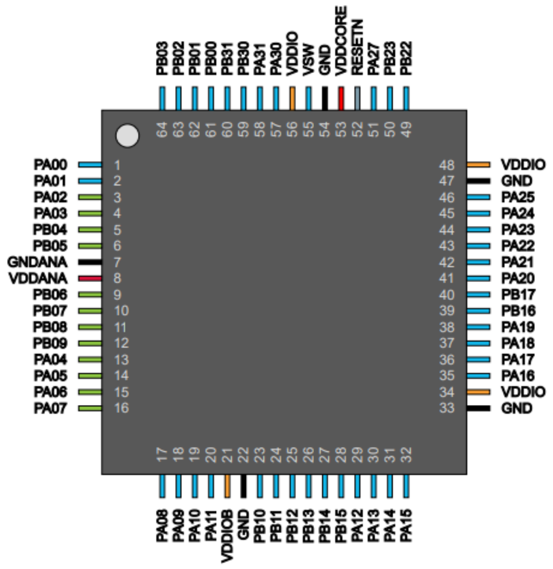


Figure 1: Samd51J20A

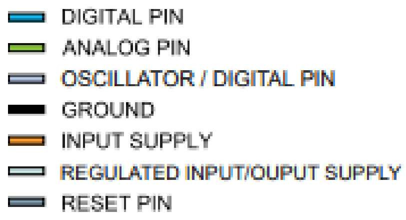


Figure 2: Pin Key

# Set up Custom SamD51 Board for Arduino IDE

Inspired by Coffee, Sleep Deprivation,
Hatred for C++ datastructures, and Github code
With love and hate for Adafruit Libraries
Compiled By: Sutton Yazzolino

11/28/2024

## Step 1: Verify PCB Board Functionality

**Check functionality:**
Check all the pins with a DMM to ensure they are no unexpected shorts and that the expected voltages are stable. If the voltages are not as expected, check the connections on the voltage regulator, as floating voltages can lead to the board acting weird.

## Step 2: Flash the bootloader to the board

**OPTIONAL: What is a bootloader?**
A bootloader is an essential piece of software that bridges the gap between the hardware and the operating system during the startup process. It initializes hardware, loads the OS, and ensures the system can boot securely and reliably.

**OPTIONAL: Where is the bootloader on Samd51?**
the bootloader occupies whatever portion it is allotted by the linker script. the linker script is just a short linker(file type: '.ld') file that tells the bootloader what portion of the chips memory to take up. For the samd51j20a that is the hex addresses from 0x00000000 to 0x4000 which are the first 16KB or 1024 lines of hex.

- for the arduino sparkfun samd51 thing plus board this is generally located at

  ```
  "C:\Users\username\Documents\ArduinoData\packages\SparkFun
  \hardware\samd\1.8.13\variants\SparkFun_SAMD51_Thing_Plus
  \linker_scripts\gcc\flash_with_bootloader.ld"
  ```

- otherwise, you can find in the sketch folder for Arduino IDE which you can locate by opening Arduino IDE then File>Preferences>Sketchbook Location:

  ```
  \ArduinoData\packages\SparkFun
  \hardware\samd\1.8.13\variants\SparkFun_SAMD51_Thing_Plus
  \linker_scripts\gcc\flash_with_bootloader.ld"
  ```

**OPTIONAL: What is a .UF2 file?**
UF2 (USB Flashing Format) is a file format designed to make it easy to flash firmware onto microcontrollers, embedded devices, and development boards over USB. It simplifies the process of programming devices by allowing firmware to be copied directly to the device as if it were a USB flash drive. UF2 is particularly popular in the maker community and is used in platforms like the Adafruit CircuitPython ecosystem, Arduino, and other microcontroller-based devices.

Action Items

**SubStep 1: Flash the Bootloader to the board**

- plug in the J-Link to your board and computer and power the board through battery or usb(the J-Link 10-pin connector we are using does not provide power to the board)
- open J-Link Commander and use the following commands:

```
J-Link>connect
Device>atsamd51j20a
TIF> S) SWD
Speed> 4000
J-Link> loadbin "insert path for bootloader .bin file" 0
J-Link> qc
```

- Bootloader available at: `SAMD51J20A Bootloader .Bin File in Google Drive`
- JLink Commander available at: `JLink commander download`

