Blog-introduction

“A feature-rich MCU with integrated Wi-Fi and Bluetooth connectivity for a wide range of applications”

- Developers of ESP32,

Espressif systems

GPS tracker, weather systems, security cameras, smart watches, or even a live-streaming robot car

Not just these but many more projects are becoming viable as the days pass for a common man. Be it DIY or IoT, the banner is irrelevant, these are projects that can be made by an enthusiastic hobbyist, within the budget.

And a major part of this is due to the processor at the center. Many chips, with the advent of Arduino in 2005, have a significant role in making all this possible.

Among these, ESP32 is one of the new ones, debuting as the successor of the ESP8266.

So, in this blog series, I wanted to focus on the communication peripherals present within the ESP32. By communication, I mean wired communication.

What is a peripheral, you might ask. They are devices that connect to your chips from outside. Simply think of them as devices that aren't integrated with the chip.

Their purpose, in simple terms, is to automate tasks for freeing up the processor. Be it rules or encoding, peripherals perform those tasks, so your processor can simply write and read from it without spending much of its valuable resources on it.

And in this blog series, I will speak about communication peripherals. These are peripherals that help you send data from one device to another, be it a sensor or another processor like your computer.

What really is ESP32?

Ever wonder, what is an ESP32? What does this board contain or what does WROOM32-D do different from your Adafruit board?

To answer that, you would first need to know what an ESP32 is. So, let us start, what is an ESP32?

This board here?

<>

</> [/ pinout of doit esp32 devkit v1]

<>

Well, not so correct. This is the ESP32 development board, specifically the doit ESP32 devkit v1.

The development board, or dev board for short, simplifies the job of using the ESP32 chip it sports. It allows to directly use jumpers and wires to connect the pins of ESP32, a USB-UART bridge chip so that you can use that USB cable to communicate with the ESP32 chip, a voltage regulator to free you from supplying exactly 3.3V, on board LEDs and even the boot and enable buttons.

Of course there is a bigger list of features integrated components and we can sit here talking about them all day. But I wanted to speak about one chip in particular.

You see this chip here?

<>

</> [/ picture pointing to the bridge on dev board]

<>

This is the USB-UART bridge chip which is responsible for making sure the data from processor is sent as per USB standards. It is just nothing but correcting the voltage range of signal, nothing complex.

Generally, these chips are numbered CP210x or CH340 and their software drivers should be bundled with your OS.

In case they aren’t, your computer might not recognize the dev board connected, so make sure to download them

from the silicon labs official website [/ hyperlink to the Silicon labs site].

Now tell me, what is an ESP32? Is it the chip here?

<>

</> [/ highlight the module on dev board]

<>

Well, it is only partially right. This is a MCU module, which would be WROOM32 module for doit devkit v1.

<>

</> [/ ESP32 Wroom32 pinout]

<>

Notice something different here? The total pins on the board were 30 pins but the MCU module has 38 pins.

Yes, a few pins were used by the board to make sure you would have all those features I mentioned and didn’t.

On this MCU module, you should be able to see a Wi-Fi antenna on the top, a few capacitors and resistors, a flash memory chip at the bottom and an analog quartz crystal on the right along with the chip.

Other MCU modules might have slightly different add-ons compared to this, but you get the gist.

Now, what is ESP32 again? Is it this chip here that has the Espressif logo?

<>

</> [/ point out the chip on wroom32 module]

<>

Absolutely, it is still partially correct.

<>

</> [/ ESP32 MCU unit pinout from datasheet]

<>

And yes, there is another jump in number of pins from 38 to 49. But there are a lot of problems with directly using this board. The power supply isn't as easy as connecting a 3.3V source to one of the VCC pins, there is no flash memory, no Wi-Fi antenna and many more.

But this, this is the ESP32 DOWDQ6 chip. You see, ESP32 is the name of a series of MCU units developed, just like how Samsung Galaxy is a series of a few Samsung phones, not a particular mobile phone.

For all those wondering, MCU is MicroController Unit.

There is another interesting thing about the ESP32 MCU. The ESP32 isn't an MCU, but it is a SoC MCU.

Take a look at this block diagram:

<>

</> [/ image of block diagram from ESP32 datasheet]

<>

As you can see, this contains not just the microcontroller at the middle (the core section) but also devices for Wi-Fi, Bluetooth, RTC (Real Time Clock), hardware acceleration and RF along with another bunch of peripherals on the left.

So, as you can see the Core and memory at the middle is the microcontroller, but combining many other chips with it makes it a SoC (System on Chip).

In this series, I am going to talk about some of these peripherals, so stay tuned.

Why not Arduino?

ESP32 is a popular choice, but Arduino is the popular choice, period.

In short, the comparison doesn’t make sense. ESP32 is very different from Arduino.

When talking about Arduino, we refer to the development boards and its related software environment by the company of name Arduino.

Unlike ESP32, that signifies the microcontroller at the center, Arduino is the name for the development boards. The microcontroller at center is from the ATMega family.

Arduino excels when it comes to beginner and simpler projects. Not only are the boards designed easy enough to use, there is also a dedicated software for programming it, the Arduino IDE. Not to mention the extensive support online for issues related to Arduino.

But in the end, the chip at the center of Arduino, the one that is the heart of these boards, is a simple ATMega chip. They are fit for simpler projects with their memory, clock speed and processor power.

ESP32, on other hand, is made for higher and more complex projects. With dual core 32-bit processors, it has both integrated Bluetooth and Wi-Fi. Not to mention the support for numerous different types of peripherals, ESP32 is simply not built for beginner projects.

Before we dive into the series…

The blog series explains a lot about the peripherals and their workings. But that doesn't mean it will cover all the way from zero. There are prerequisites.

First is a platform on which you can code your ESP32 board. You may use Arduino IDE with the ESP32 core, platform IO extension of VS Code or anything that works.

Next is knowledge of the programming language. A basic understanding of data types, functions and pointers in C language is a must.

Also, you should be familiar with simple functions that are used for hardware programming like setup() or digitalRead().

With all that said, we are ready to dive into the series. In the first part of the series, the next blog, I will cover about UART. So, stay tuned for it.

Blog series- 1.UART

Welcome back guys!

This is the first blog of the series, where I go over the different peripherals of ESP32. In this blog, let us tackle the UART peripheral.

UART is a serial communication protocol. The UART peripheral, thus, is a translator for converting data from processor to UART standards and vice versa.

In order to better understand it, let us take a brief overview of the protocol.

1.1 UART protocol

UART, which stands for Universal Asynchronous Receiver / Transmitter, is a serial communication protocol. So, what is a serial communication protocol?

