SAR ADC, Channel 3
GPIO5	ADC1_CH4	12-bit SAR ADC, Channel 4

7.3 ADC Pin Status on PULSAR H2

Table 7.2: PULSAR H2 Pin Status for ADC Usage

PULSAR H2 Pin	ESP32-H2 GPIO	ADC Status	ADC Channel	Notes / Alternative Function
A0/D14	N/C	NO	—	Battery control circuit (not connected to MCU)
A1/D15	N/C	NO	—	System voltage monitoring (not connected to MCU)
A2/D16	**GPIO2**	**YES**	**ADC1_CH1**	**Available for analog readings**
A3/D17	**GPIO3**	**YES**	**ADC1_CH2**	**Available for analog readings**
A4 (SDA)	**GPIO12**	**YES**	**ADC Capable**	**I2C SDA + ADC support (JST connector)**
A5 (SCL)	**GPIO22**	**YES**	**ADC Capable**	**I2C SCL + ADC support (JST connector)**
A6	**GPIO1**	**YES**	**ADC1_CH0**	**Available for analog readings**
A7	N/C	NO	—	NeoPixel (WS2812B) output pin

Warning: Pin Usage Notes:

- **A0 and A1**: These are NOT connected to the ESP32-H2 microcontroller
- **A2, A3, A6**: Dedicated analog pins - best for ADC readings
- **A4 and A5**: Dual function (I2C + ADC) - available on JST connector
- **A7**: This is for NeoPixel output, not analog input

A2, A3, A4, A5, and A6 work for analog readings!

7.4 Summary Table: Usable ADC Pins

Important: 5 pins can be used for analog readings on PULSAR H2:

Table 7.3: Usable ADC Pins

Pin Label	GPIO	ADC Channel	Notes
A2	**GPIO2**	**ADC1_CH1**	**Dedicated analog pin**
A3	**GPIO3**	**ADC1_CH2**	**Dedicated analog pin**
A4 (SDA)	**GPIO12**	**ADC Capable**	**I2C SDA + ADC (JST connector)**
A5 (SCL)	**GPIO22**	**ADC Capable**	**I2C SCL + ADC (JST connector)**
A6	**GPIO1**	**ADC1_CH0**	**Dedicated analog pin**

Note: ADC Summary:

- **ESP32-H2 chip**: Has 5 ADC channels (GPIO0, GPIO1, GPIO2, GPIO3, GPIO4, GPIO5)
- **PULSAR H2 board**: 5 ADC pins are usable (A2, A3, A4, A5, A6)

- **Resolution**: 12-bit (0-4095 values)
- **Voltage Range**: 0V to 3.3V

Important: I2C vs ADC on A4/A5: You can use A4 and A5 for ADC readings when not using I2C. If you need both I2C and ADC, use A2/A3 for ADC and A4/A5 for I2C.

7.5 Class ADC

The `machine.ADC` class is used to create ADC objects that can interact with the analog pins.

class machine.ADC(*pin*)

The constructor for the ADC class takes a single argument: the pin number.

7.6 ADC Pin Usage Examples

Use only **A2 (GPIO2)** or **A3 (GPIO3)** for analog readings:

MicroPython

```
import machine

# ADC pins available on PULSAR H2:
adc_a2 = machine.ADC(machine.Pin(2))   # A2 - ADC1_CH1 (dedicated)
adc_a3 = machine.ADC(machine.Pin(3))   # A3 - ADC1_CH2 (dedicated)
adc_a4 = machine.ADC(machine.Pin(12))  # A4 - GPIO12 (SDA + ADC)
adc_a5 = machine.ADC(machine.Pin(22))  # A5 - GPIO22 (SCL + ADC)
adc_a6 = machine.ADC(machine.Pin(1))   # A6 - ADC1_CH0 (dedicated)
```

C++

```
// ADC pins available on PULSAR H2:
#define ADC_PIN_A2  2   // GPIO2 (A2) - ADC1_CH1 (dedicated)
#define ADC_PIN_A3  3   // GPIO3 (A3) - ADC1_CH2 (dedicated)
#define ADC_PIN_A4  12  // GPIO12 (A4) - SDA + ADC (JST connector)
#define ADC_PIN_A5  22  // GPIO22 (A5) - SCL + ADC (JST connector)
#define ADC_PIN_A6  1   // GPIO1 (A6) - ADC1_CH0 (dedicated)
```

7.7 Reading Values

To read the analog value converted to a digital format:

MicroPython

```
adc_value = adc.read()  # Read the ADC value
print(adc_value)  # Print the ADC value
```

C++

```
voltage = analogRead(ADC_PIN);
```

7.8 Example Code

Below is an example that continuously reads from an ADC pin and prints the results:

