Final Project – Web Controlled Solar Tracker with Automation and Feedback Capabilities

EEE 174 / CpE 185 Lab
Section (02)

Instructors:
Eric Telles
Sean Kennedy

Tuesday 6:30-9:10pm

Edgar Granados
Richard Castro Jr.

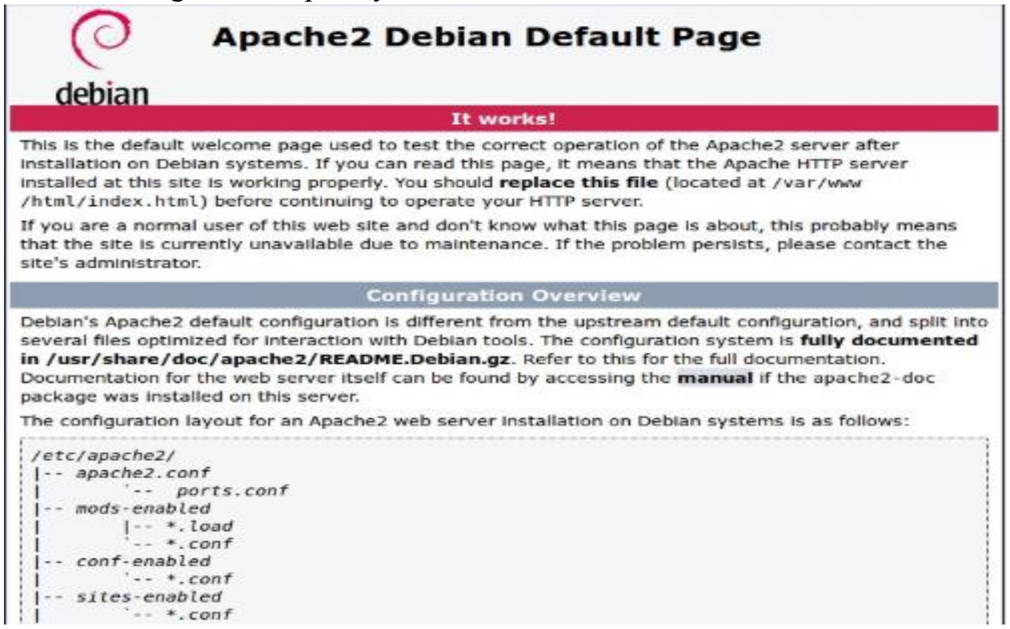
Final Project Overview

The prime emphasis for this project was to design and create a dual axis
solar tracker that automatically follows the movement of the sun. It would determine the most suitable position that yields the highest amount of solar energy. In addition, we utilized the Raspberry Pi, in order to, host a web server that would be responsible for: initiating the automatic tracking system, sending/receiving information about the current solar tracker, and displaying readings onto a simple web page. In addition, we utilized the small microcontroller, STM32F303K8 Nucleo in order to convert the analog signals coming out of the sensors into digital values, send those over to the Raspberry Pi via UART Communication, and control the servo motors.

Part 1: Setting Up The Web Page Interface

Step 1.1: Downloading Apache

Ideally, the way we pictured our design was, we would be utilizing many different components in our system that would need to be able to communicate with one another. For this reason, our Raspberry Pi, would be utilized as the master component of the system, and the microcontroller as the second master component. The Raspberry Pi was responsible for hosting a web server and sending/receiving from the microcontroller. In order to have a proper Web server interface, we needed to download Apache Web Server onto the Raspberry Pi. Apache Web Server can serve HTML files over HTTP, and with additional modules can serve dynamic web pages using scripting languages such as PHP. By default, Apache puts a test HTML file in the web folder. This default web page, can confirm whether you have installed apache correctly onto your Raspberry Pi. Figure 1 shows the image of the default web page that can be accessed through the Raspberry Pi's IP Address.


Apache2 Debian Default Page

debian

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Debian systems. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at /var/www/html/index.html) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

Configuration Overview

Debian's Apache2 default configuration is different from the upstream default configuration, and split into several files optimized for interaction with Debian tools. The configuration system is **fully documented in /usr/share/doc/apache2/README.Debian.gz**. Refer to this for the full documentation. Documentation for the web server itself can be found by accessing the **manual** if the apache2-doc package was installed on this server.

The configuration layout for an Apache2 web server installation on Debian systems is as follows:

```
/etc/apache2/
|-- apache2.conf
|       `--  ports.conf
|-- mods-enabled
|       |-- *.load
|       `-- *.conf
|-- conf-enabled
|       `-- *.conf
|-- sites-enabled
|       `-- *.conf
```

Figure 1: Default WebPage

Step 1.2: Downloading PHP

After Apache has been successfully been installed, we can now install PHP onto the Raspberry Pi. Installing PHP, is necessary if you want to allow your Apache server to process PHP files; the latest version of PHP will need to be installed and the PHP module for Apache. Figure 2 demonstrates the commands needed to install PHP.

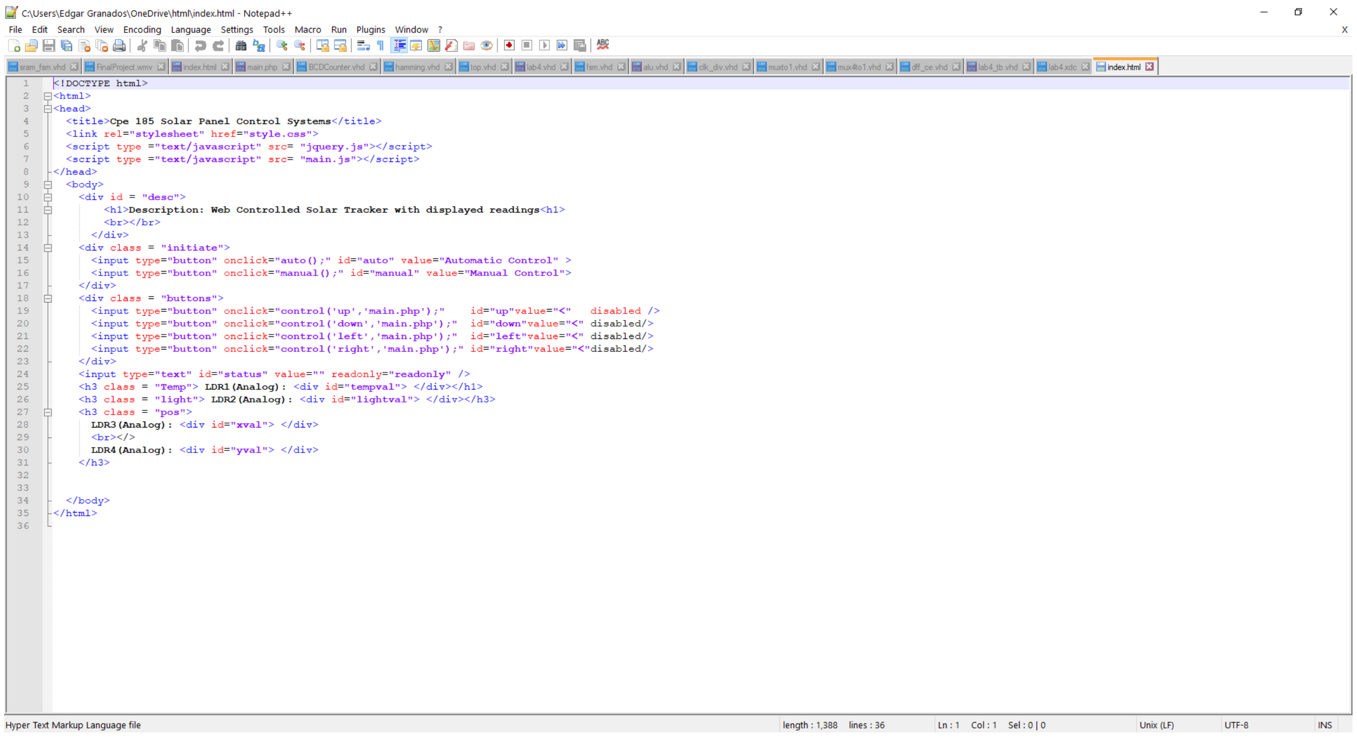
```
sudo apt-get install php libapache2-mod-php -y
```

Figure 2: Command needed to Install PHP

After installation has completed we are now able to effectively process PHP scripts through Apache. In addition, our Web Pages will have the potential to run more dynamically; and behave in such a manor like a high-level language program.

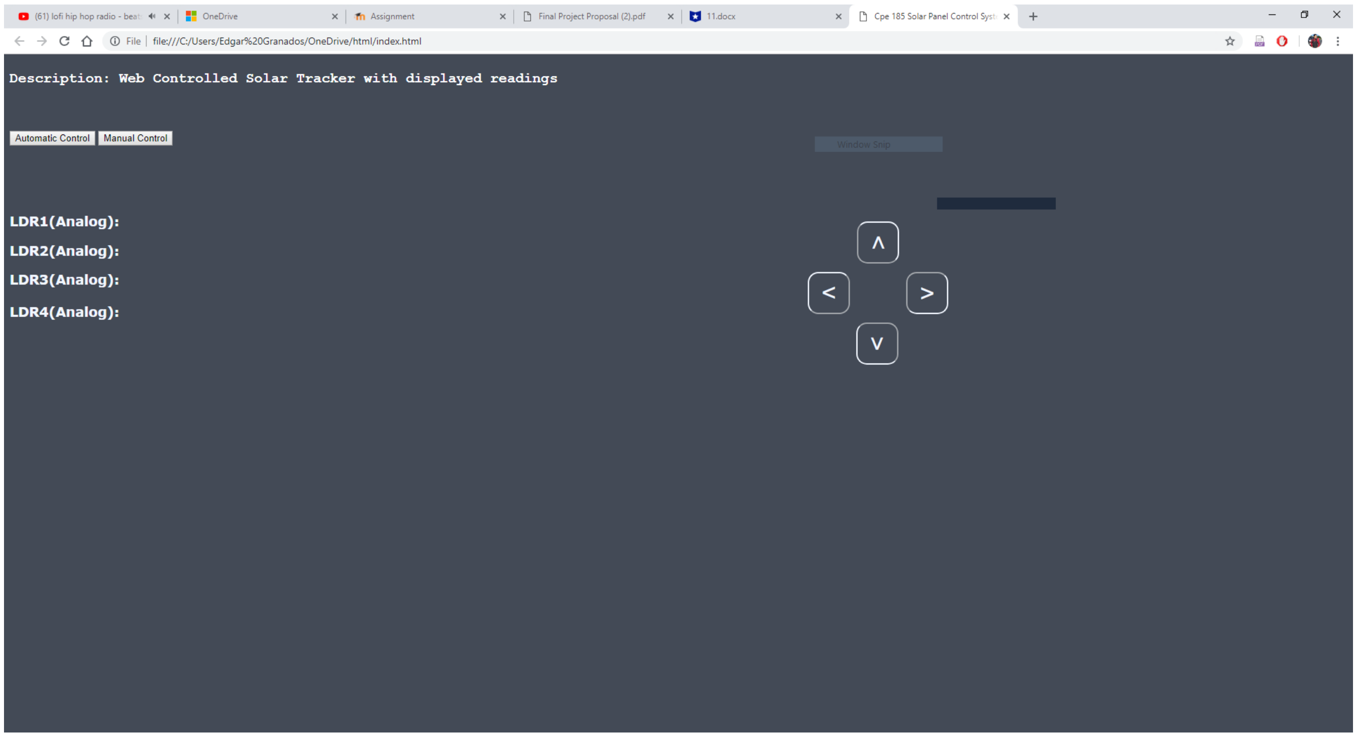
Step 1.3: Created the Main Webpage Interface

To create our user interface, we had to utilize several different scripting languages in order to have a structurally sound and alethically pleasing web page. The first step write out the basic structure of how the page would look like. To do this, we wrote an HTML file that contained all the main components of our site. The main components included, two buttons; one labeled "Automatic Mode" and the other "Manual Mode". These names are pretty self-explanatory; the other components include a small description header that provided the user with a brief description of what the web page was supposed to do. Then under the description header, was a list of elements each designed to demonstrate the reading of an individual LDR sensor. Lastly, to the right of the page, was a graphical design of a common D-pad, that would be used to manually control the Solar Tracker. Figure 2 demonstrates the code that was written within the HTML code and Figure 3 demonstrates the actual Web Page and its contents; after it was loaded onto Google Chrome.