You see, processors *speak* in bytes. A byte is also the smallest unit of data that a processor can understand. But to transmit a byte of data, you would need to send 8 bits, physically translating to 8 independent lines.

But this makes the hardware expensive and more space consuming. So, rather than sending all 8 bits at once like parallel communication, you send in one bit after another in serial communication. This decreases the number of lines required from 8 to 1.

The rules for breaking down the bytes into bits and stitching them back into bytes, for sending and receiving the said bits are the serial communication protocols.

Among them, the UART protocol is the simplest and a widely used protocol. You even use this protocol when you upload the code to your ESP32! (Remember the UART-USB bridge chip?)

This is due to two reasons. First is that UART uses only two lines to send and receive data at the same time. One line is dedicated to sending and other to receiving.

This method of communication is called full-duplex communication. In this method, any data you receive is

directed into the Rx buffer, which holds the data until you read it from. This makes it possible to both receive and send at the same time.

The second reason is that the devices on both sides need not be synchronized. Only the UART peripherals have to tune to the same data transfer rate, referred to as baud rate. Baud rate specifies how the rate at which bits are sent or received from the peripheral, measured in bps or bits per second.

Although, since UART is asynchronous, the data transfer rate isn't too high. Standard baud rates include:

9600, 14400, 19200, 38400, 57600, **115200**, 230400,460800, 921600.

The baud rates in bold, 9600 and 115200, are commonly used baud rates for Arduino and ESP32 respectively.

Once the baud rate is matched, data will be exchanged in the form of packets.

<>

</> [/ UART data packet structure]

<>

A UART packet can consist anywhere between 7 bits to 11 bits. Each bit takes one clock cycle with the clock frequency being equal to the baud rate.

The packet starts with a start signal. The transmitting line, which is by default left at high state, is pulled low for one cycle.

Next, the data is sent one bit after another. This is generally 5 to 8 bits long. If there is no parity bit, it can be 9 bits long too. We generally send 8 bits only.

The parity bit is sent after the data is sent. If the parity of data received is the same as the parity bit received, the data is received properly.

Finally, the stop signal is sent. The transmitting line is driven high for one or two clock cycles.

Number of bits sent, whether the parity bit is sent and how many stop bits are configurable settings like baud rate and thus, should also be the same for both transmitter and receiver.

Before we dive into how to set those conditions or perform some data transfer, let us see what the ESP32's UART peripheral offers.

1.2 UART in ESP32

ESP32 has not one but three independent UART peripherals. The peripherals are referred to as UART0, UART1 and UART2.

Each peripheral’s port has two pins. The Rx pin, for receiving and the Tx pin, for transmitting the data. There are default GPIO pins for each port, available on the chip.

</>Table title: default pins of UART ports

Ports	Rx pin	Tx pin
UART0	GPIO3	GPIO1
UART1	GPIO9	GPIO10
UART2	GPIO16	GPIO17

</> [/ align the table in center]

Although they are the default pins, some of them aren’t available on the dev board.

Specifically, the default pins of UART1 are missing. This is because those pins are used to communicate with the flash memory chip of the MCU module, through another peripheral.

But ESP32 allows you to connect any pins to the peripherals through its IO mux. So, you can use any pins besides the default ones as the Tx and Rx pins of the peripheral. Just make sure the pins you select can function as Tx and Rx pins.

<>

</> [/ devkit pinout]

<>

Speaking of occupied ports, UART0 is, well, partially occupied. This has to do with the serial monitor. When you use the serial monitor, what you are using is the UART0 port.

The default pins of UART0 are connected to the USB-UART bridge chip on board internally, thus any data sent from UART0 port through the default pins specifically is also sent to the computer, which is displayed on the Serial monitor.

So, in case you are using the serial monitor, do not allocate the GPIO3 and GPIO1 pins to any other uses.

That brings us to the UART2. The default pins of the UART2 port are completely free. It is recommended that you use this port if you are already using the serial monitor for debugging or just message printing.

Now that we have covered the protocols and the ports of UART, let us see how you actually code them to send and receive data through the GPIO pins.

1.3. Coding the UART peripheral

Generally, there are classes for such peripherals and other features. The classes have the functions we need written into them, so all we need to do is initialize an object of the class and use the functions to exchange data.

For the UART peripheral, there is the ‘HardwareSerial’ library. Let us find the library’s header and source code.

You may go through the IDE files and figure out where the library is, but that is a daunting way of doing this. Let us use a cheat instead.

In order to use this cheat, you need to know at least a command of the library and one of their arguments. In my case, we will be using ‘*Serial.begin()*’.

So, in setup, after including the library, write this code:

<>

</> *Serial.begin(“Wrong”);*

<>

This is obviously a wrong argument (come on, that was good…). Run the code. If your IDE is well and fine, you will get a compilation error. Notice something yet?

<>

</> [/picture of hardware serial compilation error]

<>

Yes, within the error message, it shows where the error is reflected from. In other words, it shows the location of the library we want. Go to the location.

You might find two files with the same name as the library. One is a header file, with .h extension, which contains the functions definition only and another is the source code file, with .cpp extension, which has the code for the functions defined in the header file.

Now, go through both of them. Use the source code file to understand the functions' workings and the header file to know how they are defined and what all are defined.

By default, objects named Serial, Serial1 and Serial2 are defined and connected to the UART0, UART1 and UART2 respectively.

You must be familiar with 'Serial' as it is the object you use whenever you communicate with the serial monitor. Since HardwareSerial is a core library, you need not specifically write the include statement but it is a good habit to do so.

In case you want to define your own object, you can use the same format used for defining and connecting them:

<>

</> *HardwareSerial <name here> (<port number here>);*

<>

Remember to declare this globally as you will use it in at least setup and loop functions.

The first function we use is the begin function. As the name suggests, it begins the connection.

In the header file, it is shown to have 6 arguments and all except the first argument, the baud rate, have a default argument.

<>

</> *begin(unsigned long* ***baud****, uint32_t* ***config****=SERIAL_8N1, int8_t* ***rxPin****=-1, int8_t* ***txPin****=-1, bool* ***invert****=false, unsigned long* ***timeout_ms*** *= 20000UL)*

<>

The baud rate was already explained, so let us see the config argument. By default, it is set to SERIAL_8N1. The 8N1 means 8 data bits, No parity bit and 1 stop bit per packet. This is the one that sets the settings for communication.

Furthermore, there are multiple configs except SERIAL_8N1 available.

<>

</> [/ image of serial configs]

<>

Now comes the pins. First, we pass the argument for the Rx pin and then the Tx pin. Again, you could just not pass any and they will initialize with the default ones mentioned in the table[\ hyperlink to the table on default] above.