MicroPython

```python
import machine
import time

# Setup - All available ADC pins on PULSAR H2
adc_a2 = machine.ADC(machine.Pin(2))   # A2 - dedicated ADC
adc_a3 = machine.ADC(machine.Pin(3))   # A3 - dedicated ADC
adc_a4 = machine.ADC(machine.Pin(12))  # A4 - SDA + ADC (JST)
adc_a5 = machine.ADC(machine.Pin(22))  # A5 - SCL + ADC (JST)
adc_a6 = machine.ADC(machine.Pin(1))   # A6 - dedicated ADC

# Continuous reading from all ADC pins
while True:
    # Read all ADC pins
    value_a2 = adc_a2.read_u16()
    value_a3 = adc_a3.read_u16()
    value_a4 = adc_a4.read_u16()
    value_a5 = adc_a5.read_u16()
    value_a6 = adc_a6.read_u16()

    # Convert to voltages
    voltage_a2 = (value_a2 / 65535) * 3.3
    voltage_a3 = (value_a3 / 65535) * 3.3
    voltage_a4 = (value_a4 / 65535) * 3.3
    voltage_a5 = (value_a5 / 65535) * 3.3
    voltage_a6 = (value_a6 / 65535) * 3.3

    print(f"A2: {voltage_a2:.2f}V | A3: {voltage_a3:.2f}V | A4: {voltage_a4:.2f}V | A5:
↪{voltage_a5:.2f}V | A6: {voltage_a6:.2f}V")
    time.sleep(1)
```

C++

```cpp
// Only these pins work for ADC on PULSAR H2
const int adcPin_A2 = 2; // GPIO2 (A2) - ADC1_CH1
const int adcPin_A3 = 3; // GPIO3 (A3) - ADC1_CH2

void setup() {
  Serial.begin(115200);
  analogReadResolution(12); // Set resolution to 12-bit (0-4095)
  delay(1000);
  Serial.println("PULSAR H2 ADC Test - Only A2 and A3 work!");
}

```

(continues on next page)

(continued from previous page)

```
void loop() {
  // Read from A2
  int value_A2 = analogRead(adcPin_A2);
  float voltage_A2 = (value_A2 / 4095.0) * 3.3;

  // Read from A3
  int value_A3 = analogRead(adcPin_A3);
  float voltage_A3 = (value_A3 / 4095.0) * 3.3;

  // Print results
  Serial.print("A2: "); Serial.print(value_A2);
  Serial.print(" ("); Serial.print(voltage_A2); Serial.print("V) | ");
  Serial.print("A3: "); Serial.print(value_A3);
  Serial.print(" ("); Serial.print(voltage_A3); Serial.println("V)");

  delay(1000);
}
```

esp-idf

```
#include <stdio.h>
#include "esp_log.h"
#include "esp_err.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_adc/adc_oneshot.h"

static const char *TAG = "ADC_MIN";

void app_main(void)
{
    adc_oneshot_unit_handle_t adc_handle;
    adc_oneshot_unit_init_cfg_t init_cfg = {
        .unit_id = ADC_UNIT_1,
    };
    ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_cfg, &adc_handle));

    adc_oneshot_chan_cfg_t chan_cfg = {
        .bitwidth = ADC_BITWIDTH_DEFAULT,
        .atten = ADC_ATTEN_DB_12,  // <- Usa el recomendado
    };
    ESP_ERROR_CHECK(adc_oneshot_config_channel(adc_handle, ADC_CHANNEL_2, &chan_cfg)); //
↪ GPIO2

    int adc_raw;
    while (1) {
        ESP_ERROR_CHECK(adc_oneshot_read(adc_handle, ADC_CHANNEL_2, &adc_raw));
        ESP_LOGI(TAG, "Lectura ADC (GPIO2): %d", adc_raw);
        vTaskDelay(pdMS_TO_TICKS(1000));  // <- Necesitabas incluir FreeRTOS
    }
}
```

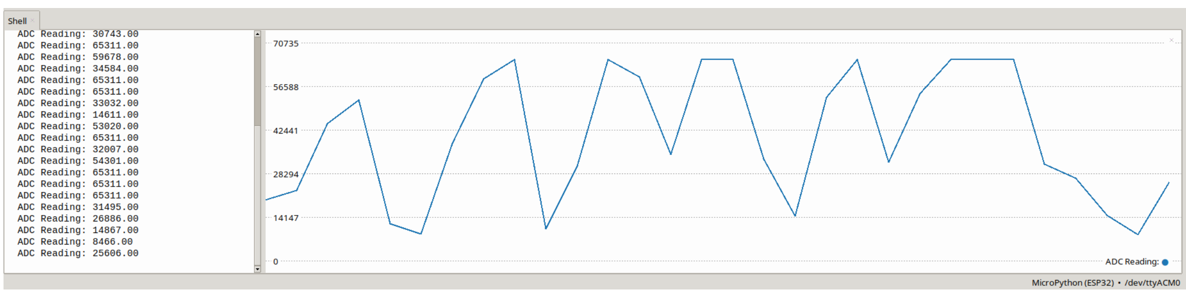


Fig. 7.1: Example of ADC usage on the **PULSAR H2** board.

CHAPTER

EIGHT

I2C (INTER-INTEGRATED CIRCUIT)

Discover the I2C communication protocol and learn how to communicate with I2C devices using the PULSAR H2 board. This section will cover I2C bus setup and communication with I2C peripherals.