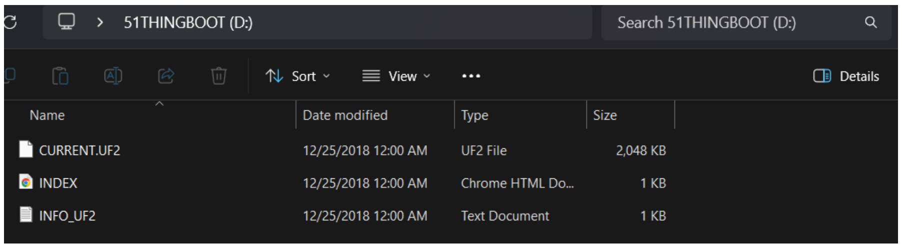
**SubStep 2: Allow the .UF2 interface to populate**

Disconnect the USB from your computer then reconnect it and you should see the board show up as a USB with the name "51THINGBOOT"

**SubStep 3: Begin programming in Arduino**

From here on you should be able to upload sketches to your board with Arduino IDE. If you have the error where you can only upload one sketch before the board becomes unresponsive, I do not know how to properly fix this but a intermediary solution is to repeat SubStep1 onward to upload the next sketch. Additionally, the pin out at this step may not be suitable for your Custom PCB Board's pinout, so continue onto the next section to re-map the Arduino Pins to the proper ports(e.g. PB01).

- Note: after uploading a arduino sketch, the device should change com ports within the arduino IDE(important if you are using the Serial Communication function) and should no longer be visible as "51THINGBOOT"
- Comment: There is an issue I can not solve with this: you can only upload one sketch before having to re-flash the board. I suspect it has something to do with memory allocation in the Arduino sketch or binary file, but have not been able to confirm it.



| Name | Date modified | Type | Size |
|---|---|---|---|
| CURRENT.UF2 | 12/25/2018 12:00 AM | UF2 File | 2,048 KB |
| INDEX | 12/25/2018 12:00 AM | Chrome HTML Do... | 1 KB |
| INFO_UF2 | 12/25/2018 12:00 AM | Text Document | 1 KB |

# Adding Pins to the SparkFun Thing Plus Board in Arduino

## 1. Locate the Board's Variant Files

The SparkFun Thing Plus uses the SAMD51 microcontroller, and its board files are defined in the SparkFun SAMD boards package. You can find the variant files in the Arduino Board Manager or the SparkFun GitHub repository.
From above, find the sketchbook folder for Arduino then access the variant file. for example, my path was:

### Paths to Variant Files

- **If installed via Arduino IDE:**

  ```
  <Arduino Sketch Folder>/hardware/sparkfun/samd/variants/<board_name>
  ```

  For the Thing Plus, the folder might be named `thing_plus`.

- **From GitHub:** Download the SparkFun SAMD board definitions from the SparkFun Arduino Boards repository:

  ```
  https://github.com/sparkfun/Arduino_Boards
  ```

  Navigate to the appropriate variant folder:

  ```
  Arduino_Boards/samd/variants/thing_plus
  ```

## 2. Understand the Variant Files

The **variant files** describe how the microcontroller's hardware (pins, peripherals, etc.) is mapped to Arduino pin numbers. These files include:

- **variant.h:** Declares pin mappings and constants (e.g., `PIN_LED`, `PIN_A0`, etc.).
- **variant.cpp:** Contains the `g_APinDescription` array, which maps Arduino pins to microcontroller pins (e.g., `PA00`, `PB01`).

## 3. Add or Modify Pins

**Edit** `variant.h`

1. **Define New Pins:** Add new macros for the pin numbers or rename existing ones. For example:

```
    #define PIN_CUSTOM_1 (44) // Assign Arduino pin 44 to your custom pin
    #define PIN_CUSTOM_2 (45) // Assign Arduino pin 45 to another custom pin
```

2. **Update Pin Count:** If you're adding pins, ensure `NUM_DIGITAL_PINS` includes the new pins:

```
    #define NUM_DIGITAL_PINS (46) // Increment total digital pins
```

**Edit** `variant.cpp`

1. **Add Pins to `g_APinDescription`:** Add entries to the `g_APinDescription` array to associate the new Arduino pins with the microcontroller's hardware pins.