But if you want any different pins as Rx and/or Tx pins, enter their GPIO number, not the pin number on board.

The invert parameter is used to decide whether the signal we are sending/receiving is inverted or the original. This is used when the device we are talking to perceives a different logic level, 0 for high and 1 for low.

The last parameter, the timeout_ms is used when the baud rate isn't given. If you are trying to communicate with a device whose baud rate was prefixed and unknown, you can just pass on 'null'.

Within the time given by timeout_ms in milliseconds, the processor tries to figure out the baud rate of the other devices. So, timeout_ms dictates how long the processor can try to figure out the unknown baud rate.

Once the baud rate is found, the peripheral is also set to the same baud rate. If not, an error log is written.

Now, let us use an example. I am using the object 'test', plugging it with the UART2 port, and setting the GPIO18 and GPIO19 as the Rx and Tx pins respectively at a baud rate of 9600. The code will be:

<>

</>

#include <HardwareSerial.h>

```
HardwareSerial test (2); // creating the object test

void setup(){
  test.begin(9600,SERIAL_8N1,18,19);
}
void loop(){
//nothing here yet
}
</>
<>
```

Once initialized, we are ready to exchange data through our UART pins.

To read data from the Rx pin, we can use the read function. Read function actually reads the data from the Rx buffer. Since the communication is full duplex, there is a chance of receiving data while sending, thus the received data is placed in the Rx buffer until the processor wants to read it.

In the header file, there are three different versions of the read function.

<>

</>

```
size_t read(uint8_t *buffer, size_t size)
```

*size_t read(char *buffer, size_t size)*

int read(void)

</>

<>

Here, size_t means unsigned integer of largest size processed by the processor. On a 32-bit processor, it will be 32 bit long and on a 64-bit machine, it will be 64 bits long.

The first two functions are for reading multiple bytes while the third function, the one with no arguments, returns the oldest byte the Rx pin received.

For the first two functions, the first argument is a pointer to where the read bytes will be stored and the second argument defines how many bytes to read.

Notice that I am referring to the data as bytes, not characters or integers. This is because the UART exchanges the bytes, not what type they are. That is something you should be aware of.

Also, the read function returns the number of bytes it reads. You can check it with the number of bytes you passed for, in case the Rx buffer has less data than you requested, they might not match.

Next, let us look at the write function. Similar to read, write function also has multiple definitions in the header file.

<>

</>

size_t write(uint8_t data);

size_t write(unsigned long data);

size_t write(long data);

size_t write(unsigned int data);

size_t write(int data);

*size_t write(const uint8_t *buffer, size_t size);*

*size_t write(const char *buffer, size_t size);*

*size_t write(const char *s);*

<>

For the first 5 definitions, they only send in 8 bits of data. Be it a 16 bit signed integer or 32 bit long unsigned integer or a 8 bit character, they send in only the LSB 8 bits of data.

Remember, the data types are assumed by the processor, physically the bits are all the same.

The next two can be used to send multiple bytes of information, the first byte of which is pointed by the buffer pointer while the size defines how many bytes to send.

The last definition is specifically used for strings. All the definitions return the number of bytes they have written.

Now, let us discuss functions that are not used for setting up the channel or exchanging data but monitoring the status of communication.

The available function tells us the number of characters available in the Rx buffer, ready to be read. In most cases, it is used to check if there has been any input on the Rx pin.

The flush() function stops the code from running until everything in the Tx buffer is sent. Generally, you write the data you want to send to the Tx buffer and the UART peripheral takes care of sending in the background while your processor proceeds with the code. But this function stops the processor from executing further code until the peripheral gives the signal that the Tx buffer contents are sent. This slows down the code but makes sure the data is transferred. The Rx buffer though is unaffected.

Next is the setRxBufferSize(). It takes one argument on how many bytes the buffer should be. The Rx buffer is first in first out, so if the older data hogs all the memory, the newer ones aren’t read. But if you can’t clear data in time, you can increase the buffer size using this function.

Last is the BaudRate() and updateBaudRate(). First takes no argument and returns the baud rate. Second returns nothing and takes one argument which is the new baud rate.

Endnotes

And here ends the blog. I have discussed all I have set to discuss about the UART port. If you have any doubts, no matter how stupid it is, just ask in the comments.

So far, what we discussed in the blog are:

- UART is an asynchronous serial communication protocol that is widely used for communication between two processors.
- While not fast, it allows transfer between two devices with only 2 lines and is easy to set up as it doesn't require the synchronization between two devices.
- ESP32 has 3 UART ports. UART0, along with its default pins, is used for the serial monitor. UART1 is unoccupied but its default pins are used for connecting to the flash chip on your development board. UART2 is completely free.
- The library and class to use UART is HardwareSerial. UART0, UART1 and UART2 are already registered as Serial, Serial1 and Serial2 respectively by default.
- Syntax to instantiate an object of HardwareSerial class is

</> HardwareSerial <object name> (<port number>);

- Use </> *begin(<baud rate>, <serial mode>, <Rx pin>, <Tx pin>)* </> function to configure the pins. For default settings of serial mode and pins, just pass in the baud rate.
- Use the </>*write()*</> and </>*read()*</> functions to write and read from the UART port.
- Use the </>*available()*</> function to check if the UART port has received any data, waiting to be read.
- Use the </>*flush()*</> function to make sure you have sent all characters on the Tx buffer before proceeding.

And here ends the blog. I have discussed all I have set to discuss about the UART port. If you have any doubts, no matter how simple it is, just ask in the comments.

Until next time peeps, stay tuned.

Blog series- 2.SPI

Welcome back guys!

This is the second blog of the series. For all the first timers, this blog is part of a series where I go through several peripherals of ESP32. In the previous blog, I have covered UART, so if you are interested, check that out.

In this blog, I am going to cover the SPI peripheral. SPI is also a serial communication protocol, but is different from the UART protocol discussed in the previous blog.

In order to get a better grip of that, let us see what the SPI protocol is first.

2.1 SPI protocol

** don’t worry. This is the first section in this blog. But in the overall series, it is the second peripheral to cover, thus the index.

SPI stands for Serial Peripheral Interface. This is also a serial communication protocol.

If explained via an example, serial communication means speaking word after another, rather than saying all the words of the sentence at once. In the computer world, this reduces how many cables or lines you need to use.

SPI is a synchronous protocol, thus both the devices speaking are in sync via the common clock signal. The clock tells the device when to sample for bits, similar to making sure devices are on the same wavelength for speaking.