8.1 I2C Overview

I2C (Inter-Integrated Circuit) is a synchronous, multi-master, multi-slave, packet-switched, single-ended, serial communication bus. It is commonly used to connect low-speed peripherals to processors and microcontrollers.

The PULSAR H2 development board, powered by the ESP32-H2, features advanced I2C communication capabilities with **two I2C controllers** that support:

- **Standard mode** (100 kbit/s) and **Fast mode** (400 kbit/s)
- **Master and slave modes**
- **7-bit and 10-bit addressing**
- **Dual address mode support**
- **Programmable digital noise filtering**
- **SCL clock stretching in slave mode**

8.2 Pinout Details

Below is the pinout table for the I2C connections on the PULSAR H2, detailing the pin assignments for SDA and SCL.

Table 8.1: ESP32-H2 I2C Pinout

Pin	Function
A4 (SDA)/D18	GPIO12 / I2C SDA
A5 (SCL)/D19	GPIO22 / I2C SCL

8.3 I2C Features on ESP32-H2

The ESP32-H2 is designed as an ultra-low-power microcontroller, making it inherently energy-efficient for I2C communication. Key features include:

- **Ultra-low power consumption**: The ESP32-H2 is optimized for battery-powered applications
- **Two I2C controllers**: Support for multiple I2C buses
- **Flexible GPIO assignment**: I2C pins can be assigned to any available GPIO
- **Standard and Fast modes**: Support for 100 kbit/s (Standard) and 400 kbit/s (Fast mode)
- **Master and slave modes**: Can operate as either I2C master or slave device

---

Note: Unlike other ESP32 variants, the ESP32-H2 doesn't have a separate "LP I2C" mode because the entire chip is designed for low power operation.

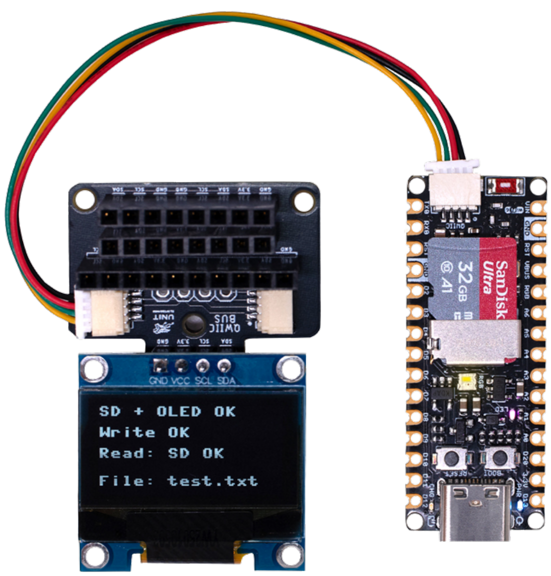


Fig. 8.1: I2C Device Connection Example

8.4 Scanning for I2C Devices

To scan for I2C devices connected to the bus, you can use the following code snippet:

MicroPython

```
import machine

# PULSAR H2 I2C pins: A4 (SDA) = GPIO12, A5 (SCL) = GPIO22
i2c = machine.I2C(0, scl=machine.Pin(22), sda=machine.Pin(12))
devices = i2c.scan()
```

(continues on next page)

(continued from previous page)

```
for device in devices:
    print("Device found at address: {}".format(hex(device)))
```

C++

```
#include <Wire.h>

void setup() {
  // PULSAR H2 I2C pins: A4 (SDA) = GPIO12, A5 (SCL) = GPIO22
  Wire.setSDA(12);
  Wire.setSCL(22);
  Wire.begin();
  Serial.begin(9600); // Start serial communication at 9600 baud rate
  while (!Serial); // Wait for serial port to connect
  Serial.println("\nI2C Scanner");
}

void loop() {
  byte error, address;
  int nDevices;

  Serial.println("Scanning...");

  nDevices = 0;
  for(address = 1; address < 127; address++ ) {
    // The i2c_scanner uses the return value of the Write.endTransmisstion to see if
    // a device did acknowledge to the address.
    Wire.beginTransmission(address);
    error = Wire.endTransmission();

    if (error == 0) {
      Serial.print("I2C device found at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.print(address, HEX);
      Serial.println("  !");

      nDevices++;
    }
    else if (error==4) {
      Serial.print("Unknown error at address 0x");
      if (address<16)
        Serial.print("0");
      Serial.println(address, HEX);
    }
  }
  if (nDevices == 0)
    Serial.println("No I2C devices found\n");
  else
    Serial.println("done\n");

  delay(5000);           // wait 5 seconds for next scan
```

(continues on next page)

(continued from previous page)

```
}
```

8.5 SSD1306 Display

Fig. 8.2: SSD1306 Display

The display 128x64 pixel monochrome OLED display equipped with an SSD1306 controller is connected using a JST 1.25mm 4-pin connector. The following table provides the pinout details for the display connection.