C:\Users\Edgar Granados\OneDrive\html\index.html - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
sram_fsm.vhd
FinalProject.wmv
index.html
main.php
BCDCounter.vhd
hamming.vhd
top.vhd
lab4.vhd
fsm.vhd
alu.vhd
clk_div.vhd
muxto1.vhd
mux4to1.vhd
dff_ce.vhd
lab4_tb.vhd
lab4.xdc
index.html
Hyper Text Markup Language file
length : 1,388 lines : 36
Ln : 1 Col : 1 Sel : 0 | 0
Unix (LF)
UTF-8
INS

```html
<!DOCTYPE html>
<html>
<head>
  <title>Cpe 185 Solar Panel Control Systems</title>
  <link rel="stylesheet" href="style.css">
  <script type ="text/javascript" src= "jquery.js"></script>
  <script type ="text/javascript" src= "main.js"></script>
</head>
  <body>
    <div id = "desc">
        <h1>Description: Web Controlled Solar Tracker with displayed readings<h1>
        <br></br>
      </div>
    <div class = "initiate">
      <input type="button" onclick="auto();" id="auto" value="Automatic Control" >
      <input type="button" onclick="manual();" id="manual" value="Manual Control">
    </div>
    <div class = "buttons">
      <input type="button" onclick="control('up','main.php');"    id="up"value="<"   disabled />
      <input type="button" onclick="control('down','main.php');"  id="down"value="<" disabled/>
      <input type="button" onclick="control('left','main.php');"  id="left"value="<" disabled/>
      <input type="button" onclick="control('right','main.php');" id="right"value="<"disabled/>
    </div>
    <input type="text" id="status" value="" readonly="readonly" />
    <h3 class = "Temp"> LDR1(Analog): <div id="tempval"> </div></h1>
    <h3 class = "light"> LDR2(Analog): <div id="lightval"> </div></h3>
    <h3 class = "pos">
      LDR3(Analog): <div id="xval"> </div>
      <br></>
      LDR4(Analog): <div id="yval"> </div>
    </h3>


  </body>
</html>

```

Figure 3: HTML Code used to structure Webpage

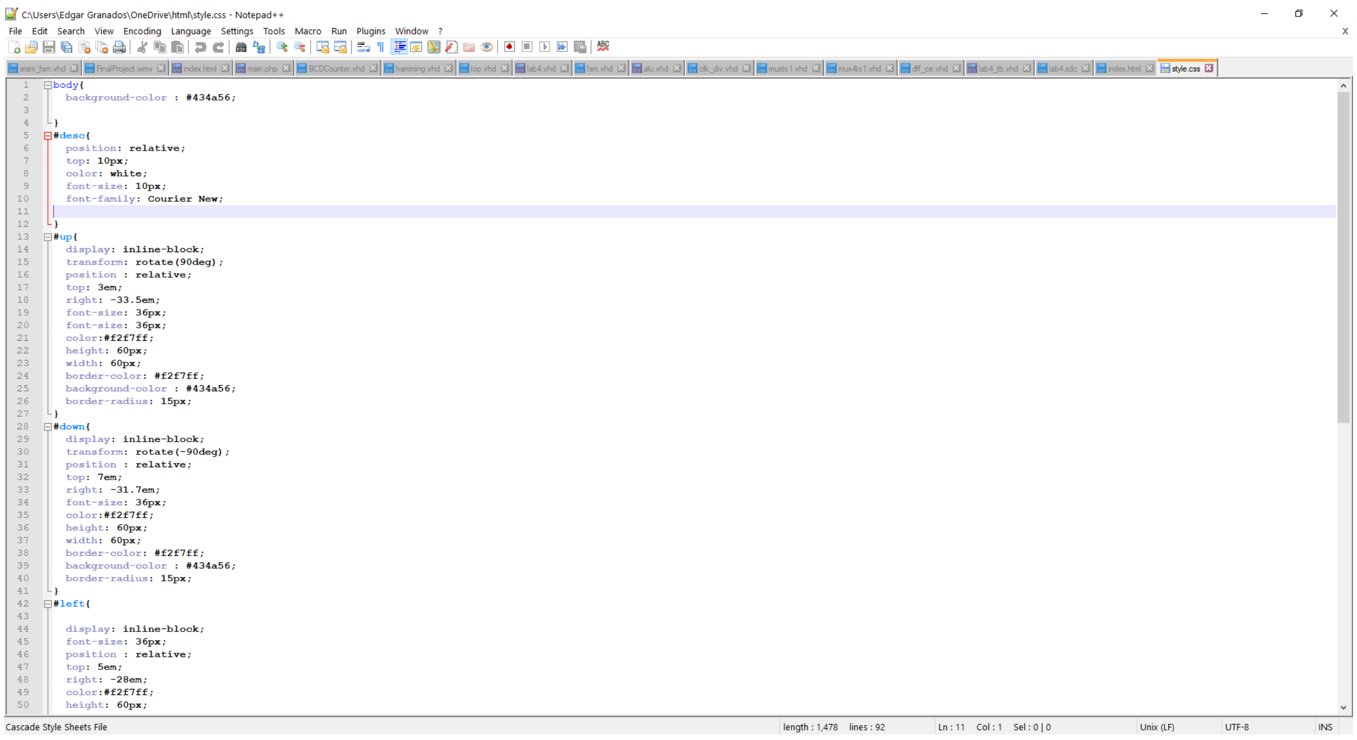

Description: Web Controlled Solar Tracker with displayed readings

LDR1(Analog):
LDR2(Analog):
LDR3(Analog):
LDR4(Analog):

Figure 4: Web Page Interface after loaded onto Google Chrome

Step 1.3: Styling the webpage and linking JavaScript and PHP scripts

Another useful scripting language is known as CSS, which is used to style the content of a webpage. In our case, we used CSS in order to organize the position of where each component of the site was going to reside. In addition, we also changed the "look" and color of certain elements to give them a more appealing look. We could have just had a simple web page with no background color or special images. However, we wanted the page to look like a main hub for a user, and make it user friendly, so that the majority of people would be able to use it. Figure 3 shows the look of the page after styling it and Figure 4 demonstrates the CSS code used to style the site.



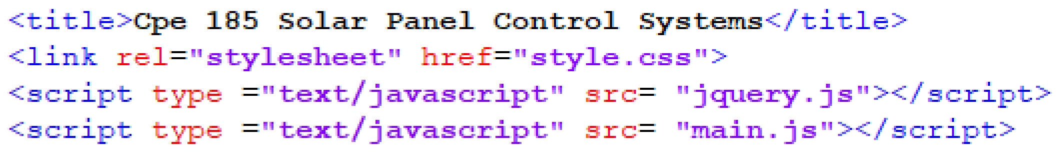
```css
body{
  background-color : #434a56;

}
#desc{
  position: relative;
  top: 10px;
  color: white;
  font-size: 10px;
  font-family: Courier New;

}
#up{
  display: inline-block;
  transform: rotate(90deg);
  position : relative;
  top: 3em;
  right: -33.5em;
  font-size: 36px;
  font-size: 36px;
  color:#f2f7ff;
  height: 60px;
  width: 60px;
  border-color: #f2f7ff;
  background-color : #434a56;
  border-radius: 15px;
}
#down{
  display: inline-block;
  transform: rotate(-90deg);
  position : relative;
  top: 7em;
  right: -31.7em;
  font-size: 36px;
  color:#f2f7ff;
  height: 60px;
  width: 60px;
  border-color: #f2f7ff;
  background-color : #434a56;
  border-radius: 15px;
}
#left{

  display: inline-block;
  font-size: 36px;
  position : relative;
  top: 5em;
  right: -28em;
  color:#f2f7ff;
  height: 60px;
```

Cascade Style Sheets File
length : 1,478 lines : 92
Ln : 11 Col : 1 Sel : 0 | 0
Unix (LF)
UTF-8
INS

Figure 5: CSS code used to style the Website

In order to have a dynamically running Web page, we also needed to use link together a JavaScript file and PHP. JavaScript is a programming language for the web. It is supported by most web browsers including Chrome, Firefox, Safari, internet Explorer, Edge, Opera, etc. Most mobile browsers for smart phones support JavaScript too. It is primarily used to enhance web pages to provide for a more user-friendly experience. These include dynamically updating web pages, user interface enhancements such as menus and dialog boxes, animations, 2D and 3D graphics, interactive maps, video players, and more. It is often referred to as a client-side programming language. On the other hand, PHP is a server-side scripting language designed for Web development, and also used as a general-purpose programming language.

We needed to make sure that JavaScript and PHP were linked to our HTML so that Apache could successfully process the scripts. Therefore, to do this, we went ahead and wrote the following lines of code as shown in Figure 5.



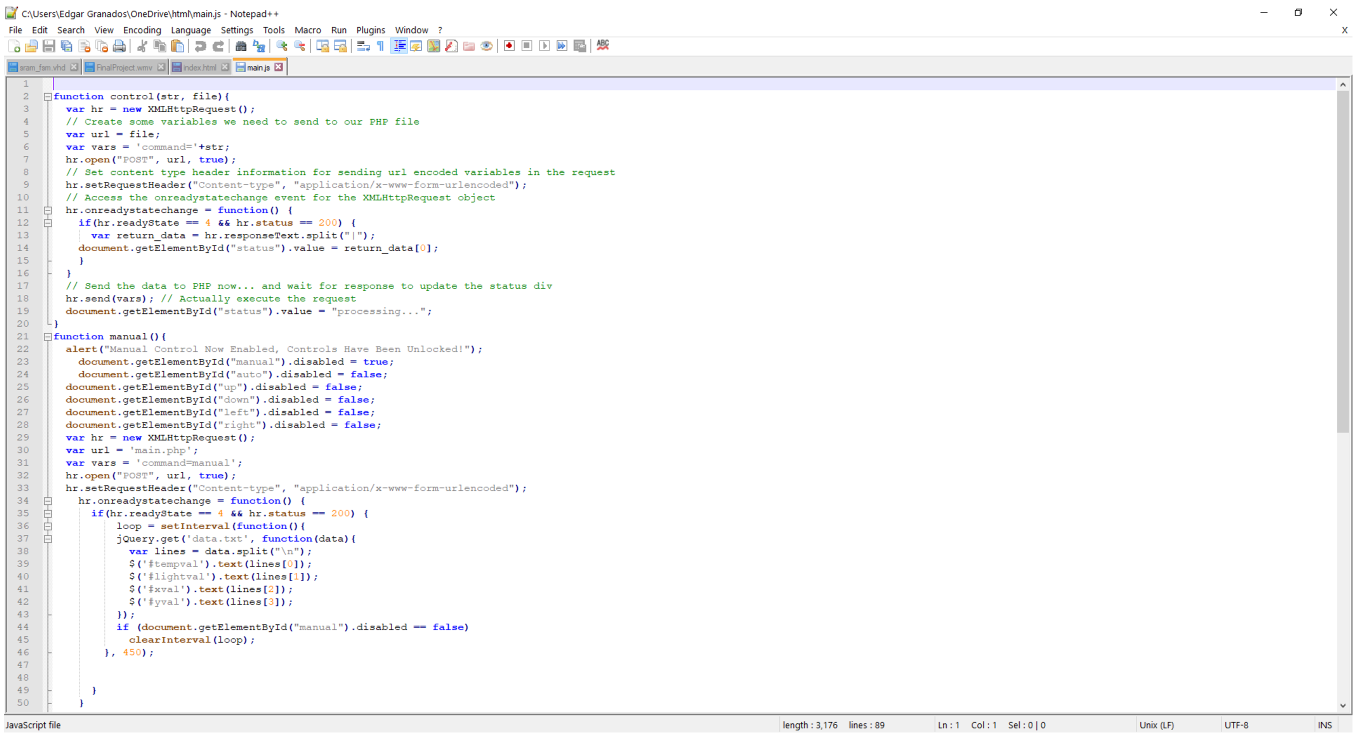
```
<title>Cpe 185 Solar Panel Control Systems</title>
<link rel="stylesheet" href="style.css">
<script type ="text/javascript" src= "jquery.js"></script>
<script type ="text/javascript" src= "main.js"></script>
```

Figure 6: Linking our JavaScript and PHP script to the HTML file

Step 1.4: AJAX Calls

Now that we had all our scripts linked together, our web page now had the potential to dynamically perform tasks. AJAX is a set of Web development techniques using many web technologies on the client side to create asynchronous Web applications. With Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. Ajax allows web pages, and by extension web applications, to change content dynamically without the need to reload the entire page.