Another feature of synchronous communication is the master-slave devices. A slave is a device that can only

read or write data when the master requests the same. AS such, the master is the one who produces the clock signal too.

SPI can support multiple slaves but only one master. Additionally, the clock frequencies of SPI are generally high, in MHz. Even ESP32 has a default SPI clock frequency of 1MHz.

SPI also uses the duplex mode of communication, where the devices can send and receive data at once. The incoming data is stored in a buffer, which the processor can read at its own pace.

Speaking in terms of speed, with default clock frequencies of 1MHz, no ‘head’s up’ signals, SPI can achieve quite the data transfer rates.

Compared to UART, which has default bit transfer rates of 115200 and the need for 2 to 4 head’s up for every 8 bits(say), the speed is really high.

But the problem comes with actual hardware implementation. UART only uses 2 lines, one for sending and one for receiving. But SPI uses at least 4 lines for data transfer.

SCK or Serial ClocK line transmits the synchronous clock signal produced by the master. MOSI or Master Out Slave In line acts as transmitting end when in master mode and receiving end when in slave mode. For devices that can

only be slaves, it might be marked as SDI or Serial Data In.

Similarly, there is MISO or Master In Slave Out that acts as transmitting end when in slave mode and receiving end when in master mode. For slave-only devices, it might be marked as SDO or Serial Data Out.

These are common lines which can be connected in parallel to all the slaves. Just make sure the last letters map. And then, we have slave specific lines, the CS or SS line.

The CS line is an indicator for the slave to be ready for transmission. When set low, the corresponding slave participates in data exchange. When set high, it will ignore the data exchange happening.

This means that for every slave, there needs to be their own CS pin. This is also the reason why at most, SPI uses only three slaves, else there would be too many pins for the peripheral.

<>

</Insert a picture of SPI connections here, at least 2 slaves>

<>

Think of the CS pins as the ID for the slaves, which is active low. To talk with a certain slave, activate only its CS

pin(set low) and deactivate all other slave’s CS pins(set high).

2.2 SPI in ESP32

ESP32 has four SPI peripherals, namely SPI0, SPI1, SPI2 and SPI3. Each port of the SPI peripheral has at maximum three CS pins.

Of the four SPI peripherals present, two peripherals are already used in your dev board. SPI0 is used as buffer for external memory while SPI1 is used for connecting to the flash memory on your MCU module. Thus, both of them are not present on the dev board, and even if they are, do not use them.

Additionally, one should not reroute the port pins of SPI0 or SPI1. Unlike with UART1, whose default pins were used, SPI0 and SPI1 ports along with their default pins are used, so stay away from them.

SPI2 and SPI3, or commonly referred to as HSPI and VSPI, are free for usage and VSPI is the default SPI port used.

<>

</> [/ pinout of doit devkit v1]

<>

The SPI_Ds and SPI_Qs in the diagram are the MOSI and MISO respectively of the respective SPI ports. So, V_SPI_D refers to the MOSI of the VSPI port and so on.

By default, when you initiate SPI without specifying the port, the VSPI port is used and the default pins are used.

Now, even if you are using different boards, you can easily find your default pins. They are marked MOSI, MISO, SCK and SS on board.

If the markings aren't sure or if they are not given, worry not. The default pins are already named constants in the library. Thus, </>*Serial.println(MOSI);*</> prints the MOSI pin number on the Serial monitor. This is only for the default SPI pins, which means default pins of VSPI.

Similar to UART, ESP32 allows you to use any of the available GPIO pins for the VSPI and HSPI port.

Additionally, there is another special feature about the serial clock.

The clock has four different modes called SPI_MODE0, SPI_MODE1, SPI_MODE2 and SPI_MODE3 in the library. These refer to the states of idle configuration and sampling configuration.

Idle state is the state of the SCK line when there is no transmission happening, whether it is high or low. And the sampling state refers to on which edge of the clock signal

the peripheral should read in data and store it in the Rx buffer.

This also means the peripheral sends out data on the other edge. For example, SPI_MODE0 is idle low and sampling out at rising edges. Thus, when there is no transmission, the clock line is set to 0 and when communicating, at the rising edge, the data is sent out while data is read in at the falling edge.

Similarly, Mode 1 is idle low and sending at a falling edge. Mode 2 is idle high and samples out at falling edge while Mode 3 is idle high and samples out rising edge.

<>

</> [/ four clocks for four modes with sampling and shifting]

<>

Now that we have covered the necessary information, let us see how to put that in code.

2.5. Setting up and using SPI

Buckle up guys, SPI isn't a simple library, unlike UART.

First of all, SPI is majorly used to communicate between sensor and board. That obviously means there are libraries for most sensors out there, which take care of

writing and reading using the SPI peripheral. We can only modify the setting up of the SPI port part.

But there would be cases where you want to use your own library for using the sensors. Whatever the reason, you want to learn how to use SPI peripherals, not just setup but also read and write.

For people of both types, you can continue. Further down the lane, I will tell where the people of the first type can switch over to the next section.

For master mode, we use the SPI library. Or more precisely, the SPI library supports only master mode. So, first let us explore this.

To locate it, remember the cheat. You need to know one function of this library. Universally, there is the begin() function. Now, just give it a wrong argument.

<>

</>

```
#include <SPI.h>


void setup(){

  SPI.begin("Wrong");

}
```

```
void loop(){

}
```

<>

Of course, you could leave it blank and compile but there might be default parameters.

Here I have used the SPI object, which is the default SPI mapped to VSPI port and VSPI default pins onboard with default settings. More on all that later.

If you compile an error message and in the error message, you should be able to find the library location.

Go there and you should find the header and source code file (.h and .cpp file with the same name as library). Go through them, or skim through them.

The library has 2 classes, called SPIClass and SPISettings.

The SPISettings class is used for defining the settings of the SPI communication channel. Do make sure the settings on either end of the channel are the same. Each SPISettings class object has three arguments passed when initializing.

The first is the clock frequency, which will be the frequency of the serial clock in hertz. By default, this will be 1MHz.

Next is the BitOrder that specifies which bit is sent first, whether MSB first or LSB first. You can pass on the

defined constants MSBFIRST and LSBFIRST based on the requirement. Alternatively, the SPI library also defines the respective constants SPI_MSBFIRST and SPI_LSBFIRST. SPI_MSBFIRST is the default for this argument.

And lastly, the data mode which specifies the idle state and sampling state of the clock. As discussed earlier, there are mode0, mode1, mode2 and mode3 corresponding to the defined constants SPI_MODE0, SPI_MODE1, SPI_MODE2 and SPI_MODE3. By default, SPI_MODE0 is set.