Table 8.2: SSD1306 Display Pinout

Pin	Connection
1	GND
2	VCC
3	SDA
4	SCL

8.5.1 Library Support

MicroPython

The *ocks.py* library for MicroPython on ESP32 & RP2040 is compatible with the SSD1306 display controller.

Installation

1. Open Thonny.
2. Navigate to **Tools** -> **Manage Packages**.
3. Search for `ocks` and click **Install**.

Alternatively, download the library from ocks.py.

Microcontroller Configuration

```
SoftI2C(scl, sda, *, freq=400000, timeout=50000)
```

Change the following line depending on your microcontroller:

For PULSAR H2 (ESP32-H2):

```
>>> i2c = machine.SoftI2C(freq=400000, timeout=50000, sda=machine.Pin(12), scl=machine.
↪Pin(22))
```

For ESP32 (Generic):

```
>>> i2c = machine.SoftI2C(freq=400000, timeout=50000, sda=machine.Pin(21), scl=machine.
↪Pin(22))
```

For RP2040:

```
>>> i2c = machine.SoftI2C(freq=400000, timeout=50000, sda=machine.Pin(4), scl=machine.
↪Pin(5))
```

Example Code

```
import machine
from ocks import SSD1306_I2C

# PULSAR H2 I2C pins: A4 (SDA) = GPIO12, A5 (SCL) = GPIO22
i2c = machine.SoftI2C(freq=400000, timeout=50000, sda=machine.Pin(12), scl=machine.
↪Pin(22))

oled = SSD1306_I2C(128, 64, i2c)

# Fill the screen with white and display
oled.fill(1)
oled.show()

# Clear the screen (fill with black)
oled.fill(0)
oled.show()

# Display text
oled.text('UNIT', 50, 10)
oled.text('ELECTRONICS', 25, 20)
oled.show()
```

Replace `sda=machine.Pin(*)` and `scl=machine.Pin(*)` with the appropriate GPIO pins for your setup.

C++

The *Adafruit_SSD1306* library for Arduino is compatible with the SSD1306 display controller.

Installation

1. Open the Arduino IDE.
2. Navigate to **Tools** -> **Manage Libraries**.

3. Search for `Adafruit_SSD1306` and click **Install**.

Example Code

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// OLED display TWI (I2C) interface
#define OLED_RESET     -1 // Reset pin # (or -1 if sharing Arduino reset pin)
#define SCREEN_WIDTH   128 // OLED display width, in pixels
#define SCREEN_HEIGHT  64  // OLED display height, in pixels
#define SDA_PIN        4   // SDA pin
#define SCL_PIN        5   // SCL pin

// Declare an instance of the class (specify width and height)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

void setup() {
  Serial.begin(9600);

  // Initialize I2C - PULSAR H2 pins: A4 (SDA) = GPIO12, A5 (SCL) = GPIO22
  Wire.setSDA(12);
  Wire.setSCL(22);
  Wire.begin();
  // Start the OLED display
  if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x64
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
  }

  // Clear the buffer
  display.clearDisplay();

  // Set text size and color
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0,0);
  display.println(F("UNIT ELECTRONICS!"));
  display.display();  // Show initial text
  delay(4000);        // Pause for 2 seconds
}

void loop() {
  // Increase a counter
  static int counter = 0;

  // Clear the display buffer
  display.clearDisplay();
  display.setCursor(0, 10); // Position cursor for new text
  display.setTextSize(2);   // Larger text size

  // Display the counter
  display.print(F("Count: "));
```

(continues on next page)

(continued from previous page)

```
  display.println(counter);

  // Refresh the display to show the new count
  display.display();

  // Increment the counter
  counter++;

  // Wait for half a second
  delay(500);
}
```

esp-idf

```
#include "ssd1306.h"
#include "driver/i2c.h"
#include "esp_log.h"

#define I2C_MASTER_NUM I2C_NUM_0
#define I2C_MASTER_SDA_IO 12  // PULSAR H2 A4 (SDA) = GPIO12
#define I2C_MASTER_SCL_IO 22  // PULSAR H2 A5 (SCL) = GPIO22
#define I2C_MASTER_FREQ_HZ 100000

static const char *TAG = "MAIN";

void scan_i2c_bus(void) {
   ESP_LOGI(TAG, "Scanning I2C bus...");
   for (uint8_t addr = 1; addr < 127; addr++) {
      i2c_cmd_handle_t cmd = i2c_cmd_link_create();
      i2c_master_start(cmd);
      i2c_master_write_byte(cmd, (addr << 1) | I2C_MASTER_WRITE, true);
      i2c_master_stop(cmd);
      esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, 100 / portTICK_PERIOD_
↪MS);
      i2c_cmd_link_delete(cmd);
      if (ret == ESP_OK) {
        ESP_LOGI(TAG, "Found device at 0x%02X", addr);
      }
   }
   ESP_LOGI(TAG, "Scan complete.");
}

void app_main(void) {
   i2c_config_t conf = {
      .mode = I2C_MODE_MASTER,
      .sda_io_num = I2C_MASTER_SDA_IO,
      .scl_io_num = I2C_MASTER_SCL_IO,
      .sda_pullup_en = GPIO_PULLUP_ENABLE,
      .scl_pullup_en = GPIO_PULLUP_ENABLE,
      .master.clk_speed = I2C_MASTER_FREQ_HZ,
   };
```

(continues on next page)

(continued from previous page)