   Example:

```
    const PinDescription g_APinDescription[] = {
        // Existing pins...
        {PORTA, 0, PIO_DIGITAL, PIN_ATTR_DIGITAL}, // Arduino Pin 0 = PA00
        {PORTA, 1, PIO_DIGITAL, PIN_ATTR_DIGITAL}, // Arduino Pin 1 = PA01
        // Add new pins here
        {PORTA, 20, PIO_DIGITAL, PIN_ATTR_DIGITAL}, // Arduino Pin 44 = PA20
        {PORTA, 21, PIO_DIGITAL, PIN_ATTR_DIGITAL}, // Arduino Pin 45 = PA21
    };
```

2. **Add Pin Names for Analog Pins (if needed):** If your custom pins are analog-capable, add them to the `g_APinDescription` array with `PIN_ATTR_ANALOG` instead of `PIN_ATTR_DIGITAL`.

   Example:

```
    {PORTA, 2, PIO_ANALOG, PIN_ATTR_ANALOG}, // Arduino Pin 46 = PA02 (Analog)
```

3. **Update Peripheral Attributes:** If the new pins use peripherals (e.g., PWM, I2C, SPI), ensure the attributes are set correctly.

## 4. Update `boards.txt` (Optional)

If you're creating a new board definition or modifying the pin count of an existing board, edit the `boards.txt` file:

1. Locate the board entry for the SparkFun Thing Plus.
2. Update the `build.extra_flags` to reflect the new pin definitions (if needed):

```
    thing_plus.build.extra_flags=-DNUM_DIGITAL_PINS=46
```

## 5. Test Your Changes

1. Save all changes and restart the Arduino IDE.
2. Select the **SparkFun Thing Plus** board from the Tools menu.
3. Write a simple sketch to test the new pins:

```
    void setup() {
        pinMode(44, OUTPUT); // Set custom pin as output
        digitalWrite(44, HIGH); // Turn it on
    }

    void loop() {
        // Test your new pin mappings
    }
```

## 6. Debugging

If the new pins don't work:

- Double-check the microcontroller's pin number (e.g., `PA20`).
- Verify that the pin is not reserved for other peripherals (e.g., UART, SPI).
- Confirm that the `g_APinDescription` entry matches the desired pin behavior.

## Additional Notes

- **Pin Conflicts:** Avoid using pins reserved for onboard peripherals (e.g., LEDs, I2C, SPI).
- **Datasheet Reference:** Refer to the *SAMD51 datasheet* for the functionality and limitations of specific pins.

# Programming MOSI/MISO peripheral

The Serial Peripheral Interface (SPI) is a high-speed, synchronous communication protocol used for communication between microcontrollers and peripheral devices such as sensors, displays, and storage modules. The Arduino IDE provides a built-in `SPI` library to facilitate SPI communication. This guide explains how to program an SPI peripheral using the Arduino IDE.

—

## 1. Setup for SPI Communication

### Hardware Connections

To use SPI, you need to connect the SPI pins of your microcontroller to the peripheral. The standard SPI pins are:

- **MOSI (Master Out Slave In):** Data sent from the master to the slave.
- **MISO (Master In Slave Out):** Data sent from the slave to the master.
- **SCK (Serial Clock):** Clock signal generated by the master.
- **SS (Slave Select):** Signal to select the specific slave device.

Refer to your board's pinout diagram to locate the SPI pins.

—

## 2. Include the SPI Library

First, include the SPI library in your sketch. This library provides functions to configure and use SPI communication.

```
// Include the SPI library
#include <SPI.h>
```

—

## 3. Initialize SPI Communication

The SPI library requires initialization before communication. Use the `SPI.begin()` function in the `setup()` function to initialize SPI.

```
void setup() {
    // Start SPI communication
    SPI.begin();
}
```

—

## 4. Configure SPI Parameters

You can configure the SPI communication parameters to match the requirements of your peripheral. Use the following functions:

- `SPI.setClockDivider(divider)`: Sets the SPI clock speed. Common values are `SPI_CLOCK_DIV2`, `SPI_CLOCK_DIV4`, etc.
- `SPI.setDataMode(mode)`: Sets the SPI communication mode (`SPI_MODE0`, `SPI_MODE1`, etc.).
- `SPI.setBitOrder(order)`: Sets the bit order (e.g., `MSBFIRST` or `LSBFIRST`).

Example:

```
void setup() {
    SPI.begin();

    // Configure SPI with custom parameters
    SPI.setClockDivider(SPI_CLOCK_DIV4); // Set clock speed
    SPI.setDataMode(SPI_MODE0);          // Set data mode
    SPI.setBitOrder(MSBFIRST);           // Set bit order
}
```

—

## 5. Sending and Receiving Data

Use the `SPI.transfer()` function to send and receive data over SPI. This function sends one byte of data and simultaneously receives one byte of data from the slave.