So, if you want to initialize the object, named setting_1, with frequency of 100KHz, MSBFIRST order and MODE2, the syntax will be:

<>

</> *SPISettings settings_1 (100000, MSBFIRST, SPI_MODE2);*

<>

Next, we have the main objective, the SPIClass. An object of the SPI class has only one passable argument, which is the SPI bus number.

In case you don’t remember the SPI port number, you can also use the defined constants, VSPI and HSPI directly.

Thus, a sample initialization of a SPI object ‘test_spi’ connected to the VSPI port will be :

<>

</> *SPIClass test_spi (VSPI) ;*

<>

By default, an object named ‘SPI’ is defined, which is connected to the VSPI port.

Once we have defined the object(globally), we can use the begin function to set up the pins for this SPI port.

<>

</> *void begin(int8_t sck, int8_t miso, int8_t mosi, int8_t ss);*

<>

int8_t refers to the 8 bit integer data type. Pass on the pins you want to connect. If no arguments were passed, then the object will be initialized with default pins. For the case of ESP32 doit devkit, the default pins are as follows.

Port	**SCK**	**MISO**	**MOSI**	**SS**
VSPI	GPIO18	GPIO19	GPIO23	GPIO5
HSPI	GPIO14	GPIO12	GPIO13	GPIO15

This once again depends on the library code and such, but the default pins of VSPI are the constants SCK, MOSI, MISO and SS.

The begin() function doesn’t just set the pins but also configures the communication channel with the default settings mentioned above.

In order to change these parameters, you can use the beginTransmission function or their individual functions.

First, let us look at the individual functions to change these settings:

1. void SPIClass:: setFrequency(uint32_t freq) and void SPIClass:: setClockDivider(uint32_t clockDiv)

In the SPI peripheral, the serial clock frequency is obtained using the processor’s own clock. Thus the clock frequencies are directly related through a parameter called clock divider. What this means is, in short, there are discrete frequencies that are accepted.

If you pass on a frequency of 3Mhz, you might end up at 2.87Mhz. In order to find at which frequency the serial clock runs for your written frequency, run this code:

<>

</>

```
#include <SPI.h>
#include "esp32-hal-spi.h"

void report(int freq){
```

```
  int x,y;
  SPI.setFrequency(freq);
  x=SPI.getClockDivider();
  y=spiClockDivToFrequency(x);
  Serial.print("Clock divider:");
  Serial.println(x);
  Serial.print("Frequency:");
  Serial.println(y);
  delay(1000);
}

void setup() {
  int freq=1000000;   //put your frequency here
  SPI.begin();
  Serial.begin(115200);
  report(freq);
  delay(200);
}

void loop() {
  ;
}
<>
```

2. void SPIClass:: setBitOrder(uint8_t bitOrder)

This function is straightforward. The input argument is generally SPI_MSBFIRST or SPI_LSBFIRST, defined constants with values 1 and 0 respectively. Alternatively, you can also pass on predefined MSBFIRST and LSBFIRST constants.

3. void SPIClass:: setDataMode(uint8_t dataMode)

You can pass on one of the defined constants, SPI_MODE0, SPI_MODE1, SPI_MODE2 or SPI_MODE3, so as to set the data mode to the specific mode. They are also defined constants that come with the SPI library.

If you are trying to learn about using SPI to use in sensor libraries, you may hop over to the next section now. But there is no harm in knowing more than just the minimum.

Now, the other way to set these parameters is through the beginTransmission function.

The beginTransmission function is called to lock the resources exclusively for the usage of SPI. When a processor runs the SPI peripheral, a part of its resources are allocated to the peripheral management.

Some of these are common resources, which will be allotted to other peripherals or purposes when SPI peripheral isn’t in idle mode and others are exclusive.

The beginTransmission function restricts the processor from assigning these common resources to other tasks. This is particularly useful when dealing with codes that take up a lot of processor power.

It is important to remember that after calling the beginTransmission function, the endTransmission function should be called once the data exchange has been done. All the data exchange through SPI should be done within these two lines of codes.

<>

</> *void SPIClass:: beginTransmission(SPISettings settings)*

void SPIClass:: endTransmission()

<>

The beginTransmission function takes in one argument, which is an object of the SPISettings class we discussed earlier.

In case you are going to call beginTransmission, you need not bother with setting the channel parameters individually. Just pass them onto a SPISettings object and then pass the object to the beginTransmission function.

Once the SPI channel and port are set up, we can now select the slave to communicate with. This is fairly simple as all we need to do is to set the CS pin of the slave we want to communicate to GND and all other CS pins of

remaining slaves to high. Once the data transfer is done, set the CS pin back to high.

Now, let us take a look at the data transfer functions.

First are the write functions. Unlike the case with the UART library, the functions aren't exhaustive. Thus, you need to be careful with the data types of your arguments.

<>

</>

```
void write(uint8_t data);

void write16(uint16_t data);

void write32(uint32_t data);

void writeBytes(const uint8_t * data, uint32_t size);

void writePattern(const uint8_t * data, uint8_t size,
uint32_t repeat);
```

<>

The first three functions deal with integers. Since integers are stored in 16 bits and long integers in 32 bits, they can't be fully transmitted through just the write function, which only sends in 8 bits. Thus, use the write16 and write32 function for these integer data types.

write8() function is useful for sending character data types that are stored in 8 bits or smaller integers that can be stored within the 8 bits.

Additionally, the peripheral doesn't differentiate between signed and unsigned integers, as long as it is within the bit range. So, we need not worry for these cases.

Next, we have the writeBytes() function which can be used to send in strings or arrays of characters. The first argument is the pointer to the first character and the second argument is the number of bytes/characters to be sent.

writePattern() is similar to writeBytes(), with the first two arguments, put in a loop for 'repeat' number of times.

Next up, we have the receiving part or the transfer functions.

The SPI protocol we use is full duplex communication. Adding in the presence of the master, SPI allows slaves to send data only when a request from master arrives, for which data from master should be taken in.

So, instead of just a read function, we have transfer functions that send the data within the passed argument and then read in data from the slave into the same argument.

Following are the transfer functions:

<>

</>

uint8_t transfer(uint8_t data);

uint16_t transfer16(uint16_t data);

uint32_t transfer32(uint32_t data);

*void transfer(uint8_t * data, uint32_t size);*

*void transferBytes(const uint8_t * data, uint8_t * out, uint32_t size);*

*void transferBits(uint32_t data, uint32_t * out, uint8_t bits);*

<>

The first three functions are meant to exchange 8 bits, 16 bits and 32 bits of data respectively. They are mainly used to exchange integer, or character in case of 8 bits, data types. The data within the passed variable is sent and received data is written into the variable and returned by the functions.