```
    i2c_param_config(I2C_MASTER_NUM, &conf);
    i2c_driver_install(I2C_MASTER_NUM, conf.mode, 0, 0, 0);

    scan_i2c_bus(); // Optional

    ssd1306_init(I2C_MASTER_NUM);
    ssd1306_clear(I2C_MASTER_NUM);
    ssd1306_draw_text(I2C_MASTER_NUM, 0, "ESP32-H2 ");
    ssd1306_draw_text(I2C_MASTER_NUM, 2, "I2C Scan + OLED");
    ssd1306_draw_text(I2C_MASTER_NUM, 4, "Monosaurio");
}
```

CHAPTER

NINE

SPI (SERIAL PERIPHERAL INTERFACE)

9.1 SPI Overview

SPI (Serial Peripheral Interface) is a synchronous, full-duplex, master-slave communication bus. It is commonly used to connect microcontrollers to peripherals such as sensors, displays, and memory devices. The PULSAR H2 development board features SPI communication capabilities, allowing you to interface with a wide range of SPI devices including microSD cards.

9.2 SDCard SPI

Warning: Ensure that the Micro SD contain data. We recommend saving multiple files beforehand to facilitate the use. Format the Micro SD card to FAT32 before using it with the ESP32-H2.

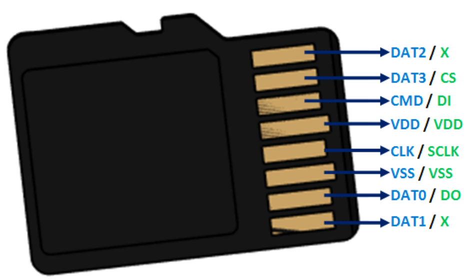


Fig. 9.1: Micro SD Card Pinout

The conections are as follows:

This table illustrates the connections between the SD card and the GPIO pins on the ESP32-H2 (PULSAR H2)

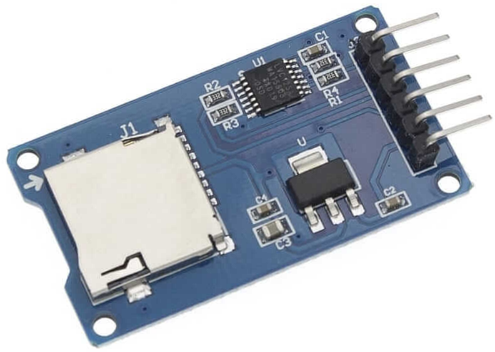


Fig. 9.2: Micro SD Card external reader

Table 9.1: MicroSD SPI Connections

SD Card Pin	Function	ESP32-H2 GPIO	PULSAR H2 Pin
1	D2 (Not Connected)	N/C	—
2	D3/CS (Chip Select)	GPIO25	D10/SS
3	CMD/MOSI	GPIO5	D11/MOSI
4	VDD (3.3V)	3.3V	3.3V
5	CLK/SCK	GPIO4	D13/SCK
6	VSS (GND)	GND	GND
7	D0/MISO	GPIO0	D12/MISO
8	D1 (Not Connected)	N/C	—

Note: MicroSD Connection Notes:

- The microSD card is connected via SPI interface using **4 wires** (MOSI, MISO, SCK, CS)
- **CS (Chip Select)** is connected to **GPIO25** (D10/SS pin)
- **MOSI** is connected to **GPIO5** (D11/MOSI pin)
- **MISO** is connected to **GPIO0** (D12/MISO pin)
- **SCK** is connected to **GPIO4** (D13/SCK pin)
- **D2** and **D1** pins of the SD card are not used in SPI mode
- Make sure the SD card is formatted as **FAT32** before use
- The SD card operates at **3.3V** - no level shifters needed

MicroPython

```
import machine
import os
from sdcard import SDCard
```

(continues on next page)

(continued from previous page)

```
# Pines SPI para microSD
MOSI_PIN = 5
MISO_PIN = 0
SCK_PIN = 4
CS_PIN = 25

# Inicializar SPI
spi = machine.SPI(1, baudrate=500000, polarity=0, phase=0,
                  sck=machine.Pin(SCK_PIN),
                  mosi=machine.Pin(MOSI_PIN),
                  miso=machine.Pin(MISO_PIN))

# Inicializar tarjeta SD
sd = SDCard(spi, machine.Pin(CS_PIN))

# Montar la SD en el sistema de archivos
os.mount(sd, "/sd")

# Listar archivos y directorios en la SD
print("Archivos en la SD:")
print(os.listdir("/sd"))
```

C++

```
#include <SPI.h>
#include <SD.h>

// Pines SPI para microSD
#define MOSI_PIN 5
#define MISO_PIN 0
#define SCK_PIN 4
#define CS_PIN 25