Example:

```
void loop() {
    digitalWrite(SS, LOW);  // Select the slave device

    byte receivedData = SPI.transfer(0x42); // Send 0x42 and
        receive a byte

    digitalWrite(SS, HIGH); // Deselect the slave device
}
```

—

## 6. Complete Example Code

Here is a complete example for communicating with an SPI peripheral:

```
// Include the SPI library
#include <SPI.h>

void setup() {
    // Start SPI communication
    SPI.begin();

    // Configure SPI parameters
    SPI.setClockDivider(SPI_CLOCK_DIV4);
    SPI.setDataMode(SPI_MODE0);
    SPI.setBitOrder(MSBFIRST);

    pinMode(SS, OUTPUT); // Set SS pin as output
    digitalWrite(SS, HIGH); // Deselect the slave
}

void loop() {
    digitalWrite(SS, LOW);  // Select the slave device

    // Send a byte and receive a response
    byte receivedData = SPI.transfer(0x42);

    digitalWrite(SS, HIGH); // Deselect the slave device

    // Use the received data
    delay(1000); // Wait before repeating
}
```

—

## 7. Debugging Tips

- Double-check the SPI pins on your microcontroller and connect them correctly to the peripheral.
- Ensure the SPI clock speed and data mode match the peripheral's requirements (refer to its datasheet).
- Use a logic analyzer or oscilloscope to verify the SPI signals.
- If communication fails, check that the `SS` pin is toggled correctly to select the slave.

—

## Conclusion

Using the SPI library in Arduino IDE makes programming SPI peripherals straightforward. By following this guide, you can configure and communicate with SPI devices using basic functions such as `SPI.begin()` and `SPI.transfer()`. Always refer to the peripheral's datasheet for proper configuration.

# Programming I2C peripheral

I2C (Inter-Integrated Circuit) is a two-wire communication protocol used to interface microcontrollers with peripheral devices such as sensors, displays, and memory modules. In Arduino IDE, the `Wire` library is used to facilitate I2C communication. This guide explains how to program an I2C peripheral using the Arduino IDE.

—

## 1. Hardware Connections

I2C communication requires two main lines:

- **SDA (Serial Data Line):** Transfers data between the master and the slave.
- **SCL (Serial Clock Line):** Synchronizes the data transfer.

Connect the I2C pins of your microcontroller to the corresponding pins on the peripheral device:

- On most Arduino boards:
  - **SDA:** A4 (e.g., on Arduino Uno) or dedicated SDA pin.
  - **SCL:** A5 (e.g., on Arduino Uno) or dedicated SCL pin.
- Refer to the pinout diagram of your board for accurate I2C pin locations.

Pull-up resistors (typically 4.7 kΩ) are required on the SDA and SCL lines to ensure proper communication, though some peripherals may already include them.

—

## 2. Include the Wire Library

Include the `Wire` library in your sketch to enable I2C communication.

```
// Include the Wire library
#include <Wire.h>
```

—

## 3. Initialize I2C Communication

Initialize the I2C communication in the `setup()` function:

- **Master Mode:** Use `Wire.begin()` to initialize the microcontroller as the I2C master.
- **Slave Mode:** Use `Wire.begin(address)` to initialize the microcontroller as a slave with a specified 7-bit address.

Example for Master Mode:

```
void setup() {
    Wire.begin(); // Initialize as I2C master
}
```

Example for Slave Mode:

```
void setup() {
    Wire.begin(0x08); // Initialize as I2C slave with address 0x08
}
```

—

## 4. Communicate with an I2C Device

### 4.1 Writing to an I2C Device

To send data to an I2C device (slave), use the following steps:

1. Call `Wire.beginTransmission(address)`, where `address` is the 7-bit I2C address of the slave device.

2. Use `Wire.write(data)` to send data to the slave.

3. Complete the transmission with `Wire.endTransmission()`.

Example:

```
void loop() {
    Wire.beginTransmission(0x3C); // Start communication with
        device at 0x3C
    Wire.write(0x01);             // Send a command or data byte
    Wire.endTransmission();       // End the transmission
    delay(1000);                  // Wait before sending again
}
```

**Note:** Replace `0x3C` with the correct I2C address of your peripheral.
—

### 4.2 Reading from an I2C Device

To read data from an I2C device:

1. Request data using `Wire.requestFrom(address, quantity)`, where `address` is the 7-bit I2C address of the slave, and `quantity` is the number of bytes to read.