Additionally, the transfer function has another definition with two arguments, one is a character pointer and the other is the number of characters to exchange.

The transferBytes function sends out the </size> number of bytes(or characters as they are 1 byte long), starting from the character pointed by the </data> pointer. Similar number of bytes are also read in, starting from the location pointed by the </out> pointer.

The last function, transferBits function, is used for exchanging </bits> number of bits of data stored in the

data variable. Then, a similar number of bits are read in, starting with the location stored in the </out> pointer.

Also, note that the </data> argument is a uint32_t data type, thus can contain a maximum of 32 bits. Hence, the maximum number of bits exchanged by this function are 32, requests for more bits would have to be sent in separately.

All these functions are specific about their data types of the arguments, so typecast them in case of some errors about wrong data type given.

Ex: When the uint8_t data type is used, characters can also be passed by typecasting them into the uint8_t data type (uint8_t data type literally means unsigned character data type). This is same for character arrays and const uint8_t * data types.

As a matter of fact, when asked for pointer data types, send in pointers for an array element, while making sure the number of bytes exchanged or written does not exceed the space used by the array. Else, there is a chance of overwriting some other program data as it was not explicitly stated for the data exchange purpose.

Thus, be careful with the data types and pointers used when dealing with SPI, or even in general.

2.51. Setting up SPI: for sensors

SPI communication is majorly used to communicate between the microcontroller and different sensors. In most cases, each sensor has its own library, which would take care of the data exchange part.

Thus we only have the freedom to define which SPI port, GPIO pins and communication settings to use. I have defined the default settings in previous section, but here is a quick recap of it:

SCK pin	GPIO18
MOSI pin	GPIO23
MISO pin	GPIO19
CS pin	GPIO5
SPI port	VSPI (object : SPI)
Serial clock frequency	1Mhz
Bit Order	SPI_MSBFIRST
SPI mode	SPI_MODE0

You can let these functions be or pass on your own arguments. But do remember that the BME680 library doesn't facilitate changes for the last three parameters.

For demonstration purposes, let us proceed with the Adafruit BME680 sensor. If any of the readers are referring to other blogs, please note that we are using the 680 variant as opposed to the 280 variant.

I will also explain how to generalize this process for any sensor library, not just for the BME680 sensor. Fair warning though, we are going to open up some libraries, which means things can get messy fast. Unless you are comfortable with scouring through libraries and headers, it is better to ask help online.

This sensor is used for obtaining temperature, pressure and humidity levels. As a bonus, this sensor also supports the I^2C communication protocol in addition to SPI protocol.

First, find the library for the sensor. You can use the method mentioned in the previous section. But since this is an additional library installed, it should be within the staging/libraries folder of your Arduino IDE appdata.

Within the library, there is a class called Adafruit_BME680. To define an object within this class, there are two ways to instantiate, well three technically.

For instantiating objects for SPI, either hardware SPI or software SPI can be chosen.

Hardware SPI means that the microprocessor has hardware support for the SPI protocol, which translates to having either a SPI peripheral or connecting a SPI controller to the microcontroller. The connected pins can be used to define a SPI class object, which has to be passed here.

Software SPI, on other hand, means that there is no hardware support. The SPI device is directly connected to the GPIO pins of the microcontroller, which has to send the data as per the SPI protocols. This eats up a good amount of the microcontroller resources as it has to generate the SPI serial clock while sending and receiving data accordingly.

This also causes the software SPI to not have as high speeds as hardware SPI as some time is inevitably wasted in the setting up.

Since ESP32 has the SPI peripheral we can proceed with the hardware SPI method. The object declaration requires two arguments, one is the CS pin number to which the sensor’s CS pin is connected and the other is a pointer to the SPI object related to the SPI port connected to this sensor.

The declaration and setting various parameters of the SPI class object has already been discussed in the previous section.

For example, if the CS pin is GPIO17 pin and the SPI object is test_spi, then for the object of Adafruit_BME680 called test_bme, the declaration statement will be:

<>

</> *Adafruit_BME680 test_bme (17, &test_spi) ;*

<>

This will define a Adafruit_SPIDevice object with our given SPI port and pins through CS pin and SPI object, while the serial clock frequency, bit order and SPI mode is set to the predefined settings of 1MHz, MSBFIRST and SPI_MODE0 respectively.

Emphasizing again, the channel settings, namely the serial clock frequency, bit order and SPI mode, can't be changed through any function but have to be manually changed in the library itself. But this is a hassle as whenever the library is updated, the settings will be set to the previous ones.

Further, in the begin() function, it can be seen that there is nothing more to define in terms of setting up the SPI parameters. From here on, it is similar to how you would use the BME library on its own.

Endnotes

With that we reach the end of this blog session. In this blog, we have seen what the SPI protocol is, how it works briefly, how it is implemented in ESP32 on a block level and finally how to code for the SPI devices and sensors.

The major points discussed were:

- SPI is a full-duplex, synchronous serial communication protocol.
- It requires at least four lines: 3 common lines, namely the Serial clock, data in and data out lines, and one unique line called CS line per each slave to act as identification.
- Devices communicating are classified as either masters or slaves
- Masters command and slaves follow. Thus, masters drive the serial clock and the CS pin. Setting the CS pin low enables the slave to exchange data and setting it high turns it off.
- ESP32 has 4 SPI ports. The first is by default used for flash memory and the second is used for connecting to the on-board flash chip of your development board.
- Thus, only the last two ports, VSPI and HSPI, are available.
- The SPI library is used for utilizing the SPI controllers of ESP32, configured in master mode.
- The library has a predefined object called SPI, which is connected to the VSPI port and its default pins.

- Further objects can be defined using the following syntax:

</> *SPIClass <object name> (<port name>) ;* </>

- The port name can either be their port number like 2 and 3, or the defined constants VSPI and HSPI.
- After defining the objects, the pins can be connected using the begin function : </>*begin(<SCK pin>, <MISO pin>, <MOSI pin>, <SS pin>)*</>
- The channel settings are set to 1MHz serial clock frequency, MSB first bit order and MODE0 for SPI data mode.
- These can be used using the individual functions: </>*setFrequency(<frequency>)*</>,</>*setBitOrder(<bit order>)*</> and </>*setDataMode(<SPI data mode constant>)*</>
- Or they can be changed by passing them through a SPISettings class objects, defined as the following syntax:

</> *SPISettings <object name> (<frequency>, <bit order>, <data mode>) ;* </>

- Then pass this object to the beginTransmission() function.
- If </>*beginTransmission()*</> is called, </>*endTransmission()*</> should be called after completing the data exchange.