File myFile;

void setup() {
  Serial.begin(115200);
  while (!Serial) ; // Esperar a que el puerto serie esté listo

  SPI.begin(SCK_PIN, MISO_PIN, MOSI_PIN, CS_PIN);

  Serial.println("Inicializando tarjeta SD...");

  if (!SD.begin(CS_PIN)) {
    Serial.println("Error al inicializar la tarjeta SD.");
    return;
  }

  Serial.println("Tarjeta SD inicializada correctamente.");

  // Listar archivos
```

(continues on next page)

(continued from previous page)

```
  Serial.println("Archivos en la SD:");
  listDir(SD, "/", 0);

  // Crear y escribir en el archivo
  myFile = SD.open("/test.txt", FILE_WRITE);
  if (myFile) {
    myFile.println("Hola, Arduino en SD!");
    myFile.println("Esto es una prueba de escritura.");
    myFile.close();
    Serial.println("Archivo escrito correctamente.");
  } else {
    Serial.println("Error al abrir test.txt para escribir.");
  }

  // Leer el archivo
  myFile = SD.open("/test.txt");
  if (myFile) {
    Serial.println("\nContenido del archivo:");
    while (myFile.available()) {
      Serial.write(myFile.read());
    }
    myFile.close();
  } else {
    Serial.println("Error al abrir test.txt para lectura.");
  }

  // Volver a listar archivos
  Serial.println("\nArchivos en la SD después de la escritura:");
  listDir(SD, "/", 0);
}

void loop() {
  // Nada en el loop
}

// Función para listar archivos y carpetas
void listDir(fs::FS &fs, const char * dirname, uint8_t levels) {
  File root = fs.open(dirname);
  if (!root) {
    Serial.println("Error al abrir el directorio");
    return;
  }
  if (!root.isDirectory()) {
    Serial.println("No es un directorio");
    return;
  }

  File file = root.openNextFile();
  while (file) {
    Serial.print("  ");
    Serial.print(file.name());
    if (file.isDirectory()) {
```

(continues on next page)

(continued from previous page)

```
      Serial.println("/");
      if (levels) {
        listDir(fs, file.name(), levels - 1);
      }
    } else {
      Serial.print("\t\t");
      Serial.println(file.size());
    }
    file = root.openNextFile();
  }
}
```

esp-idf

```
#include <string.h>
#include <sys/stat.h>
#include "esp_log.h"
#include "esp_vfs_fat.h"
#include "sdmmc_cmd.h"

#define MOUNT_POINT "/sdcard"

#define PIN_NUM_MISO  CONFIG_EXAMPLE_PIN_MISO
#define PIN_NUM_MOSI  CONFIG_EXAMPLE_PIN_MOSI
#define PIN_NUM_CLK   CONFIG_EXAMPLE_PIN_CLK
#define PIN_NUM_CS    CONFIG_EXAMPLE_PIN_CS

static const char *TAG = "SDCARD";

void app_main(void)
{
    esp_err_t ret;
    sdmmc_card_t *card;

    ESP_LOGI(TAG, "Initializing SD card...");

    esp_vfs_fat_sdmmc_mount_config_t mount_config = {
        .format_if_mount_failed = false,
        .max_files = 3,
        .allocation_unit_size = 16 * 1024
    };

    sdmmc_host_t host = SDSPI_HOST_DEFAULT();

    spi_bus_config_t bus_cfg = {
        .mosi_io_num = PIN_NUM_MOSI,
        .miso_io_num = PIN_NUM_MISO,
        .sclk_io_num = PIN_NUM_CLK,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .max_transfer_sz = 4000,
    };
```

(continues on next page)

(continued from previous page)

```
    ret = spi_bus_initialize(host.slot, &bus_cfg, SDSPI_DEFAULT_DMA);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "Failed to init SPI bus.");
        return;
    }

    sdspi_device_config_t slot_config = SDSPI_DEVICE_CONFIG_DEFAULT();
    slot_config.gpio_cs = PIN_NUM_CS;
    slot_config.host_id = host.slot;

    ret = esp_vfs_fat_sdspi_mount(MOUNT_POINT, &host, &slot_config, &mount_config, &
↪card);
    if (ret != ESP_OK) {
        ESP_LOGE(TAG, "Failed to mount filesystem.");
        return;
    }