2. Use `Wire.available()` to check if data is available.

3. Read the data using `Wire.read()`.

Example:

```
void loop() {
    Wire.requestFrom(0x3C, 2); // Request 2 bytes from device at 0x3C

    while (Wire.available()) { // Check if data is available
        byte data = Wire.read(); // Read a byte
        Serial.println(data);    // Print the received data
    }
    delay(1000); // Wait before reading again
}
```

—

## 5. Complete Example Code

Here is a complete example for communicating with an I2C device in Master Mode:

```
// Include the Wire library
#include <Wire.h>

void setup() {
    Wire.begin();              // Initialize as I2C master
    Serial.begin(9600);        // Start the Serial monitor
}

void loop() {
    // Write to the I2C device
    Wire.beginTransmission(0x3C); // Start communication with device at 0x3C
    Wire.write(0x01);             // Send a command or data byte
    Wire.endTransmission();       // End the transmission

    delay(500);

    // Read from the I2C device
    Wire.requestFrom(0x3C, 2); // Request 2 bytes from device at 0x3C
    while (Wire.available()) { // Check if data is available
        byte data = Wire.read(); // Read a byte
        Serial.println(data);    // Print the received data
    }

    delay(1000);
}
```

—

## 6. Debugging Tips

- Ensure that the SDA and SCL pins are connected correctly.
- Use a logic analyzer or oscilloscope to verify I2C signals.

- Check the device's datasheet for the correct I2C address and communication protocol.
- If communication fails, try adding pull-up resistors (4.7 kΩ) to the SDA and SCL lines.
- Use the `I2C_Scanner` sketch to detect connected devices:

```
// Include the Wire library
#include <Wire.h>

void setup() {
    Wire.begin();
    Serial.begin(9600);
    Serial.println("I2C Scanner");
}

void loop() {
    for (byte address = 1; address < 127; address++) {
        Wire.beginTransmission(address);
        if (Wire.endTransmission() == 0) {
            Serial.print("I2C device found at address 0x");
            Serial.println(address, HEX);
        }
    }
    delay(1000);
}
```

—

## Conclusion

The `Wire` library in Arduino IDE makes it easy to program I2C peripherals. By following the steps in this guide, you can configure and communicate with I2C devices using functions like `Wire.begin()`, `Wire.write()`, and `Wire.requestFrom()`. Always consult the device's datasheet for proper configuration.

## Works Cited

1. Google Document 1: `EE 156 Board Requirements`

   Description: *[Write a brief description of this document here. For example, what the document is about or its purpose.]*

2. Google Document 2: `Final Project EE 156`

   Description: *[Write a brief description of this document here. For example, what the document is about or its purpose.]*

3. Google Document 3: `EE 156 Final project Planning`

   Description: *[Write a brief description of this document here. For example, what the document is about or its purpose.]*

4. Data Sheet: `Samd51 Family Datasheet`

   Description: *Data sheet for the SAMD51 Family, useful for pin mappings*

5. Custom SAMD51 Board (Add pins in variant files) Issue - Microcontrollers - Arduino Forum: `Custom SAMD51 Board (Add pins in variant files) Issue` Description: *This forum discussion provides detailed steps and troubleshooting tips for modifying the variant files of a custom SAMD51 board to enable additional pins for analog and digital functionality.*

6. SAMD51 with the Arduino IDE - Microcontrollers - Arduino Forum: `SAMD51 with the Arduino IDE` Description: *This thread explores how to set up and use SAMD51-based boards like the Adafruit Grand Central M4 with the Arduino IDE, including driver installation and board package configuration.*

7. SAMD51 Thing Plus Hookup Guide - SparkFun Learn: `SAMD51 Thing Plus Hookup Guide` Description: *A comprehensive guide for the SparkFun SAMD51 Thing Plus, covering hardware setup, code examples, and integration with the Arduino IDE for project development.*

8. SAMD51 Thing Plus Graphical Datasheet: `SAMD51 Thing Plus Graphical Datasheet` Description: *A graphical datasheet for the SparkFun SAMD51 Thing Plus, providing an overview of the board's features, pinouts, and specifications in an easy-to-read format.*

9. samd51j20a.h at master · microsoft/uf2-samdx1 · GitHub: `samd51j20a.h at master` Description: *A header file from Microsoft's UF2-SAMDX1 repository, detailing the pin and port configurations for the SAMD51J20A microcontroller, useful for low-level programming and configuration.*