- For writing data, any of the </>*void write(<data>)*</>, </> *void write16(<data>)*</>, </> *void write32(<data>)* </>, </> *void writeBytes(<data pointer>, <size>)* </>, </> *void writePattern(<data pointer>, <size>, <repeat>)* </> can be used accordingly.
- For transferring data, any of the </>*uint8_t transfer(<data>)* </>, </> *uint16_t transfer16(<data>)* </>, </> *uint32_t transfer32(<data>)* </>, </> *void transfer(<data pointer>, <size>)* </>, </> *void transferBytes(<sending data pointer>, <receiving data pointer>, <size>)* </>, </> *void transferBits(<sending data>, <receiving data pointer>, <number of bits>)* </> can be used accordingly.
- In case of sensors, search through the sensor library's initializing functions and identify which parameters can be changed.
- Most sensor libraries allow for the change of SPI port, SPI class object and GPIO pins used. For changing serial clock frequency, bit order and data mode, check if those arguments can be taken in via any function.

With that, all I have set out to cover within this blog have been covered. If you have any doubts pertaining to the topic, feel free to ask in the comments section no matter how simple it is.

Until next time peeps, stay charged.

//

Blog series- 3.I^2C

Welcome back to the third blog of this blog series, where I explore different peripherals of ESP32. For those of who are interested, I have already covered UART and SPI with ESP32 in previous blogs of the series.

In this blog, I will be talking about the I^2C protocol and how to use it with ESP32. After discussing UART and SPI, I^2C will be no different as you will come to see that it has quite the similarities with both of them.

First, let us start with what I^2C protocol is and how it runs.

3.1 I^2C protocol

Inter-Integrated Circuit serial communication protocol (or I^2C for short) is a synchronous serial communication, similar to SPI. And guys, it is spelled ‘eye-squared-see’ and not ‘eye-two-see’.

Similar to SPI, there is a clock line here which helps in synchronizing the devices. In other words, understand when and how to exchange bits. Subsequently, there is master and slave. Master is the one who commands, thus drives the serial clock while the slave follows.

Yet, similar to UART, the data here is sent in packets. There is a heads up signal and an end signal, along with a few error handling features.

This is due to I^2C using the half-duplex mode of communication. Both SPI and UART used the full duplex mode of communication, where information could be sent and received at the same time.

But in half duplex mode, information can be either sent or received at the given instant. This enables a very nice feature while using I^2C, the redundancy in the number of lines used.

I^2C uses only two lines. One is the serial clock as the communication is synchronous, while the other is the serial data line dedicated to data exchange.

I^2C also allows for multiple masters to control the same network of slaves. All of this enables a simple wiring in

case of using multiple sensors with different processors. Although, the masters can't communicate with each other.

But the main issue here would be speed. Since there is no one main master, the lines are open drained. They are connected to high voltage through a resistor, generally +5V via a 1k resistor.

This limits how fast the state on the line can change from 0 state to 1 state.

I^2C also has master and slave devices. To recap, a slave can only send and receive data if and only if the master commands it so. And the communication can only happen between master and slave.

Unlike SPI, I^2C supports multiple masters and slaves. For a given set of slaves, there can be more than one masters.

Moreover, there are only two lines in I^2C hardware, the SDA (Serial DAta) and SCL (Serial CLock) lines. All the masters and slaves connect to the same two lines, thereby decreasing the amount of wires and space needed.

Since there is no 'main' master per say to drive the lines to their default states, we should open drain them manually.

Just connect them to the positive voltage (VCC or +5V) through a pull-up resistor, typically a 4.7k ohm resistor.

This setup actually causes a drop in speed for I^2C compared to SPI. The reason is a little simple if you know a little of RC circuits (resistor-capacitor).

In synchronous setting, the serial clock is the upper limit for the data transfer rate. The more rapidly you can switch the serial clock, the faster you can send out the bits.

But the wires that carry the signal have some resistance and capacitance. This slightly flattens what should be a vertical increase of voltage in time.

<>

</> [/ image with 2 clocks, an ideal clock pulse and a real clock pulse]

<>

Initially, since the wires' resistance and capacitance are low, the time taken to switch is also low and our clock can switch at very high rates.

When you connect the pull-up resistors to the wires, the resistances of wires jump from noticeable to huge. This causes the time delay in switching to widen, limiting as to how fast we can switch the clock signal.

This is also the reason why you can't have high speed signals over long wires as longer wires have higher resistances and more delay in switching.

Now that the circuit's lesson is done, let us talk about the lines, specifically the SDA line. Unlike in the previous two cases, where we have the two lines each for receiving and transmitting, there is only one line here.

This is because I^2C follows half-duplex method of communication. There is only one line and the devices take turns to send data. While the other device is sending, the other one either waits or reads in. Of course, this further restricts how fast the exchanges could be.

One more reason as to why the data transfer rate is even slower is the I^2C packet structure.

Unlike in SPI, we don't have a pin for each slave here. Instead, each slave has an address that uniquely identifies them. For sensors, this must be written on their board as I2C or IIC address.

<>

</Insert a picture of I^2C connections here, at least 2 slaves and 2 masters>

<>

Note that I^2C is more often written as I2C, but still read as i-squared-c and not i-2-c.

So, to send the data, the master first sends in the 7-bit or 10-bit address of slave, followed by a bit signifying whether to read or write, then an ACK bit (I will explain this

shortly). Next, you send a byte or two followed their corresponding ACK bits.

So, for sending 8 bits of data to a slave with 7-bit address, you need to send out 18 bits. The bit rate isn't affected by the data transfer rate is certainly down due to this.

<>

</> [/ image with i2c data packet structure]

<>

But despite all these speed reductions, we can still send data at 100k bits per second in standard mode and 400k bps (bits per second) in fast mode. These are again just bits, not bytes and certainly not all will be data bits as specified in the packet structure.

The reason I^2C is still a major protocol is because I^2C offers us a simpler hardware setup as there are only 2 lines and more importantly, the ACK bit. ACK or acknowledgement bit is sent by the slave or master responding that they have received the whole byte that should have been sent in.

It maybe the address+ read/write bit or the data byte, but the received device will respond by setting the SDA line to low when it is time to send the ACK bit. This helps us in getting more secure and confirmed transactions.

There is the clock stretching feature. Whenever the slave device is busy and can't respond to the request from

master, it pulls the SCL low. No device can send any data on SDA until SCL is released by the slave.