    ESP_LOGI(TAG, "Filesystem mounted.");

    const char *file_path = MOUNT_POINT"/test.txt";
    FILE *f = fopen(file_path, "w");
    if (f == NULL) {
        ESP_LOGE(TAG, "Failed to open file for writing.");
        return;
    }

    fprintf(f, "Hello from ESP32!\n");
    fclose(f);
    ESP_LOGI(TAG, "File written.");

    f = fopen(file_path, "r");
    if (f) {
        char line[64];
        fgets(line, sizeof(line), f);
        fclose(f);
        ESP_LOGI(TAG, "Read from file: '%s'", line);
    } else {
        ESP_LOGE(TAG, "Failed to read file.");
    }

    esp_vfs_fat_sdcard_unmount(MOUNT_POINT, card);
    spi_bus_free(host.slot);
    ESP_LOGI(TAG, "Card unmounted.");
}
```

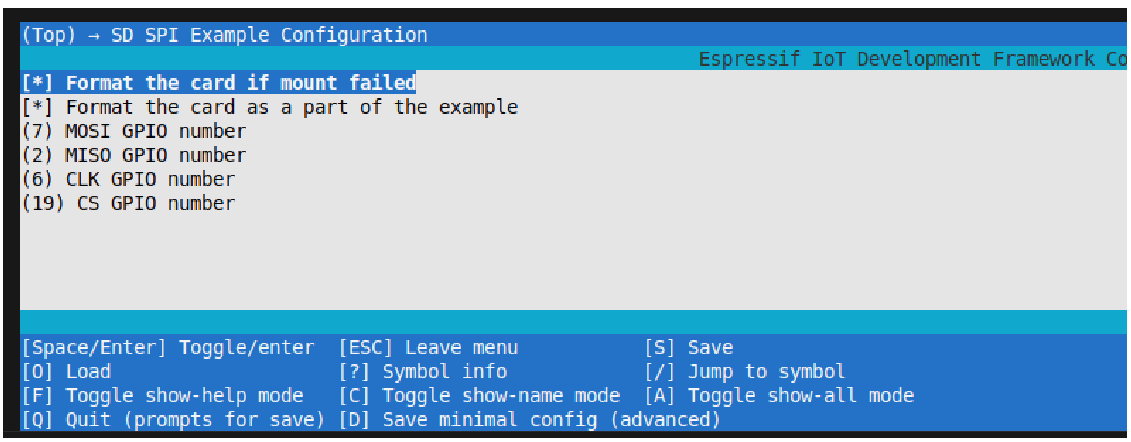


Fig. 9.3: ESP-IDF Menuconfig SD SPI Configuration

CHAPTER

TEN

WS2812 CONTROL

Harness the power of WS1280 LED strips with the PULSAR H2 board. Learn how to control RGB LED strips and create dazzling lighting effects using MicroPython.

This section describes how to control WS2812 LED strips using the PULSAR H2 board. The PULSAR H2 board has a GPIO pin embebbed connected to the single WS2812 LED.

Table 10.1: Pin Mapping for WS2812

PIN	GPIO ESP32H2
DIN	8

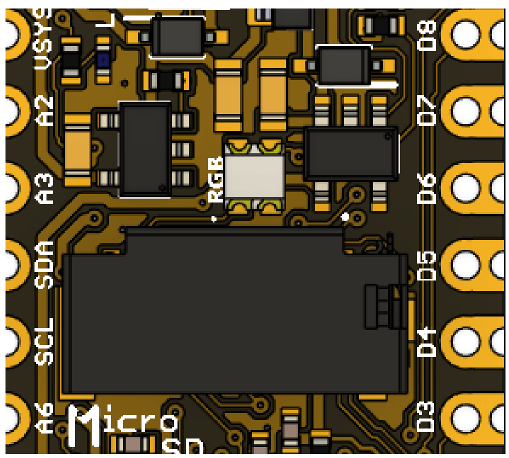


Fig. 10.1: WS28xx LED Strip

10.1 Code Example

Below is an example that demonstrates how to control WS1280 LED strips using the PULSAR H2 board

MicroPython

```
from machine import Pin
from neopixel import NeoPixel
np = NeoPixel(Pin(8), 1)
np[0] = (255, 128, 0) # set to red, full brightness
```

(continues on next page)

(continued from previous page)

```
np.write()
```

C++

```
#include <Adafruit_NeoPixel.h>
#define PIN 8
Adafruit_NeoPixel strip = Adafruit_NeoPixel(1, PIN, NEO_GRB + NEO_KHZ800);
void setup() {
   strip.begin();
   strip.setPixelColor(0, 255, 128, 0); // set to red, full brightness
   strip.show();
}
```

esp-idf

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/rmt_tx.h"
#include "esp_err.h"

void app_main(void) {
   rmt_channel_handle_t tx_channel = NULL;
   rmt_tx_channel_config_t tx_config = {
      .gpio_num = GPIO_NUM_8,
      .clk_src = RMT_CLK_SRC_DEFAULT,
      .resolution_hz = 10000000, // 10MHz resolution, 1 tick = 0.1us
      .mem_block_symbols = 64,
      .trans_queue_depth = 4,
      .flags.invert_out = false,
      .flags.with_dma = false,
   };
   ESP_ERROR_CHECK(rmt_new_tx_channel(&tx_config, &tx_channel));
   ESP_ERROR_CHECK(rmt_enable(tx_channel));

   rmt_encoder_handle_t bytes_encoder = NULL;
   rmt_bytes_encoder_config_t bytes_encoder_config = {
      .bit0 = {.level0 = 1, .duration0 = 3, .level1 = 0, .duration1 = 9},  // 0: ~0.3us␣
↪high, ~0.9us low
      .bit1 = {.level0 = 1, .duration0 = 9, .level1 = 0, .duration1 = 3},  // 1: ~0.9us␣
↪high, ~0.3us low
      .flags.msb_first = true,
   };
   ESP_ERROR_CHECK(rmt_new_bytes_encoder(&bytes_encoder_config, &bytes_encoder));

   rmt_transmit_config_t tx_trans_config = {
      .loop_count = 0,
   };

   uint8_t r = 255, g = 0, b = 0;
```