10. Adafruit/microsoft UF2 github: `sparkfun-samd-1.8.9.tar.bz2 at main` Description: *A downloadable package containing board definitions and support files for SparkFun's SAMD boards, compatible with the Arduino IDE for streamlined development.*

11. Default Sketchbook Folder Location - IDE 1.x - Arduino Forum: `Default Sketchbook Folder Location` Description: *This brief forum post explains where the default sketchbook folder is located in Arduino IDE 1.x, providing guidance for users configuring or locating their Arduino projects.*

## 1 Glossary

| **Abbreviation** | **Full Form and Description** |
|---|---|
| **EIC** | **External Interrupt Controller**: Manages external interrupt requests triggered by pins or events. |
| **ANAREF** | **Analog Reference**: A reference voltage used for analog peripherals, such as ADC and DAC. |
| **ADC0 / ADC1** | **Analog-to-Digital Converter 0 / 1**: Two independent ADC modules that convert analog signals into digital values. |
| **AC** | **Analog Comparator**: Compares two analog input voltages and outputs a digital signal based on their relative values. |
| **DAC** | **Digital-to-Analog Converter**: Converts a digital value into an analog signal (e.g., voltage output). |
| **PTC** | **Peripheral Touch Controller**: Specialized hardware for capacitive touch detection. |
| **SERCOM** | **Serial Communication Interface**: A flexible module that can be configured as UART, SPI, or I2C for serial communication. *(e.g., SERCOM0, SERCOM1).* |
| **TC** | **Timer/Counter**: A general-purpose timer module that supports basic timing and counting operations. |
| **TCC** | **Timer/Counter for Control Applications**: An advanced timer module for PWM generation, motor control, and other applications requiring precise timing. |
| **PDEC** | **Position Decoder**: A module that processes signals from rotary encoders or other position sensors. |
| **QSPI** | **Quad Serial Peripheral Interface**: An extension of SPI using four data lines for faster data transfer, often used for external flash memory. |
| **CAN0 / CAN1** | **Controller Area Network 0 / 1**: Two independent CAN modules for communication in automotive and industrial systems. |
| **USB** | **Universal Serial Bus**: A standard interface for connecting peripherals to the microcontroller. |
| **I2S** | **Inter-IC Sound**: A protocol for transmitting audio data between digital audio devices. |
| **GMAC** | **Gigabit Media Access Controller**: A peripheral that handles Ethernet communication in the microcontroller. |
| **SDHC** | **Secure Digital High Capacity**: A controller for managing SD cards and high-capacity storage devices. |
| **PCC** | **Parallel Capture Controller**: A peripheral for image or data capture from external devices like cameras. |
| **GCLK** | **Generic Clock**: A clock generation system that provides configurable clock signals to peripherals. |
| **CCL** | **Configurable Custom Logic**: A programmable logic block for creating custom hardware logic without external components. |

| Abbreviation | Full Form and Description |
|---|---|
| **CORTEX_CM4** | **ARM Cortex-M4**: The microcontroller's processor core, designed for high-performance and low-power embedded applications. |
| **ADC** | **Analog-to-Digital Converter**: Converts an analog signal (e.g., voltage) into a digital value for the microcontroller. |
| **DAC** | **Digital-to-Analog Converter**: Converts digital data into an analog signal (e.g., voltage). |
| **PWM** | **Pulse Width Modulation**: A method to control power delivery or simulate an analog output using digital signals. |
| **UART** | **Universal Asynchronous Receiver-Transmitter**: A hardware protocol for serial communication. |
| **SPI** | **Serial Peripheral Interface**: A communication protocol for high-speed data transfer between a master and one or more slave devices. |
| **I2C** | **Inter-Integrated Circuit**: A two-wire communication protocol for interfacing peripherals with microcontrollers. |
| **MISO** | **Master In, Slave Out**: Data line for sending data from the slave to the master in SPI communication. |
| **MOSI** | **Master Out, Slave In**: Data line for sending data from the master to the slave in SPI communication. |
| **SCK** | **Serial Clock**: Clock signal used in synchronous serial communication (e.g., SPI/I2C). |
| **SDA** | **Serial Data Line**: Data line used in I2C communication. |
| **SCL** | **Serial Clock Line**: Clock line used in I2C communication. |
| **FS** | **Frame Sync**: Signal used in I2S communication to synchronize audio data frames. |
| **MCK** | **Master Clock**: Main clock signal in I2S communication, used to synchronize audio devices. |