This was a vital technique we needed to implement onto our website design; because, we were going to have sensor readings that would continuously need to be updated to its current value. With AJAX calls, we could successfully accomplish this. To perform AJAX calls, we need to have our JavaScript file and PHP file in the same directory. Then from our JavaScritpt file we would simply call the onStateReady() function to initiate a connection between JS and PHP. From here, we could easily send and receive data dynamically. Figure 6 demonstrates the code needed to run AJAX calls between JavaScript and PHP.



```javascript

function control(str, file){
  var hr = new XMLHttpRequest();
  // Create some variables we need to send to our PHP file
  var url = file;
  var vars = 'command='+str;
  hr.open("POST", url, true);
  // Set content type header information for sending url encoded variables in the request
  hr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
  // Access the onreadystatechange event for the XMLHttpRequest object
  hr.onreadystatechange = function() {
    if(hr.readyState == 4 && hr.status == 200) {
      var return_data = hr.responseText.split("|");
    document.getElementById("status").value = return_data[0];
    }
  }
  // Send the data to PHP now... and wait for response to update the status div
  hr.send(vars); // Actually execute the request
  document.getElementById("status").value = "processing...";
}
function manual(){
  alert("Manual Control Now Enabled, Controls Have Been Unlocked!");
    document.getElementById("manual").disabled = true;
    document.getElementById("auto").disabled = false;
  document.getElementById("up").disabled = false;
  document.getElementById("down").disabled = false;
  document.getElementById("left").disabled = false;
  document.getElementById("right").disabled = false;
  var hr = new XMLHttpRequest();
  var url = 'main.php';
  var vars = 'command=manual';
  hr.open("POST", url, true);
  hr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    hr.onreadystatechange = function() {
      if(hr.readyState == 4 && hr.status == 200) {
          loop = setInterval(function(){
          jQuery.get('data.txt', function(data){
            var lines = data.split("\n");
            $('#tempval').text(lines[0]);
            $('#lightval').text(lines[1]);
            $('#xval').text(lines[2]);
            $('#yval').text(lines[3]);
          });
          if (document.getElementById("manual").disabled == false)
            clearInterval(loop);
        }, 450);


      }
    }
```

Figure 7: Code needed to establish a connection between JavaScript and PHP (AJAX)

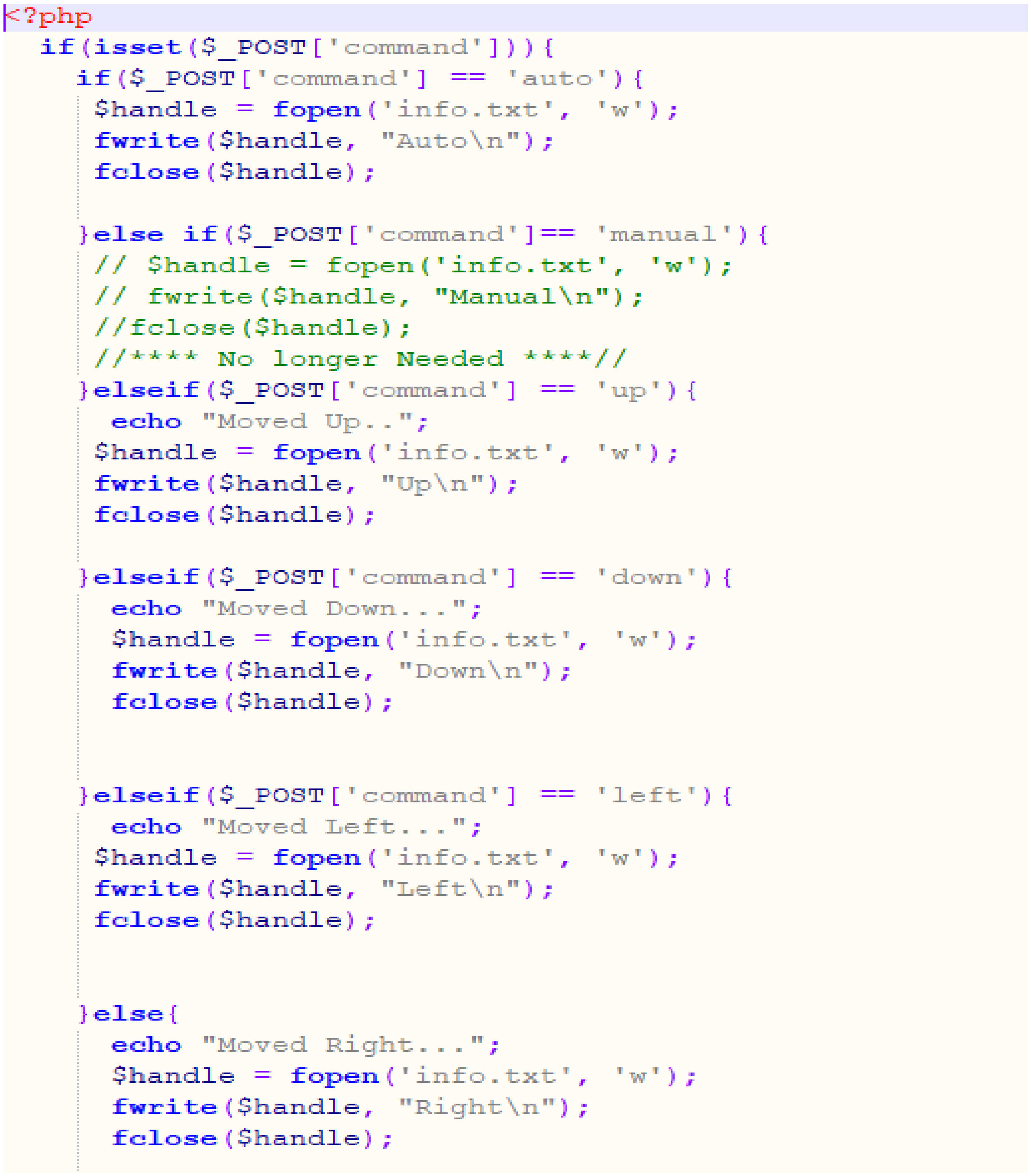
The code written on the main.js JavaScript file, asynchronously updates the content of the LDR values within the Web Page every 500 ms. To do this, we opened a text document named "data.txt." This text file contained all the readings from the LDRs that would be already have the values written by another python program on the Raspberry Pi. JavaScript would simple open the file, read the data, and write this data onto the Web page.

Part II: UART Communication

We needed our Raspberry Pi and STM32 board to communicate with one another, to send out and receive information. We had initially thought of using I2C to accomplish this task but after some careful research we found out the UART was a far more suitable option. With UART, we could send and receive data asynchronously between each device, without having to compromise too much transfer speed.

Step 2.1: PHP script writing to text document

The first step of this process was to have our PHP script write to a text document, commands that would be read by a python script; and then sent to the Nucleo board. To do this. all we really needed was three lines of code to: create or open a file, write a command onto that file, and close the file. The following figure shows the code written to write a certain command onto the text file "info.txt."



```
<?php
  if(isset($_POST['command'])){
    if($_POST['command'] == 'auto'){
     $handle = fopen('info.txt', 'w');
     fwrite($handle, "Auto\n");
     fclose($handle);

    }else if($_POST['command']== 'manual'){
     // $handle = fopen('info.txt', 'w');
     // fwrite($handle, "Manual\n");
     //fclose($handle);
     //**** No longer Needed ****//
    }elseif($_POST['command'] == 'up'){
      echo "Moved Up..";
     $handle = fopen('info.txt', 'w');
     fwrite($handle, "Up\n");
     fclose($handle);

    }elseif($_POST['command'] == 'down'){
      echo "Moved Down...";
      $handle = fopen('info.txt', 'w');
      fwrite($handle, "Down\n");
      fclose($handle);


    }elseif($_POST['command'] == 'left'){
      echo "Moved Left...";
     $handle = fopen('info.txt', 'w');
     fwrite($handle, "Left\n");
     fclose($handle);


    }else{
      echo "Moved Right...";
      $handle = fopen('info.txt', 'w');
      fwrite($handle, "Right\n");
      fclose($handle);
```

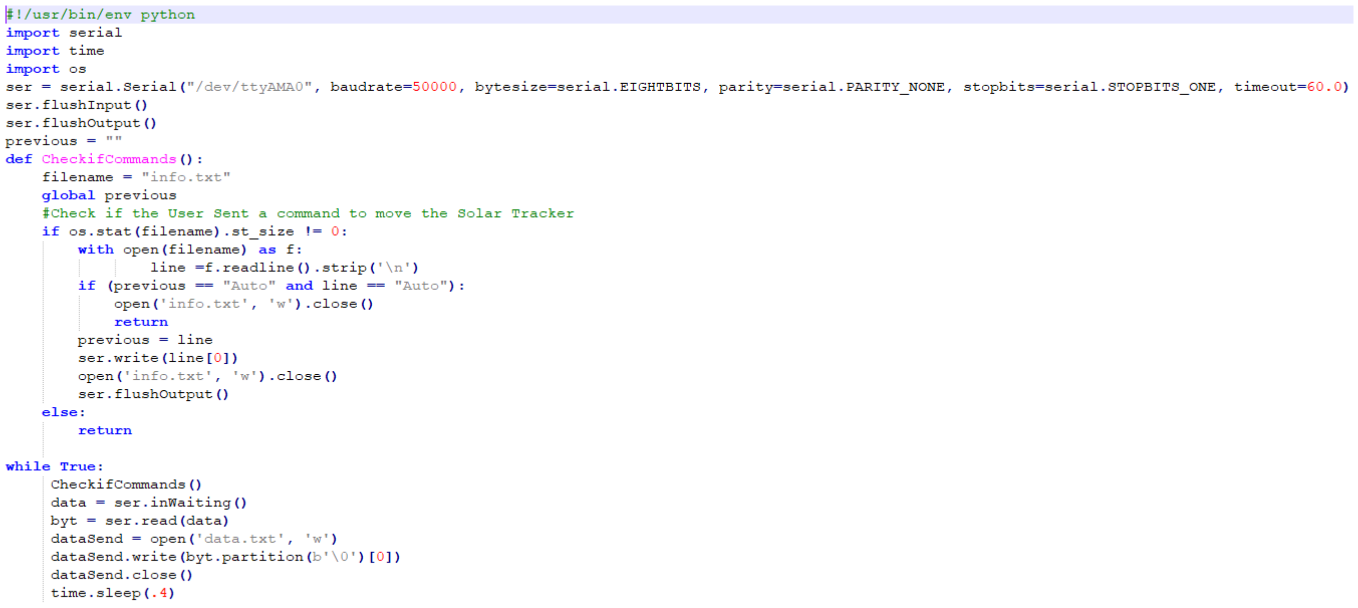
Figure 8: PHP code to write a string onto a text document

Based on the button we clicked on the web page, we needed to make sure that button corresponded to a certain command. That is the reason the script is written in such a way, that tests each condition.

