Week 4 Lab 2: *Communications!*

Introduction

In today's lab, we have three main **goals** that we would like to achieve. The first goal is to solder our **APDS9960** Gesture recognition sensor to the header pins that come with it. The second goal is to learn about how to wire up and then communicate with the **APDS9960**. The third goal is to try out the internet communication abilities of our **ESP32**!

Your lab tutor will come to your table and bring you to the soldering stations to complete **Task 1**. In the meantime, you can try out **Task 2**. We only have a limited number of stations so this may take a little while.

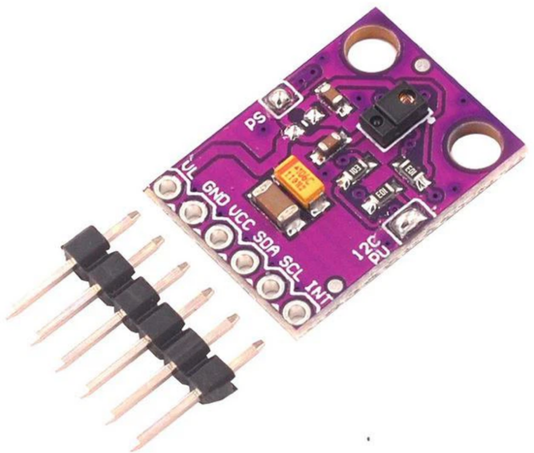
Your tasks for today are the following:

T1. No matter what, we must *solder* on!
T2. Trying out the servo
T3. Using I2C to directly communicate with our APDS9960
T4. Using a library for the APDS9960
T5. Combining our servo and APDS9960
T6. A tiny little webserver
T7. ESP32, phone home

T1. No matter what, we must *solder* on!

You may have noticed the two other little boards in those little anti-static bags that we have provided you. One of the boards is an Inertial Measurement Unit (IMU) for measuring motion, and the other board is the **APDS9960,** a proximity, colour, and gesture detection sensor that will give our ESP32 the ability to detect the world around it.

You may also notice that the boards do not have handy pins soldered to them for plugging into a breadboard. Luckily, the pins have been included in the package and we just have to connect them up.



So, today, you will be doing a little soldering! Don't worry if you have not soldered anything before. I will give a safety briefing at the start of class, and then you will be taken to the soldering stations group by group.

T2. Trying out the servo

In your bag of sensors is a servo motor. A servo motor is a motor that allows you to accurately control the rotation of the motor shaft by using an analog signal (or with PWM). The high gear ratio allows the servo to move even with very low voltage/current making them suitable for use with microcontrollers. We can use servos as actuators in our IoT projects to perform mechanical work.

Our servo is the SG90, which can rotate approximately 180 degrees (90 in each direction). It has three wires, one for power, one for ground, and one control wire. By applying an increasing voltage on the control wire, the motor will turn to the direction we desire. You can plug it directly into the expansion board as no motor driver is required.

Servo Pin	**Purpose**	**ESP32 pin**
GND (Brown)	Ground	GND
VCC (Red)	Power Supply	3.3v
Control (Orange)	PWM servo control input	Pin 13

For today, we are going to use a library **ESP32Servo** (install this using the Ardunio IDE), however I suggest trying out driving the motor yourself using the ESP32 PWM features later. This library has a collection of examples that use the ESP32's timers and PWM blocks to control the servo.

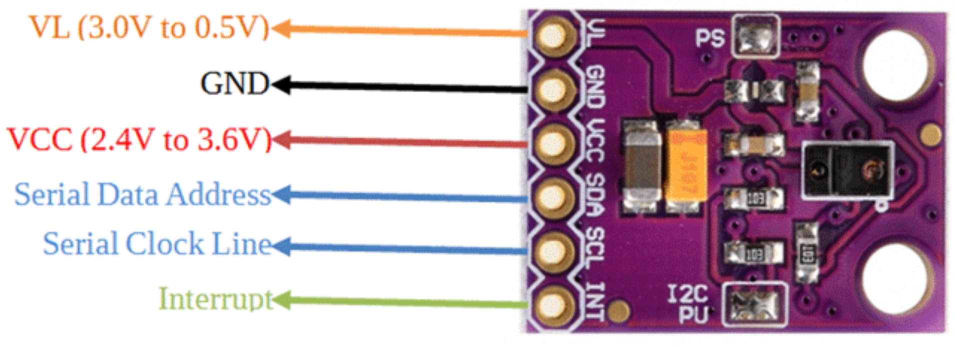
Select Examples > ESP32Servo > PWM Example. Run the program and observe what happens. Note, that the servo moves back and forth. Try modifying the program to speed up or slow down the rotation. Can you get it to change direction or move to the middle?

Lab Report Task 1: Write a program that can turn the servo to the right, to the left and to the middle. Include any relevant source code. (2 marks)

Optional Task: Can you get your servo running without the library?

