

Why particle board?

After a large amount of hours spent on looking for a microcontroller to act as not only a data logger to a remote server, but also a processor, powerful enough to control the working of the sabot. It was decided after a meeting and particle Evaluation board was selected among the other options that were presented.

Features:

The E Series Evaluation Kit contains a breakout board which provides easy access to all the pins and peripherals available on all E Series module variants. The Evaluation Board includes:

- Physical USB access for flashing and serial communications
- Li-Po battery connector for connecting external batteries
- Power barrel jack connector for external power
- MODE and RESET buttons
- Charge and status LEDs to appreciate device state at a glance

The 2G/3G E Series Evaluation board contains a soldered E310 module, which supports worldwide 2G and 3G connectivity and can be configured into a 2G only mode.

The LTE E Series Evaluation board contains a soldered E402 module, which supports LTE M1 connectivity.

All Particle Hardware is designed from the ground up to work with the Device Cloud. The Device Cloud is a powerful set of tools and infrastructure to build, connect, and manage your IoT fleet.

Access to the Device Cloud includes:

- 3MB of cellular data per device/mo (additional data [\\$0.40/MB for most countries](#))
- First 3 months of Device Cloud FREE (\$2.99 per device/mo after)
- Device Cloud Features:
 - Device Management
 - Over the Air Firmware Updates
 - Fully Managed Connectivity
 - Developer Tools
 - Integrations

Contents:

- E310 (2G/3G) or E402 (LTE) module mounted on an evaluation board
- Li-Po battery
- Taoglas Penta-band Cellular Antenna (u.FL)
- SMA to u.FL adapter for connecting external SMA antennas
- Grove temperature sensor

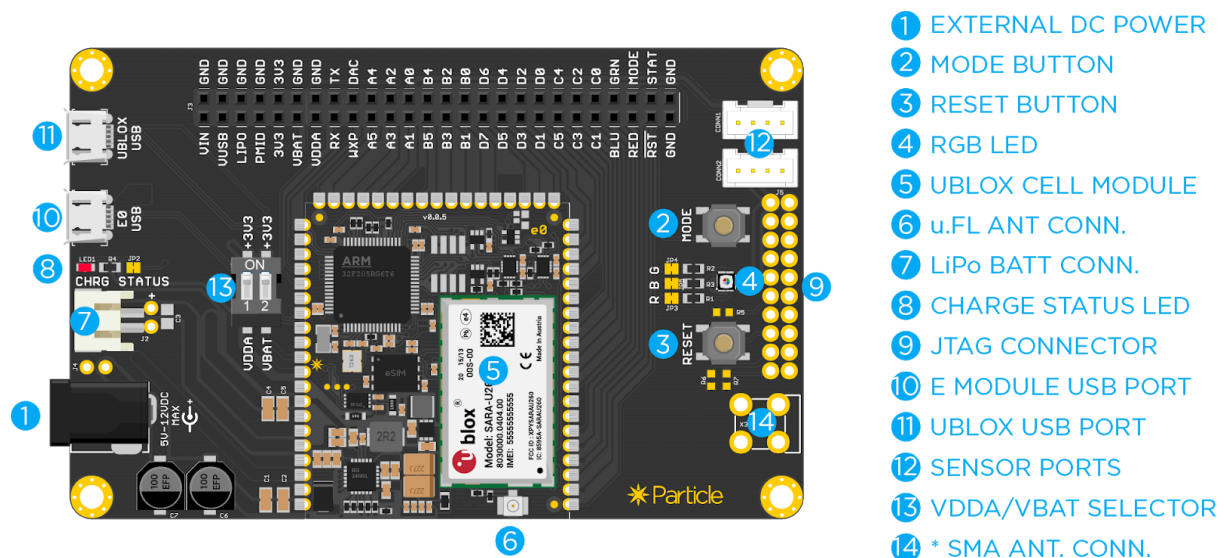
- USB Micro cable
- Pinout reference card
- 3.3v 12bit analog to digital converter.