Step 2.2: Creating Python Program that will establish UART with STM32 Nucleo

We are now veering toward the second main portion of the project; which was establishing communication with the microcontroller. Once we have communication established, all the microcontroller needed to do was read the data, and based on the data, determine what action to perform. Thus, we first needed to download a python library known as pyserial onto our Raspberry Pi. Pyserial is essential to establish serial communication with our microcontroller.

Now, we created a python script named GetData.Py. This script was responsible for reading out the commands written onto the info.txt file, and then sending those commands to the STM32 nucleo. Moreover, the python script would also receive the LDR data from the Nucleo and write the values onto the data.txt file. The python script would run every 400 ms, constantly checking and receiving information from the Nucleo board. Figure 8 shows the code that was written to the file GetData.py do this.

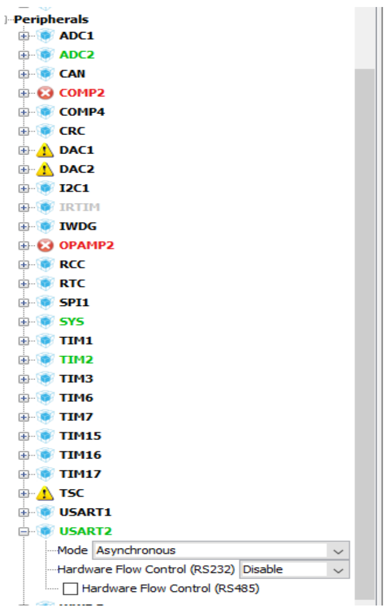


```python
#!/usr/bin/env python
import serial
import time
import os
ser = serial.Serial("/dev/ttyAMA0", baudrate=50000, bytesize=serial.EIGHTBITS, parity=serial.PARITY_NONE, stopbits=serial.STOPBITS_ONE, timeout=60.0)
ser.flushInput()
ser.flushOutput()
previous = ""
def CheckifCommands():
    filename = "info.txt"
    global previous
    #Check if the User Sent a command to move the Solar Tracker
    if os.stat(filename).st_size != 0:
        with open(filename) as f:
                line =f.readline().strip('\n')
        if (previous == "Auto" and line == "Auto"):
            open('info.txt', 'w').close()
            return
        previous = line
        ser.write(line[0])
        open('info.txt', 'w').close()
        ser.flushOutput()
    else:
        return

while True:
    CheckifCommands()
    data = ser.inWaiting()
    byt = ser.read(data)
    dataSend = open('data.txt', 'w')
    dataSend.write(byt.partition(b'\0')[0])
    dataSend.close()
    time.sleep(.4)
```

Figure 9: Python Scripts that sends and receives data

Step 2.3: Setting Up UART Communication on the STM32 board

The first step was to open up the application, STM32CubeMX and select the proper board, STM32F303K8. From there, we created a new project and on the left-hand side expanded the UART2 option. On the very first tab, we selected the Asynchronous option. The last step was to edit the parameters of the UART pins; which was located under the Configuration tab. Figure 9 and 10 demonstrate these steps taken.



Peripherals
ADC1
ADC2
CAN
COMP2
COMP4
CRC
DAC1
DAC2
I2C1
IRTIM
IWDG
OPAMP2
RCC
RTC
SPI1
SYS
TIM1
TIM2
TIM3
TIM6
TIM7
TIM15
TIM16
TIM17
TSC
USART1
USART2
Mode: Asynchronous
Hardware Flow Control (RS232): Disable
☐ Hardware Flow Control (RS485)

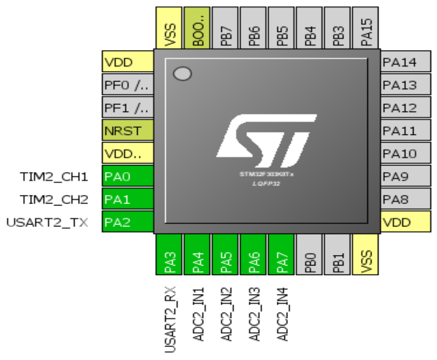
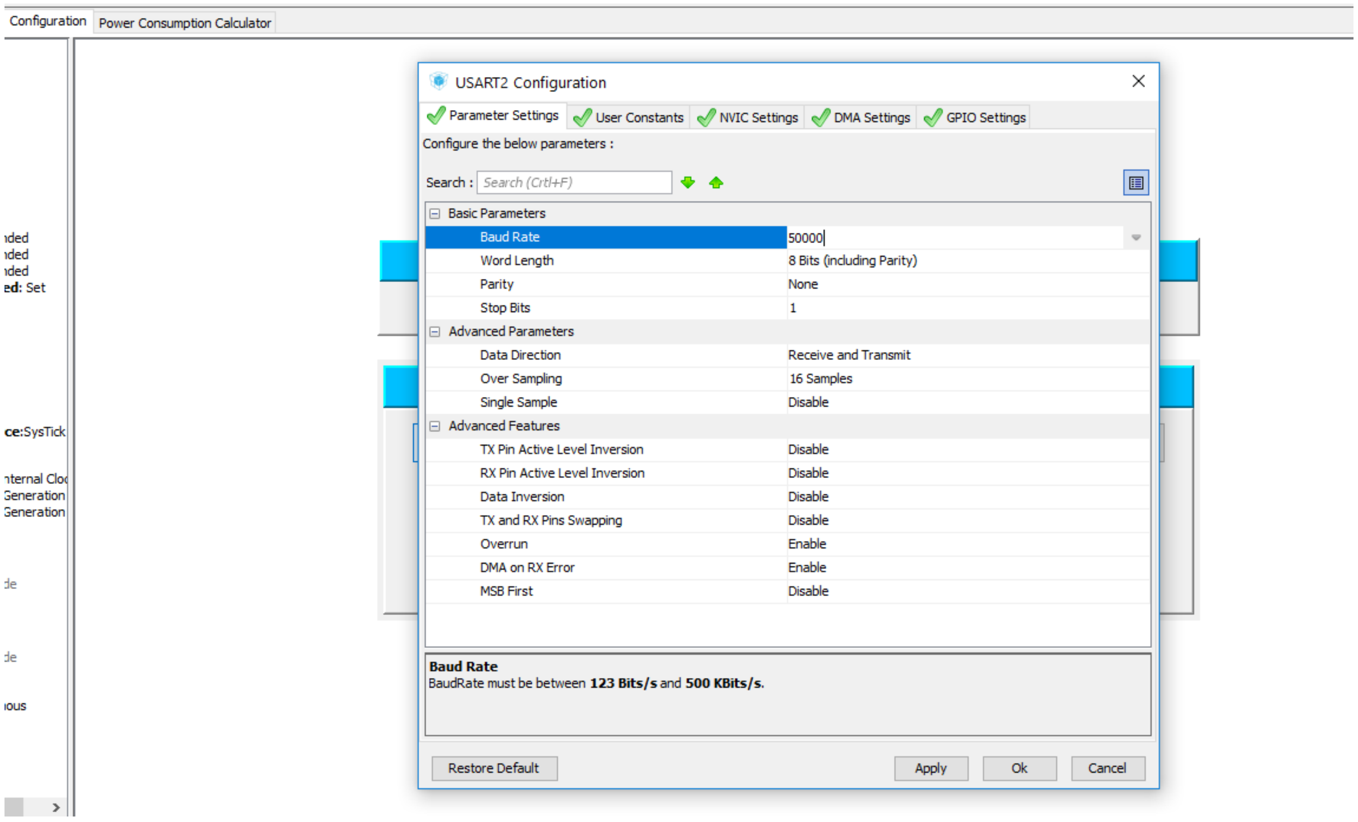


Figure 10: Asynchronous Option chosen on UART2



Basic Parameters	
Baud Rate	50000
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable
Advanced Features	
TX Pin Active Level Inversion	Disable
RX Pin Active Level Inversion	Disable
Data Inversion	Disable
TX and RX Pins Swapping	Disable
Overrun	Enable
DMA on RX Error	Enable
MSB First	Disable

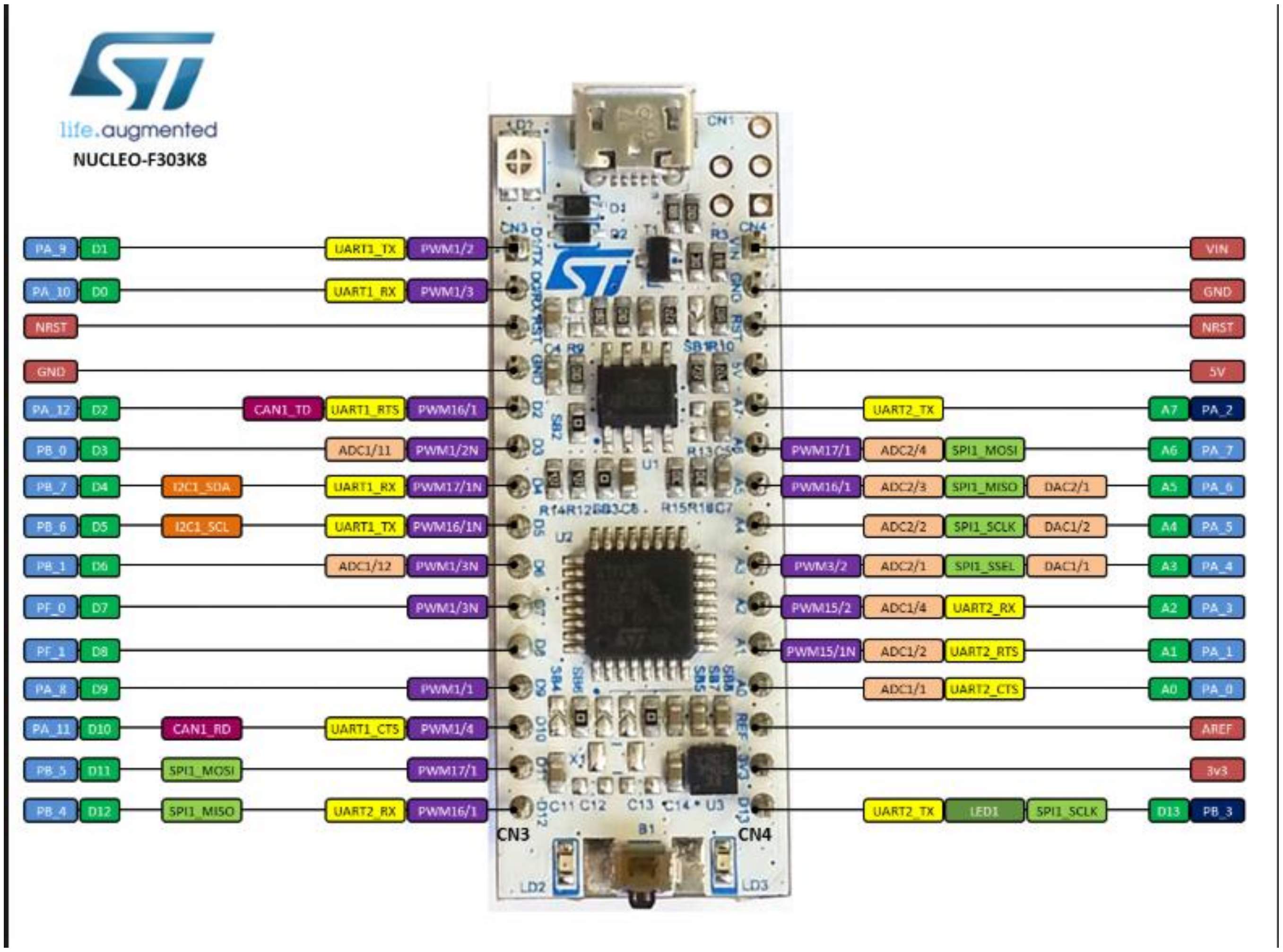
Baud Rate
BaudRate must be between **123 Bits/s** and **500 KBits/s**.