T3. Using I2C to directly communicate with our APDS9960

Hopefully, by now you have soldered the pins to your APDS9960 board. Now, let's take a good look at our board.



Notice that there is an **SCL** and an **SDA** pin. Does this ring any bells? Correct, this is an I2C device, and we can use I2C to communicate with it. But first let's wire up our device.

APDS9960 Pin	Purpose	ESP32 pin
GND	Ground	GND
VDD	Power Supply	3.3v
SCL	I2C Serial Clock	SCL (Pin 22)
SDA	I2C Serial Data	SDA (Pin 21)
INT	Interrupt pin	Not needed
VL	Optional power	Not needed

We are going to use the library Wire which lets us communicate with I2C devices: https://www.arduino.cc/reference/en/language/functions/communication/wire/

First, wire up the board and check it is responding by using the **I2CScanner.ino** program. If you have wired it correctly you should see:

```
Scanning...
I2C device found at address 0x39!
done
```

The way that we will interact with the APDS9960 is by interacting with the chip's internal registers, namely, we will start by setting up the **ENABLE** register, and then reading from the **RGBC Data** register using I2C. We are going to use the registers on page 19 of the APDS9960 Techsheet to use the colour and ambient light detection.

Use the following program (ColorSensor.ino) and try it out:

```
uint8_t address = <Put the device address here>;
uint8_t MSByte = 0, LSByte = 0;
uint16_t regValue = 0;

#include <Wire.h>

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

  Wire.beginTransmission(address);
  Wire.write(0x80);
  Wire.write(0x03);
  Wire.endTransmission();

  delay(500);
}

void loop() {
    Wire.beginTransmission(address);
    Wire.write(0x94);
    Wire.endTransmission();

    Wire.requestFrom(address, 1);
    if(Wire.available()){
        LSByte = Wire.read();
    }

    Wire.beginTransmission(address);
    Wire.write(0x95);
    Wire.endTransmission();

    Wire.requestFrom(address, 1);
    if(Wire.available()){
        MSByte = Wire.read();
    }

      // What is this result?
      regValue = (MSByte<<8) + LSByte;
      Serial.print("Register Value: ");
      Serial.println(regValue);
      // Wait 1 second before next reading
      delay(1000);
    }
}
```

I have highlighted some important parts of this program. Identify what registers or values the red highlighted numbers are.

Let’s look at the steps of this program:

1. Write to ENABLE register:
   First byte: (7-bit I2C address followed by a low R/W bit)
   Second byte: 0x80 (points to enable register)
   Third byte: 0x03 (the enable register to be written)
2. Write to an Address Pointer register:
   First byte: (first 7-bit I2C address followed by a low R/W bit)
   Second byte: 0x94 (points to the first RGBC Data register)
3. Read RGBC Data register:
   First byte: (7-bit I2C address followed by a high R/W bit)
   Second byte: the APDS960 response
4. Write to an Address Pointer register:
   First byte: (first 7-bit I2C address followed by a low R/W bit)
   Second byte: 0x95 (points to the second RGBC Data register)
5. Read RGBC Data register:
   First byte: (7-bit I2C address followed by a high R/W bit)
   Second byte: the APDS9960 response

Let’s first understand what the ENABLE register does. Take a look at the APDS9960 Techsheet (**Hint:** page 20), and identify the configuration that we have sent in the above register: 0x03

Now let’s understand what the RGBC registers do. Again, let’s look at the APDS9960 Techsheet and identify what values we are reading from the APDS9960.

Note in the program on the previous page, I have written all numbers in hexadecimal notation as is common practice, rather than in binary. Feel free to change it to binary notation if it helps you understand better.

Change this program to read all values of the RGBC Data register and print out the relevant numbers. You might want to use some sort of function abstraction to make your program more readable.

Lab Report Task 2: Explain the meaning of the register that we sent in the first version of the program. What is the value that is being printed out? Explain what are all of the values of the RGBC Data register. Describe or show your changed program. (2 marks)

T4. Using a library for the APDS9960

Now as you can see, all this can be quite tedious to do. It took me quite some time to get the previous example working for your lab. Let's do ourselves a favour and use a library. For a lot of the sensors/actuators you have, libraries already exist which you are free to use in your projects. Though you should reference them in any report you write up.

We are going to use an APDS9960 library called SparkFun APDS9960 Library. You know the drill by now: **Tools > Manage Libraries > Search: SparkFun APDS9960 > Install**

Important: Before we use this library, we have to make a small change to it. Our APDS9960 has a different chip on board and as a result the library does not out of the box.

We need to find the file called: **SparkFun_APDS9960.h**
This is the header file for the library.

On Mac OS X, it can be found within
~/Documents/Arduino/libraries/SparkFun_APDS9960_RGB_and_Gesture_Sensor/src

On Mac OS X, it can be found within
My Documents/Arduino/libraries/SparkFun_APDS9960_RGB_and_Gesture_Sensor/src

Open SparkFun_APDS9960.h and change line 34 to read:
#define APDS9960_ID_1   0xA8

Ok now, let's open the example called ColorSensor:
File > Examples > SparkFun APDS9960… > ColorSensor
Have a look through it and try running it on your ESP32. Much easier no?
Indeed, it gives you the results of all four colours.