(continues on next page)

(continued from previous page)

```
    while (1) {
        if (r == 255 && g < 255 && b == 0) {
                g++;
        } else if (g == 255 && r > 0 && b == 0) {
                r--;
        } else if (g == 255 && b < 255 && r == 0) {
                b++;
        } else if (b == 255 && g > 0 && r == 0) {
                g--;
        } else if (b == 255 && r < 255 && g == 0) {
                r++;
        } else if (r == 255 && b > 0 && g == 0) {
                b--;
        }
        uint8_t color_data[3] = {g, r, b};

        // printf("%d %d %d\n",r,g,b);

        ESP_ERROR_CHECK(rmt_transmit(tx_channel, bytes_encoder, color_data, sizeof(color_
↪data), &tx_trans_config));
        ESP_ERROR_CHECK(rmt_tx_wait_all_done(tx_channel, portMAX_DELAY));
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}
```

Tip: for more information on the NeoPixel library, refer to the NeoPixel Library Documentation.

CHAPTER

ELEVEN

COMMUNICATION

Unlock the full communication potential of the PULSAR H2 board with various communication protocols and interfaces. Learn how to set up and use Bluetooth, Zigbee, Thread, and serial communication to connect with other devices and networks.

Note: The ESP32-H2 does not support Wi-Fi. It is designed for IoT applications using Bluetooth, Zigbee, and Thread (802.15.4) protocols.

11.1 Bluetooth

Explore Bluetooth communication capabilities and learn how to connect to Bluetooth devices.

scan sniffer Code

```
import bluetooth
import time

# Initialize Bluetooth
ble = bluetooth.BLE()
ble.active(True)

# Helper function to convert memoryview to MAC address string
def format_mac(addr):
    return ':'.join('{:02x}'.format(b) for b in addr)

# Helper function to parse device name from advertising data
def decode_name(data):
    i = 0
    length = len(data)
    while i < length:
        ad_length = data[i]
        ad_type = data[i + 1]
        if ad_type == 0x09:  # Complete Local Name
            return str(data[i + 2:i + 1 + ad_length], 'utf-8')
        elif ad_type == 0x08:  # Shortened Local Name
            return str(data[i + 2:i + 1 + ad_length], 'utf-8')
        i += ad_length + 1
    return None
```

(continues on next page)

(continued from previous page)

```
# Global counter for devices found
devices_found = 0
max_devices = 10  # Limit to 10 devices

# Callback function to handle advertising reports
def bt_irq(event, data):
    global devices_found
    if event == 5:  # event 5 is for advertising reports
        if devices_found >= max_devices:
            ble.gap_scan(None)  # Stop scanning
            print("Scan stopped, limit reached.")
            return

        addr_type, addr, adv_type, rssi, adv_data = data
        mac_addr = format_mac(addr)
        device_name = decode_name(adv_data)
        if device_name:
            print(f"Device found: {mac_addr} (RSSI: {rssi}) Name: {device_name}")
        else:
            print(f"Device found: {mac_addr} (RSSI: {rssi}) Name: Unknown")

        devices_found += 1  # Increment counter

        if devices_found >= max_devices:
            ble.gap_scan(None)  # Stop scanning
            print("Scan stopped, limit reached.")

# Set the callback function
ble.irq(bt_irq)

# Start active scanning
ble.gap_scan(10000, 30000, 30000, True)  # Active scan for 10 seconds with interval and␣
↪window of 30ms

# Keep the program running to allow the callback to be processed
while True:
    time.sleep(1)
```

11.2 Serial

Learn about serial communication and how to communicate with other devices via serial ports.

CHAPTER TWELVE

HOW TO GENERATE AN ERROR REPORT

This guide explains how to generate an error report using GitHub repositories.

12.1 Steps to Create an Error Report

1. **Access the GitHub Repository**
   Navigate to the GitHub repository where the project is hosted.
2. **Open the Issues Tab**
   Click on the "Issues" tab located in the repository menu.
3. **Create a New Issue**
   - Click the "New Issue" button.
   - Provide a clear and concise title for the issue.
   - Add a detailed description, including relevant information such as:
     - Steps to reproduce the error.
     - Expected and actual results.
     - Any related logs, screenshots, or files.
4. **Submit the Issue**
   Once the form is complete, click the "Submit" button.

12.2 Review and Follow-Up

The development team or maintainers will review the issue and take appropriate action to address it.

CHAPTER THIRFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

M

`machine.ADC` (*built-in class*), 31