ESP32 supports only 2 I^2C ports. Unlike the SPI or UART ports, neither I2C_0 nor I2C_1 port are preoccupied. Moreover, only I2C_0 port has default pins set-up.

The default port used is the I2C_0 port. And the default pins for each port are as shown

PORT	**SDA**	**SCL**
I2C_0	GPIO21	GPIO22
I2C_1	-none-	-none-

So, if you are using I2C_1 port, do remember to set the pins.

3.5. Setting up I^2C ports

I^2C has two modes: master and slave mode.

For the master library, we have the Wire library. So, open it up.

If you know the location of library, that is fine. But if you don’t, here is a simple cheat for you. All you need is to know a function of this library and what types of

arguments it takes. A simple google search with the library name should suffice.

For this example, I am taking begin function. At this point, if you are following from UART and SPI, you should know this is a near universal function to initialize the peripheral.

The default I^2C object is Wire.

So, in setup, just type in

<>

</> *Wire.begin("Wrong");*

<>

This is obviously a *wrong* argument. But when you compile it, the compiler should notify some errors in code and from where the error is. There you should be able to find the location for library.

If you open the library, you should find the class named TwoWire. Since I^2C uses only two wires, it is also sometimes called two wire communication.

First is the object declaration. The syntax is similar to the one we have seen in UART and SPI.

<>

</> *TwoWire <object name here> (<port number here>);*

<>

By default, there are already Wire and Wire1 objects defined with ports I2C_0 and I2C_1 respectively. Use them or define new ones to your wish.

If not, connect your slave device to the default ports as shown in pinout. The slave will have an address in between 1 to 127. So, we will begin and end transmission on every address. The address that pings back positive, meaning it started and ended with no problem, is the slave.

<>

</>

```
#include <Wire.h>
void setup(){
    Wire.begin();
    Serial.begin(9600);
    Serial.println("Addresses found:");
    int n_slaves=0;
for (int address=1;address<127;address++){
    Wire.beginTransmission(address);
    int n_found=Wire.endTransmission();
    if (n_found==0){
```

```
            Serial.print(n_slaves++);
            Serial.println(address);
        }
        else if (n_found==4) {
            Serial.print("Unknown error at address");
            Serial.println(address);
        }
}
if (n_slaves==0){
            Serial.println("Did you connect any I2C
            devices?");
}
else {
        Serial.println("Done scanning");
}
}
void loop() {
        ;
}
<>
```

Once you got the address, let us name our I^2C ports. Again, by default, in the Wire library, they are named Wire

and Wire1. Of these, Wire has the default pins but Wire1 has no such luck.

Pay attention here, because the naming format is different**:

<>

</> *TwoWire <name> = TwoWire (<port number>);*

<>

As you might have guessed, the port name is going to be test_i2c and we will connect it to port 1.

<>

</> *TwoWire test_i2c = TwoWire (1);*

<>

****I have looked into the source file of Wire library and this is how they defined the Wire and Wire1. In previous cases of SPI and UART, we have followed the format shown in their library, so here too we will follow the library format. ****

Once naming is done, let us set up the pins. As usual, we have the begin function. The parameters are as follows:

<>

</> *test_i2c.begin(<SDA pin>, <SCL pin>, <clock frequency>);*

<>

Now, the good thing is that all these argument have default parameters. Clock frequency defaults to 100KHz, while the pins only have default if you are accessing the port0.

Since we are using port1, we have to compulsorily pass the pins.

<>

</> *test_i2c.begin(20,18);*

<>

Now, before we send or receive, we need to begin the transmission:

<>

</> *test_i2c.beginTransmission(<slave address>)*;

<>

After this, we can send and receive data.

First, for sending data, we use the write.

<>

</>test_i2c.write(<data>);

<>

Thanks to the library, you can send any data type. Also, this function returns the number of bytes it wrote, just like in SPI.

To read from slave, we can use read(). But before that, we need to request the data from slave.

<>

</> test_i2c.requestFrom(<slave address>,<bytes requested>);

<>

You need to pass two arguments, first is the slave address and next is how many bytes you want to read.

After this, we can use the read function.

<>

</>c=test_i2c.read();

<>

This function returns the byte data it read, so c will contain the data read.

Also, there is the available function, and is exactly the same as in SPI and UART.

As for the slave mode, it seems the Wire library itself only supports Master configuration. So, going any further is out of scope presently.

3.51. Setting up I^2C pins for sensors

As with the SPI, we mostly use I^2C to communicate between the ESP32 and the sensor or other slave peripherals. As such, they too have their libraries and let us see how we can deal with them.

Now, I am going to take the Adafruit BME sensor as an example. So, go to the 2.51 section and see how you can open the bme sensor library file. (This was the reason why I preferred this sensor)

<>

</> [/ picture of bme library instantiates]

<>

Notice that, for I^2C instantiating, you only need to pass only one argument. And even that one argument is optional. Since default isn't mentioned here, let us go to the header file.

Here, you will see that the default is the pointer to Wire, the default I^2C port.

You have two options: let it be and use the default pins on board with your sensor (or) chose the hell mode.

If you chose the hell mode, which means you want to customize the port or/and pins you are using, here is how you do it.

Within the Adafruit_BME680 library, you will see that you can give the custom when you are instantiating the sensor.

For this example, let us say you have an I^2C port named i2c_1, connected to port1 and want the pins to be SDA=12, SCL=14.

So, initialize the i2c_1 and connect to port1. (Do this globally)

<>

</> *TwoWire i2c_1 (1);*

<>

Next, pass it to the bme library. (Globally of course)

<>

</> *Adafruit_BME680 bme (&i2c_1);*

<>

bme is the name used to refer the sensor here. Generally, most sensor libraries won't be foolish enough to pass their default pins against your instructions.

But sometimes all your code and pin connections might be fine but the sensor doesn't respond. In that case, open up the library and probe if there are any default pins passed. If not, the issue isn't here. If there are, see if you can stop them or just overwrite them.

But most libraries aren't that painful. And bme680 library isn't too, I personally checked. Even then if you want to check, fair warning, you will hope through bme680's .cpp file, then the header file and finally find and go through the a different library called "Adafruit_I2CDevice". If you are forgetful, you might have to go to the wire library again. So, good luck!

Now, we have seen that there are no default pins from the library that will override. Thus, let us set the pins.

Additionally, if you want to set the frequency, you can do that also.

In setup(), simply initiate the i2c port.

<>

</> i2c_1.begin(<SDA pin>,<SCL pin>,<optional frequency here>);

<>

That is it, just remember to do this before you use begin on bme. And you can continue as normal.