Figure 11: Editing the Parameters Settings on UART2

It is very important that both the python script and STM32 Nucleo program are running the same baud rate. Otherwise, incorrect data may be received. The most stable baud rate , that we found for our set up was 50000. A baud rate slower than this would not transmit the data to the Raspberry Pi fast enough. And a baud rate faster than this, would change the commands sent to the STM32 Nucleo far to quickly.

Step 2.4: Wiring the Raspberry Pi and STM32 Nucleo together.

Now that the settings were properly set for the Pi and Nucleo; it was now time to wire them toward. The Raspberry Pi contains two dedicated pins that are used to establish UART communication. If we look at Figure 11, these are the 4th and 5th pins starting from the top on the second row. We need to wire up these pins in a “criss cross” manner. Meaning that the Tx pin of the Raspberry Pi will connect to the Rx Pin of the STM32 Nucleo (Pin A2) and the Rx pin of the Raspberry Pi will connect to the Tx pin of the Nucleo board (Pin A7). The UART pins for the Nucleo were chosen back in Figure 3. These were the most suitable pins we could use. Figure 11 and 12 demonstrate the pinout diagram for both the STM32 Nucleo board and Raspberry Pi.



life.augmented
NUCLEO-F303K8
PA_9 D1 UART1_TX PWM1/2
PA_10 D0 UART1_RX PWM1/3
NRST
GND
PA_12 D2 CAN1_TD UART1_RTS PWM16/1
PB_0 D3 ADC1/11 PWM1/2N
PB_7 D4 I2C1_SDA UART1_RX PWM17/1N
PB_6 D5 I2C1_SCL UART1_TX PWM16/1N
PB_1 D6 ADC1/12 PWM1/3N
PF_0 D7 PWM1/3N
PF_1 D8
PA_8 D9 PWM1/1
PA_11 D10 CAN1_RD UART1_CTS PWM1/4
PB_5 D11 SPI1_MOSI PWM17/1
PB_4 D12 SPI1_MISO UART2_RX PWM16/1
CN3
CN4
VIN
GND
NRST
5V
UART2_TX A7 PA_2
PWM17/1 ADC2/4 SPI1_MOSI A6 PA_7
PWM16/1 ADC2/3 SPI1_MISO DAC2/1 A5 PA_6
ADC2/2 SPI1_SCLK DAC1/2 A4 PA_5
PWM3/2 ADC2/1 SPI1_SSEL DAC1/1 A3 PA_4
PWM15/2 ADC1/4 UART2_RX A2 PA_3
PWM15/1N ADC1/2 UART2_RTS A1 PA_1
ADC1/1 UART2_CTS A0 PA_0
AREF
3v3
UART2_TX LED1 SPI1_SCLK D13 PB_3

Figure 12: STM32 Nucleo Board Pinout

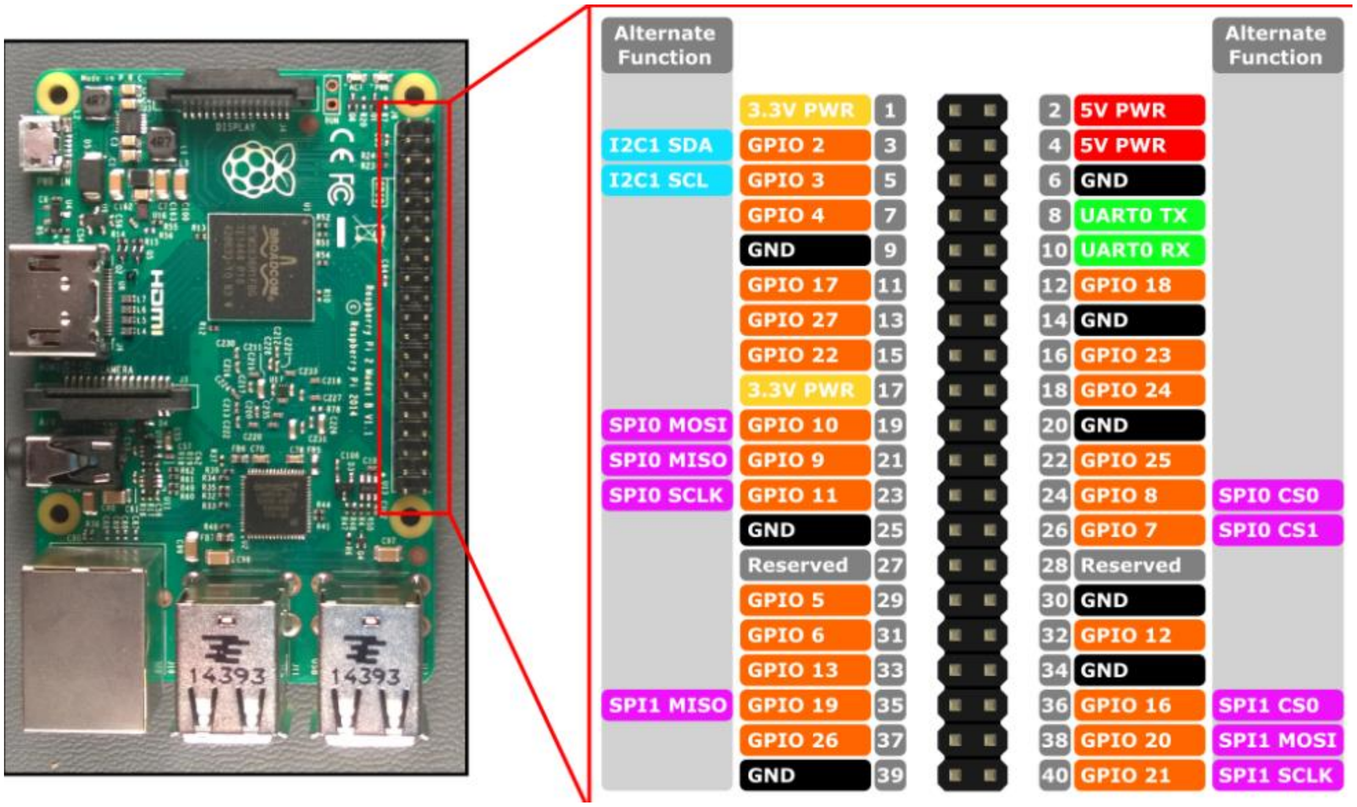


Figure 13: Raspberry Pi 3 Pinout

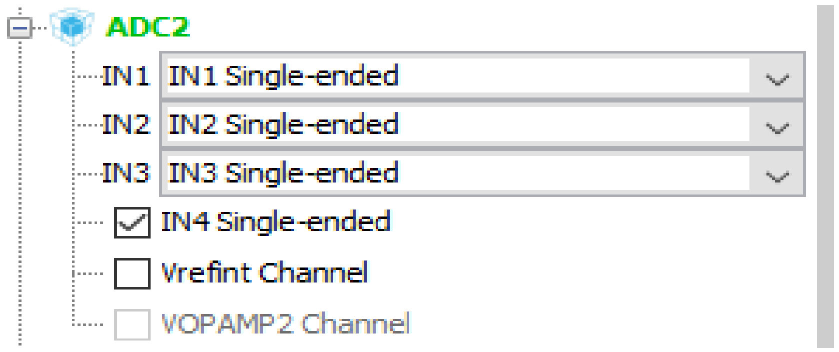
Make sure that the ground pin between the Nucleo Board and Raspberry Pi is also connected. Otherwise, you may receive unwanted data from the noisy environment.

Part III: ADC and Servo Set Up

For this part of the project, we needed to have the ADC function and PWM Generation Mode enabled on the STM32 Nucleo. The first half of this part will be dedicated to setting up Analog to Digital Conversion and wiring the LDR sensors properly. Then, the remaining portion will focus on generating Pwm signal to control the Servo Motors.

Step 3.1: Enabling ADC on STM32CubeMx

From the previous project we had created, where we establish UART communication with the Pi; we are going to include the built in ADCs. First we must hover back to the Pinout tab, and just like the UART steps expand the option ADC2. Once there we will be prompted with at least 6 options; because, we will need to read values from 4 LDR sensor, we will need to occupy four pins. Therefore, on the first four options tabs, we will choose Single-Ended as our option. All of the options that read “INx” are pins that can be used to convert analog signals to digital values. Differential Option is a dynamic option, that reads input from two pins and outputs the difference of those two pins. This is not required but it may be useful to know for the future. Also, take note of the pins that are assigned ADC on the right side of the chip diagram. Figure 13 and 14 demonstrate these changes.



ADC2
IN1 IN1 Single-ended
IN2 IN2 Single-ended
IN3 IN3 Single-ended
[✓] IN4 Single-ended
[ ] Vrefint Channel
[ ] VOPAMP2 Channel

Figure 14: Single Ended option chosen for each ADC pin

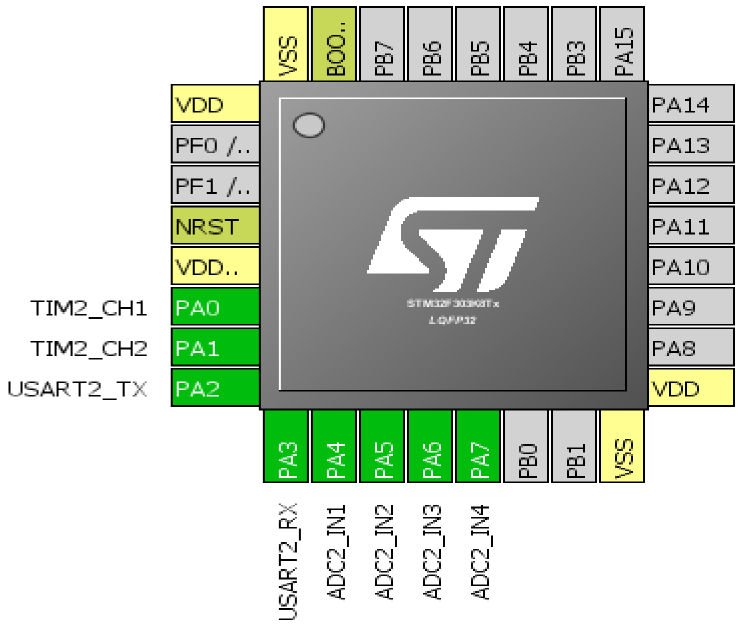
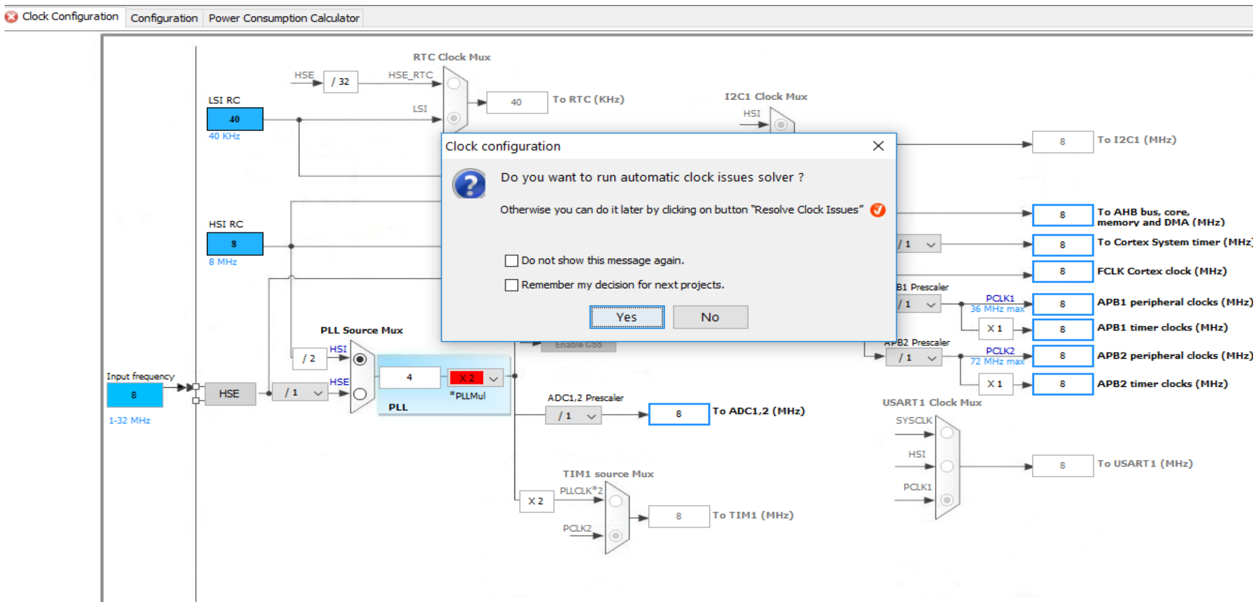


Figure 15: ADC Pinout starting at pin PA4

Step 3.2: Configuring the correct clock frequency

Once the pin configuration is complete, we will need to adjust the internal clock frequency of the pins. To do this, we must hover over to the clock configuration tab and select it. Once here we may be prompted with an error message, saying that there is an error in the clock configuration and if we want to automatically resolve it. Click “Yes” to solve the problem. On the STM32 board, ADC can function up to 16 MHz, therefore, we chose a frequency below this range which was around 4 MHz. The following figures demonstrate these changes.