The APDS9960 has many other features. Let's try its gesture detection abilities.
Open the provided Arduino sketch **GestureTest.ino** and try it out on your ESP32.
By moving your hand in front of the sensor you should see a print out in the serial describing the motion you have done. Neat! Ok let's put it to good use.

T5. Combining our servo and APDS9960

Let's try now combining our sensor and actuator into a closed-loop program. When you move your hand from left-to-right, the servo motor should move to the left. When you move your hand right-to-left, the servo motor should move to the right. You can decide what to do for Up, Down, Far or Near states of the APDS9960.

Lab Report Task 3: Describe your changes briefly in your lab report. Include your source code in your report. (3 marks)

Optional Task: Can you use the interrupt feature for gestures on the APDS9960?

T6. A tiny little webserver

Now for a change of pace. Let's try out the HTTP functionalities of our little ESP32. Open the WebServer.ino file and read through and try and see how it works. For this task, you will need to use your Smartphones mobile hotspot. You will see the SSID and password variables right at the start of the program, and these should be your Smartphone credentials. Edit these and try and get your ESP32 to connect to your Smartphone. While it is trying to get connected, it will print ".". If it connects it should look something like this:

```
Connected to Note10+AP
IP address: 192.168.37.15
MDNS responder started
HTTP server started
```

You now need to get your laptop to connect to the same network (your smartphone). Once connected you can try opening a web browser and connecting to the ESP32's IP address. In the above example, it is 192.168.37.15. You should see a "ESP32 on the Web!" message. Now try to go to another endpoint or route, such as "inline", by pointing your browser to 192.168.37.15/inline (with your ESP32's IP address).

Let's try creating new end point (toggleLED). This endpoint should toggle an LED on and off (you can use an external LED or the built in one). It should also send back the LED status ("LED On" or "LED Off"). That is, when you go to <ip-address>/toggleLED, the ESP32 should:

1. Toggle an LED
2. Report the LED status back to the web browser

So now we have seen some idea of how to affect actuation over the internet, but this is rather primitive. We will come back to this in our backend lab later in the semester.

Lab Report Task 4: Describe your changes briefly in your lab report. If you defined any other function or made source code changes please include it in your report. (2 marks)

T7. ESP32, phone home

Ok one last task for this lab! Let's try sending information to a simple server running on your computer from the ESP32 device. Again, your laptop and your ESP32 device both need to be connected to your smartphone hotspot. For this task we are going to use a simple python HTTP framework/server called **flask**. You will need to install **python3** on your laptop and then use pip to install flask (`pip install flask`). If you are having trouble, ask!

Have a look at the small_webserver.py script. You can see that it defines routes/endpoints in a similar way to the webserver in Task 5. Here we only have one endpoint, called "data". You can also see that it is expecting a payload (data1 and data2). We are going to use a HTTP GET request to send a payload from the ESP32 device to your laptop server.

One you have python and flask installed you can run the script:

```
python small_webserver.py
```

You should see something like:

```
* Serving Flask app 'small_webserver' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses (0.0.0.0)
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://127.0.0.1:3237
 * Running on http://192.168.37.119:3237 (Press CTRL+C to quit)
```

Ok great. Now we need to get your ESP32 device to use an HTTP Client to connect to our webserver. To do this open the Http_Request.ino Arduino sketch. Again, you will have to specify the SSID and password for your hotspot, and now you will also have to include your laptops IP address. Try this program out. If successful, you should see data being printed out like so every 5 seconds:

```
Data from ESP32:  31.2 76.0
```

But where does this data come from? Well let's look into the source of Http_Request.ino.

You can see the example HTTP GET payload in this line:

```
"data?data1=31.2&data2=76"
```

This payload is defined as the following:

<name of the route>**?**<var_name>**=**<value>&<var_name>**=**<value>&..

Here we are sending two values 31.2 and 76. However, they are currently hardcoded.
Pick a sensor and send its data back to your laptop using the HTTP Client. Which sensor is up to you!

Lab Report Task 5: Describe your changes briefly in your lab report, which sensor you used, and describe how you send the data back to the laptop. (2 marks).

Now, we have only looked at a very simple way to send data from your ESP32 to the Cloud (or rather your laptop). We will cover more on this in a later part of the module!

Appendix 1: I2C Scanner

```
#include <Wire.h>

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

void loop()
{
  byte error, address;
  int nDevices;
  Serial.println("Scanning...");

  nDevices = 0;
  for(address = 1; address < 127; 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++;
    }
  }
  if (nDevices == 0)
    Serial.println("No I2C devices found\n");
  else
    Serial.println("done\n");

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