Clock configuration

Do you want to run automatic clock issues solver ?

Otherwise you can do it later by clicking on button "Resolve Clock Issues"

☐ Do not show this message again.

☐ Remember my decision for next projects.

Yes No

Figure 16: Correcting the clock frequency for the ADCs

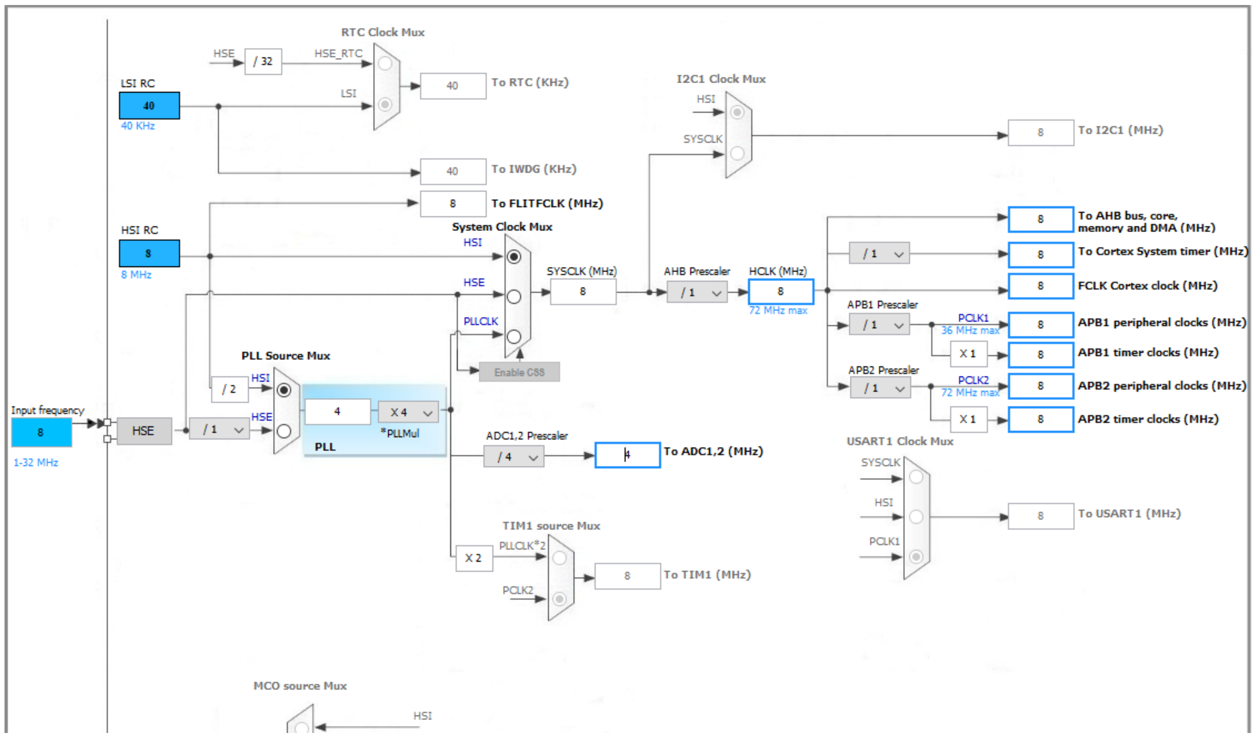
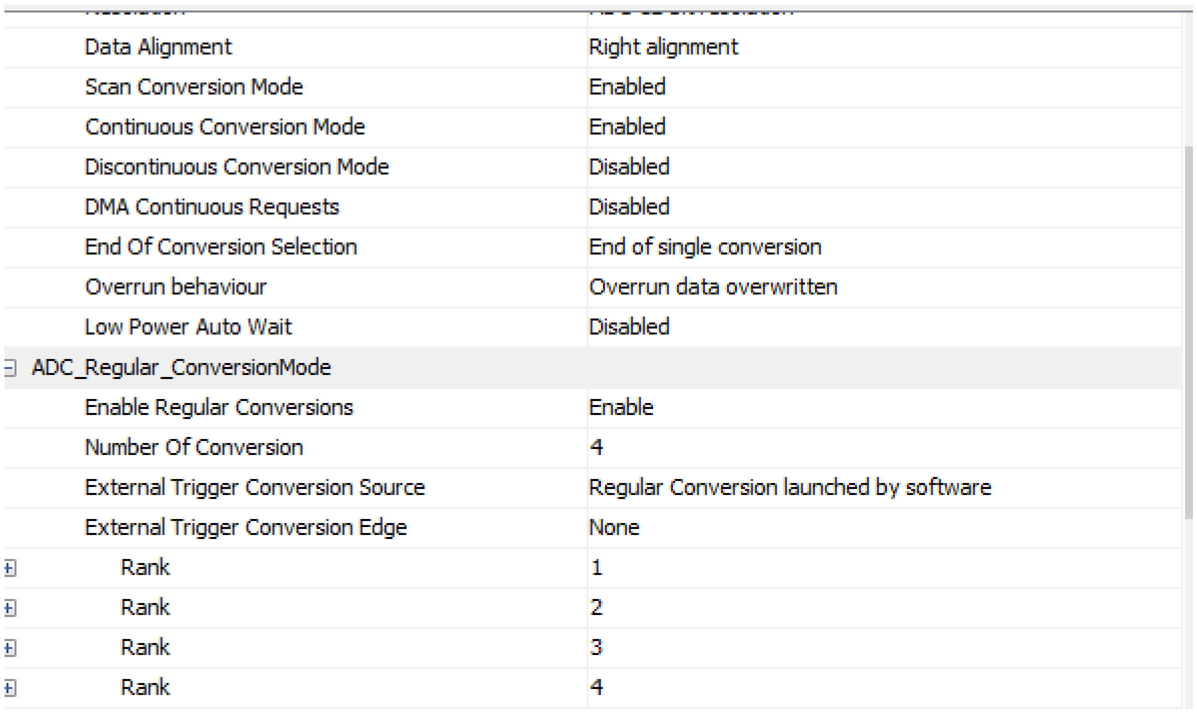


Figure 17: Clock Configuration changed to 4 MHz

The final step for this part will be to set the proper reading parameters for the ADCs. To do this we will now hover over to the Configuration tab and in it select the ADC2 button listed under the ADC category. Once here we will need to change a couple of options to have the MCU properly set up to read values from the four LDRs.



Data Alignment	Right alignment
Scan Conversion Mode	Enabled
Continuous Conversion Mode	Enabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	End of single conversion
Overrun behaviour	Overrun data overwritten
Low Power Auto Wait	Disabled
ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Number Of Conversion	4
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1
Rank	2
Rank	3
Rank	4

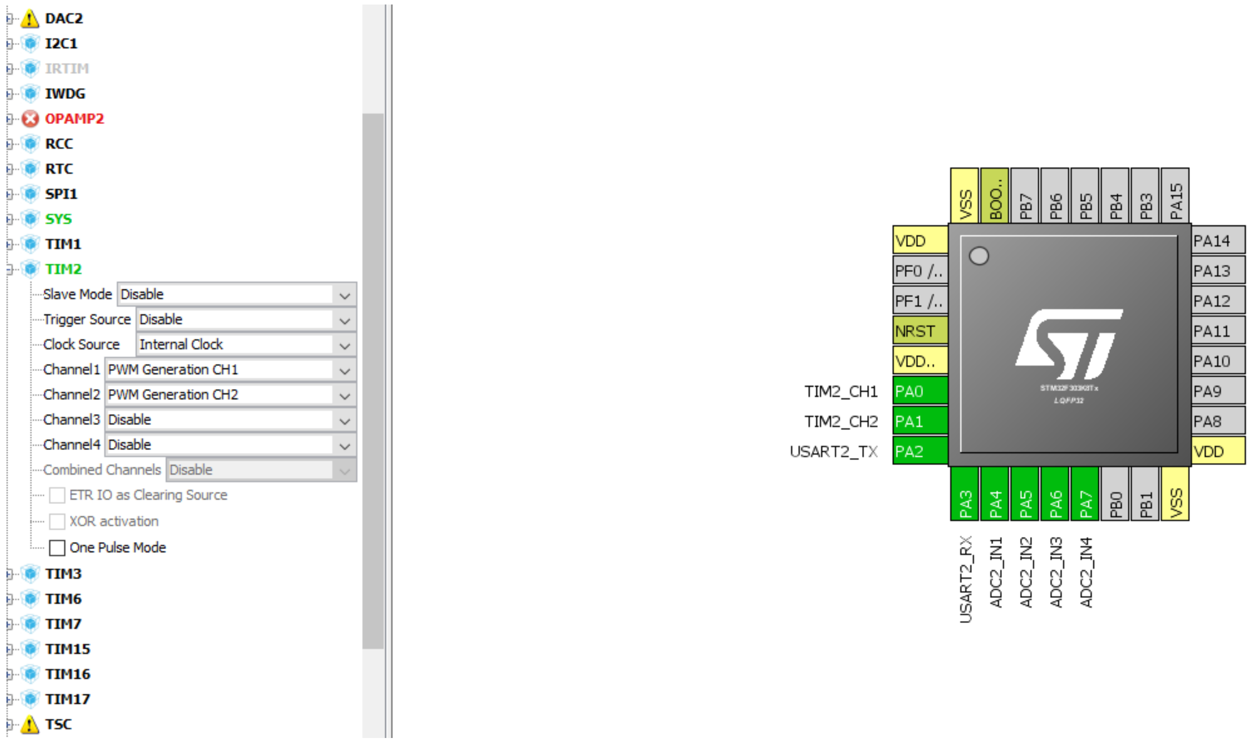
Figure 18: Parameter Setting for ADC2

Step 3.3: Wiring the LDRs up

In order to have the LDRs properly wired up, we will need to have one side of the sensor powered on 3.3 V. The other side will have a 100k Ohm resistor wired to ground. Our LDR sensors ranged in resistance from 50k Ohm up to 5 MOhms; therefore, we thought 100k Ohms was an appropriate value to have accurate readings. Lastly, we will need a jumper wire, wired between the LDR pin and 100k Ohm resistor. The other end of the jumper wire will go to the pin assign Analog to Digital Conversion. These steps must be taken for each LDR sensor.

Step 3.4: Servo Control using PWM Generation

The first step will be to set up the pins on the STM32 Nucleo board to generate PWM waves. To do this we must hover over to the pinout tab one last time. For this project, we went ahead and used TIM2 which according to the reference manual, is a 12-bit General Purpose Clock Timer capable of generating PWM waves. We will expand the TIM2 option and select for Clock Source-Internal Clock, for Channel 1-PWM Generation Mode CH1, and for Channel 2 PWM Generation Mode CH2. Since we will be using two servos to control the solar tracker from both axis', two channels will be required. Figure 18 shows these options chosen.



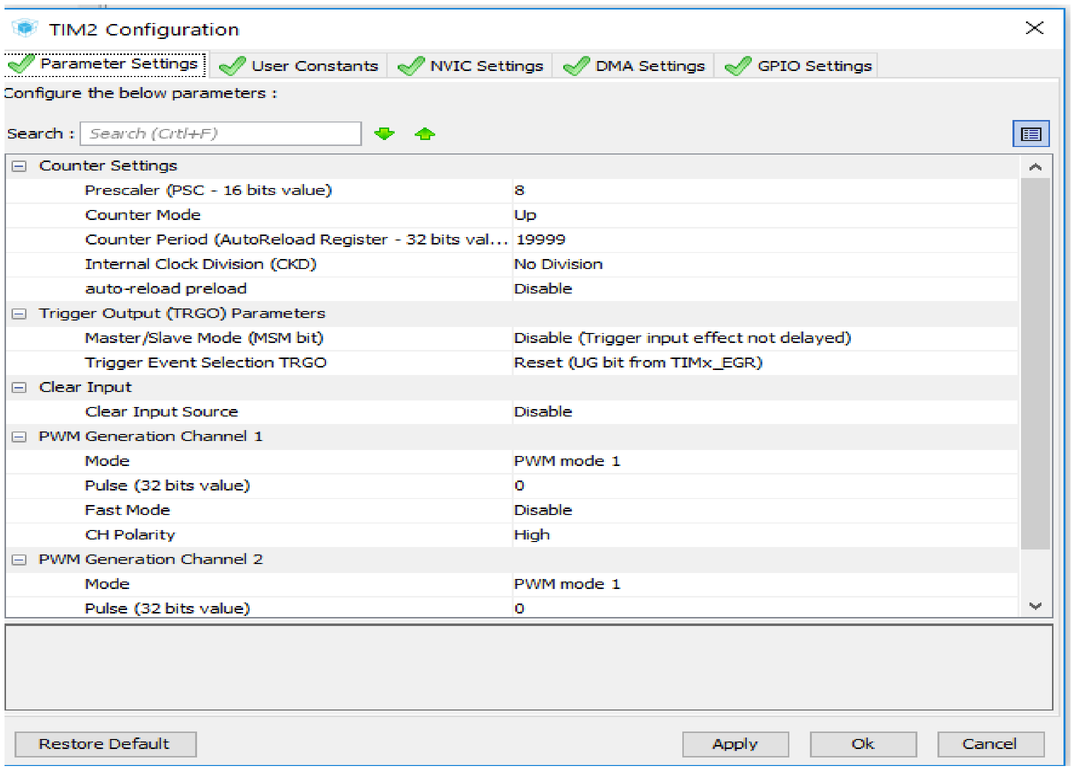
Slave Mode	Disable
Trigger Source	Disable
Clock Source	Internal Clock
Channel1	PWM Generation CH1
Channel2	PWM Generation CH2
Channel3	Disable
Channel4	Disable
Combined Channels	Disable

☐ ETR IO as Clearing Source
☐ XOR activation
☐ One Pulse Mode

Figure 19: Options chosen for the TIM2 Peripheral

Step 3.5: Setting up the Correct Clock frequency

Because most servos run 20ms periods, meaning that they run at frequency around 50Hz, we will need to lower our internal clock. By default, the clock running on the TIM2 peripheral is 8MHz. To lower this down will need to use prescalars and a counter period to lower the frequency. Figure 19 shows the proper parameter settings needed to lower the frequency to 50 Hz. Be aware that these settings only apply to a default internal clock of 8 MHz.



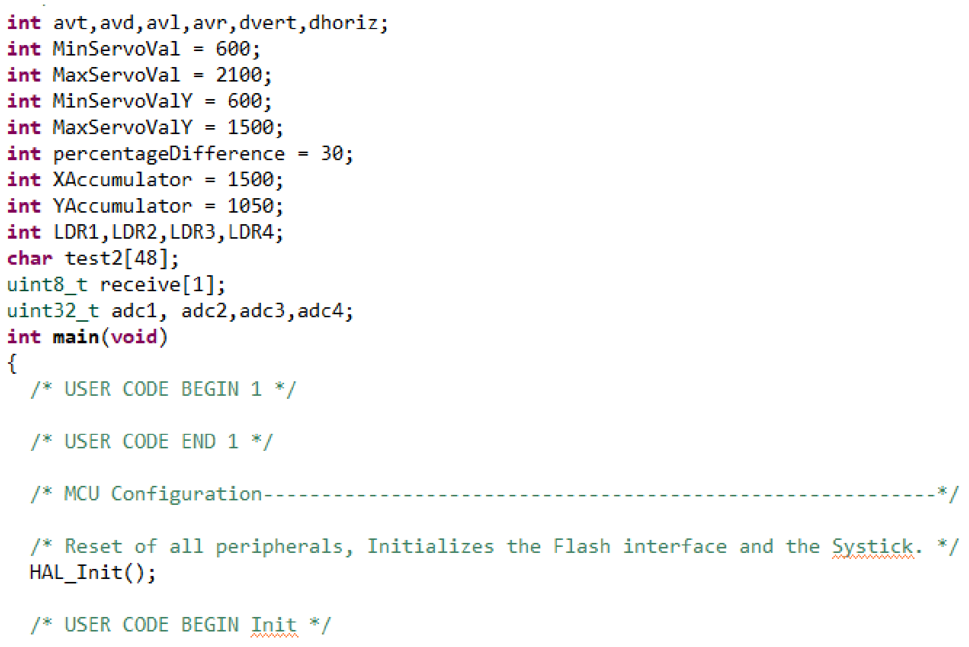
Configure the below parameters :

Counter Settings	
Prescaler (PSC - 16 bits value)	8
Counter Mode	Up
Counter Period (AutoReload Register - 32 bits val…	19999
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable
Trigger Output (TRGO) Parameters	
Master/Slave Mode (MSM bit)	Disable (Trigger input effect not delayed)
Trigger Event Selection TRGO	Reset (UG bit from TIMx_EGR)
Clear Input	
Clear Input Source	Disable
PWM Generation Channel 1	
Mode	PWM mode 1
Pulse (32 bits value)	0
Fast Mode	Disable
CH Polarity	High
PWM Generation Channel 2	
Mode	PWM mode 1
Pulse (32 bits value)	0

Figure 20: Parameter Settings for the Servos

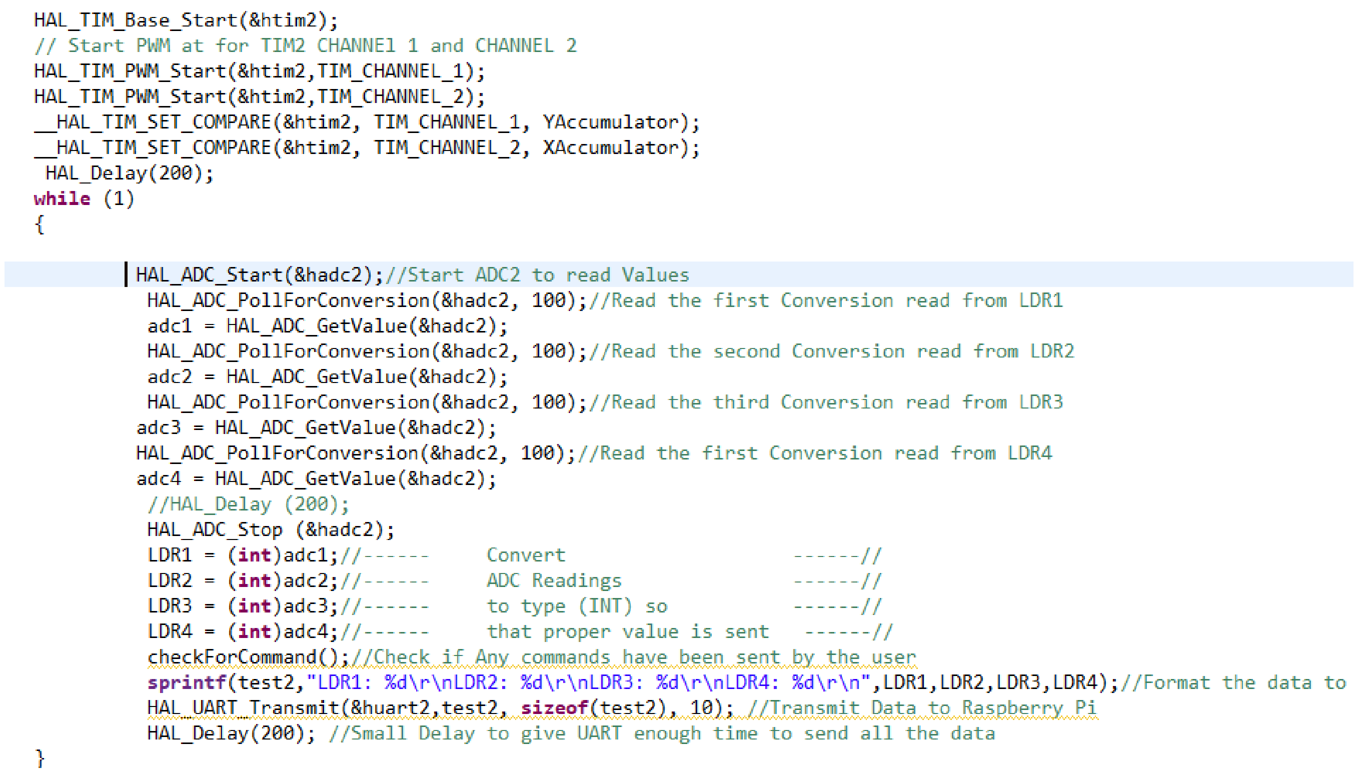
Part IV: Writing the main program for the STM32 Nucleo Board

This is the final step for having our solar tracker to successfully run. We are going need to generate the code for the project we have been setting up on the application STM32CubeMX. Once all the settings for UART, ADC, and TIM2 are set, we can go ahead generate the code. Using Attolic IDE Studio we will now begin writing our final program. First we must declare a couple variables that will need to be used. We will need variables for the Minimum and Maximum position that the Servos can turn for both the X and Y axis. We will also need 4 int variables that will store the value of each LDR reading. Once these variables are created, we will need to create a function that will check to see if any commands have been received from the Raspberry Pi. Using case statements, we will determine which command was received and move the Servos accordingly. After this, we will need to clear the buffer from the Rx pin, so that no one command runs forever. After this the final part will be to send the values of each LDR back to the Raspberry Pi. Figures 20,21, and 22 demonstrate each step discussed.



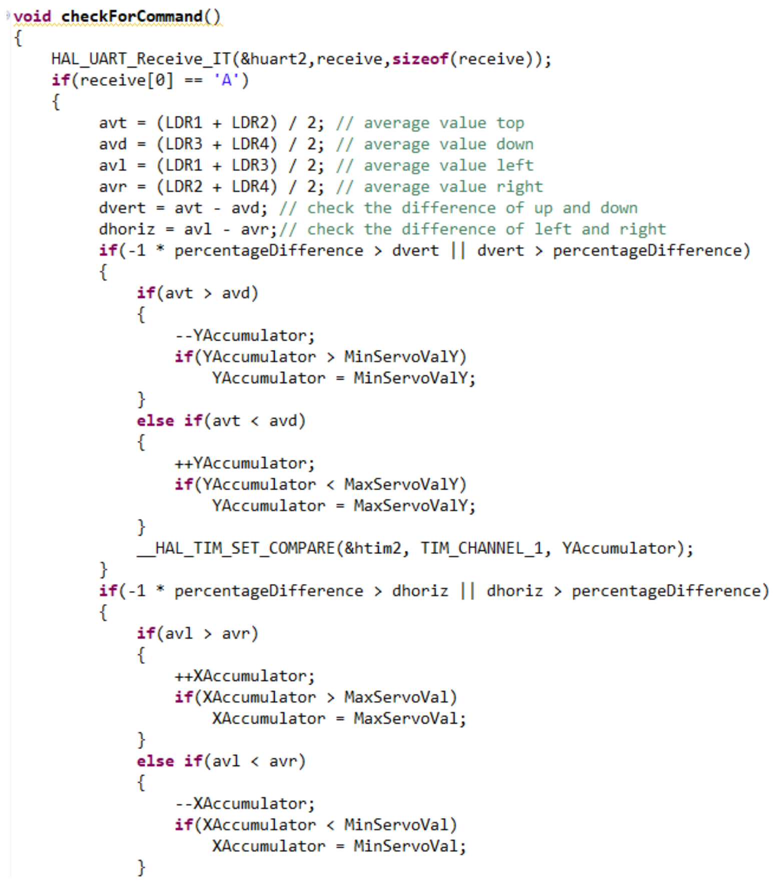
```
int avt,avd,avl,avr,dvert,dhoriz;
int MinServoVal = 600;
int MaxServoVal = 2100;
int MinServoValY = 600;
int MaxServoValY = 1500;
int percentageDifference = 30;
int XAccumulator = 1500;
int YAccumulator = 1050;
int LDR1,LDR2,LDR3,LDR4;
char test2[48];
uint8_t receive[1];
uint32_t adc1, adc2,adc3,adc4;
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration----------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */
```

Figure 21: Shows the variables that will needed for processing



```
HAL_TIM_Base_Start(&htim2);
// Start PWM at for TIM2 CHANNEl 1 and CHANNEL 2
HAL_TIM_PWM_Start(&htim2,TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim2,TIM_CHANNEL_2);
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, YAccumulator);
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, XAccumulator);
 HAL_Delay(200);
while (1)
{

         HAL_ADC_Start(&hadc2);//Start ADC2 to read Values
          HAL_ADC_PollForConversion(&hadc2, 100);//Read the first Conversion read from LDR1
          adc1 = HAL_ADC_GetValue(&hadc2);
          HAL_ADC_PollForConversion(&hadc2, 100);//Read the second Conversion read from LDR2
          adc2 = HAL_ADC_GetValue(&hadc2);
          HAL_ADC_PollForConversion(&hadc2, 100);//Read the third Conversion read from LDR3
         adc3 = HAL_ADC_GetValue(&hadc2);
         HAL_ADC_PollForConversion(&hadc2, 100);//Read the first Conversion read from LDR4
         adc4 = HAL_ADC_GetValue(&hadc2);
          //HAL_Delay (200);
          HAL_ADC_Stop (&hadc2);
          LDR1 = (int)adc1;//------     Convert                     ------//
          LDR2 = (int)adc2;//------     ADC Readings                ------//
          LDR3 = (int)adc3;//------     to type (INT) so            ------//
          LDR4 = (int)adc4;//------     that proper value is sent   ------//
          checkForCommand();//Check if Any commands have been sent by the user
          sprintf(test2,"LDR1: %d\r\nLDR2: %d\r\nLDR3: %d\r\nLDR4: %d\r\n",LDR1,LDR2,LDR3,LDR4);//Format the data to
          HAL_UART_Transmit(&huart2,test2, sizeof(test2), 10); //Transmit Data to Raspberry Pi
          HAL_Delay(200); //Small Delay to give UART enough time to send all the data
}
```

Figure 22: Demonstrates the commands needed to Read the values from each LDR [HAL_ADC_PollConversion()]

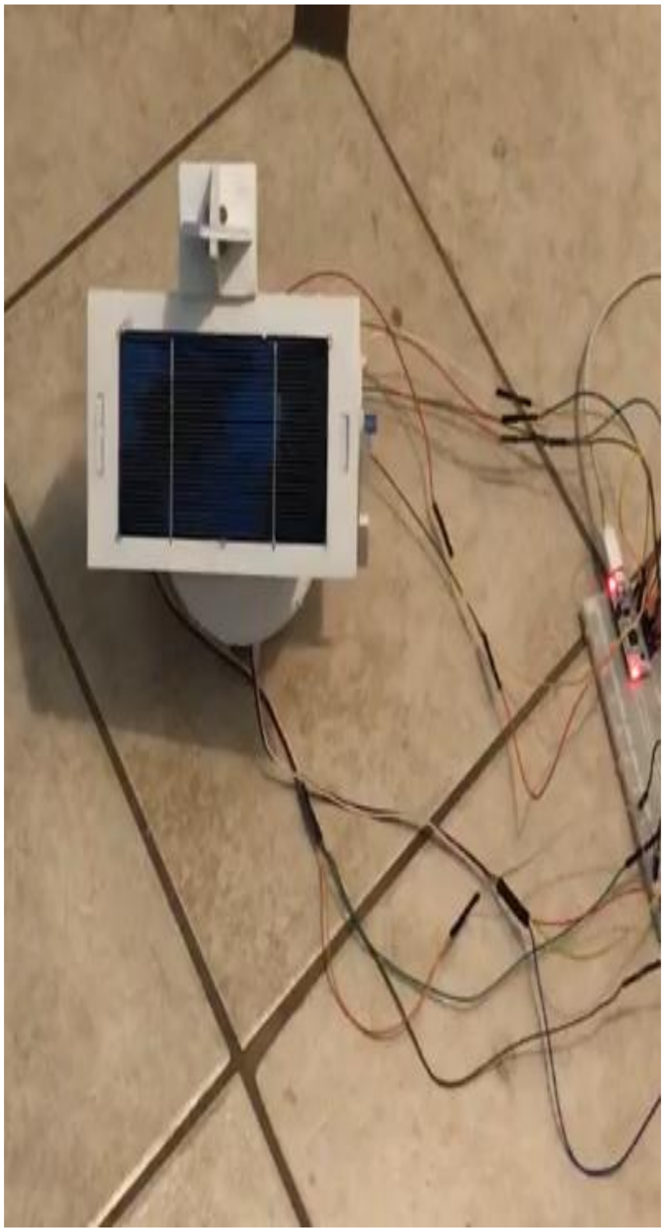


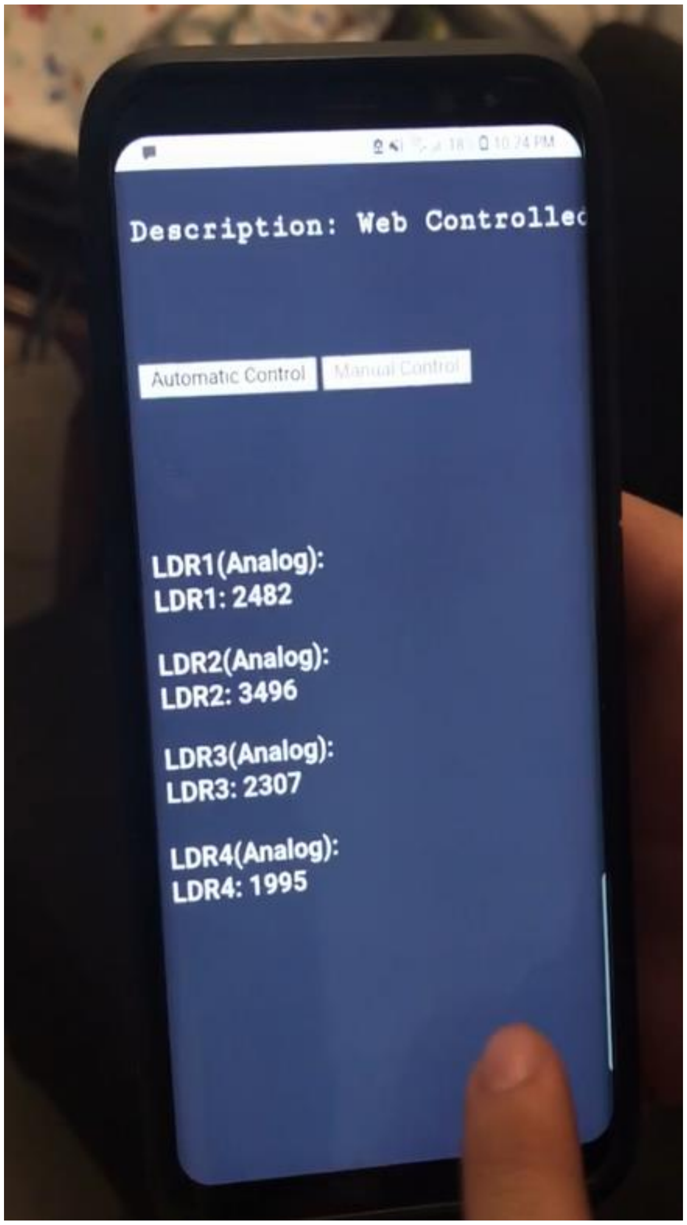
```
void checkForCommand()
{
    HAL_UART_Receive_IT(&huart2,receive,sizeof(receive));
    if(receive[0] == 'A')
    {
        avt = (LDR1 + LDR2) / 2; // average value top
        avd = (LDR3 + LDR4) / 2; // average value down
        avl = (LDR1 + LDR3) / 2; // average value left
        avr = (LDR2 + LDR4) / 2; // average value right
        dvert = avt - avd; // check the difference of up and down
        dhoriz = avl - avr;// check the difference of left and right
        if(-1 * percentageDifference > dvert || dvert > percentageDifference)
        {
            if(avt > avd)
            {
                --YAccumulator;
                if(YAccumulator > MinServoValY)
                    YAccumulator = MinServoValY;
            }
            else if(avt < avd)
            {
                ++YAccumulator;
                if(YAccumulator < MaxServoValY)
                    YAccumulator = MaxServoValY;
            }
            __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, YAccumulator);
        }
        if(-1 * percentageDifference > dhoriz || dhoriz > percentageDifference)
        {
            if(avl > avr)
            {
                ++XAccumulator;
                if(XAccumulator > MaxServoVal)
                    XAccumulator = MaxServoVal;
            }
            else if(avl < avr)
            {
                --XAccumulator;
                if(XAccumulator < MinServoVal)
                    XAccumulator = MinServoVal;
            }
```

Figure 23: Demonstrates the function checkForCommand(), which checks if a command was received from the Raspberry Pi; and performs a certain action accordingly.

Final Step: Running all the Programs

Now that all the code is completed, the final step will be to run the following programs: The GetData.py script saved onto the Raspberry Pi, the main.c Program written for the STM32 Nucleo, and opening up a web browser and navigating to the Web Page Interface. To navigate to the web page, we need to know the IP address of the Raspberry Pi, and input it onto the Search Bar of the Web browser. Then, we need to run the python script on the Raspberry Pi, by using the command “python GetData.py” on the shell terminal. Finally, on the STM32 Nucleo, we will need to program the main.c file onto the Nucleo’s memory, simply by running Debug on Atollic TrueStudio. Atollic will automatically program the Nucleo whenever you choose to debug your program. If all the steps are correctly followed, you should now have a fully functional Web Controlled Solar Tracker with Automatic and Feedback Capabilities!





LDR1(Analog):
LDR1: 2482

LDR2(Analog):
LDR2: 3496

LDR3(Analog):
LDR3: 2307

LDR4(Analog):
LDR4: 1995

Conclusion:

This was a relatively complex, diverse, and ambitious project to do; as it required vast amounts of researching in order to complete each portion of the lab. Since, this was the first semester ever that we used the STM32 architecture; we needed to research and learn a lot of its features. However, the wonderful thing about this board, is that it has so much written documentation on their official website (STMicroelectronics) that it does an exceptional job in teaching you about each feature within the board. In the addition, the Raspberry Pi has an enormous community that finding solutions to certain problems is not too hard. This project had a variety of different material, that included AJAX, ADC, PWM, etc. Overall, the Solar Tracker project helped us have a deeper understanding of microcontrollers; and also how embedded systems can be potentially built. It was great practice for the Senior Design Project that will be coming up in our future classes.