Specs:

- STM32F205 ARM Cortex M3 microcontroller
- 1MB Flash, 128K RAM
- Cellular modem: u-blox SARA U-series (3G Global with 2G fallback / LTE)
- 36 pins total: 28 GPIOs (D0-D13, A0-A13), plus TX/RX, 2 GNDs, VIN, VBAT, WKP, 3V3, RST
- Board dimensions: 36mm x 43mm

These are the features of the board. Apart from that please read the data sheet for more info. And before starting to work on particle board. Please read the datasheet thoroughly and handle the board as per the power requirements stated in the datasheet. The datasheet can be found in the following link:

<https://docs.particle.io/datasheets/cellular/e-series-eval-board/>



NOTE: PLEASE READ THE PIN INFORMATION. SOME PINS ARE NOT 5V tolerant.

NOTE: This manual written knowing whoever reads this has basic knowledge on programming concepts and electronics concepts (:

PROCESS:

The board is a 3.3v board. That is: it's reference and working voltages are 3.3v. Digitals pins are 5v tolerate. (i.e) they can withstand max input of 5v. But others pins are not 5v tolerant. So, please make sure you don't input high voltages into the pins.

SENSORS:

This part covers all about the sensors.

Current sensor:

The current sensor used is a ACS712 30A. This sensor is a hall effect sensor, bi-directional sensor. Which means it can measure current from both directions. You can see the datasheet in the datasheet folder in the project. The current sensor works as follows.

The current sensor **must** be powered by a 5v with utmost 120mA of power. Make sure the power source is stable and it is grounded properly, else the readings will deviate by a large degree giving high errors. Also, Don't try with Vusb pin of the particle board. The Vusb

The sensor has 5 pins, vcc, gnd, vout, ip+ and ip-. The wire whose current flow has to be measured must be connected to ip+ and taken out from ip-. It is to be noted that **the current sensor MUST be connected in series**. Else, you are shorting the circuit. Bad of you.

When there is no current flowing between the ip+ and ip- pins of the current sensor, and it powered by a proper 5v power supply and grounded properly; You can measure the output voltage between Vout pin and gnd. It will be 2.5volts. Or to be more accurate it will give half of whatever voltage your power supply is powering your current sensor. So, if you supply 4.8v, the Vout voltage will be 2.4v. So, when there is a current flow between the ip pin, the voltage will reduce or increase based on the direction of the current. (ip+ to ip- increase and decrease for the other way around). When +30A current is flowing between the ip pins, we can expect 5v on the Vcc pin. Now, the particle board analog pin can read a voltage upto the maximum of 3.3v. This is where the reason for using 30A current sensor kicks in. The max current from the solar panels will not exceed 8A. Which means, for a very small fluctuation in the voltage of Vout pin, the current difference will be more. For 8A it will never reach above 3.3v so, the pins are safe from over voltage and current can be measured accurately :)

PROGRAM SIDE:

Now, the program side of the sensor. We read the vout pin by reading the analog pin A1 where it is connected. If you see in the current function you can see the line "`currentValue+=(3.3/4095)*analogRead(pin);`" this read the analog input and converts it into the voltage. So, we know the reference voltage (i.e) 2.4 Or 2.5 depending on the power supply. We can calculate the error, and divide it with the sensitivity.(sensitivity can be found in the datasheet of the sensor. It is 184mV/A for 5A, 100mV/A for 10A and 66mV/A for 30A). The currentvalue needs a currentReference value inorder to give accurate readings. So, a series of samples are taken when starting the board and that is fixed as the reference point. **THIS MAKES IT OBVIOUS THAT CURRENT AT THE CURRENT SENSOR DURING START OF THE DEVICE MUST BE ZERO.** In the program, i have taken upto 3000 samples of reading and taken the average of that. Then used the average to calculate the current value.`currentValue+=(3.3/4095)*analogRead(pin);` this line takes the current value.

Voltage Sensor:

The voltage Sensor is nothing but a voltage divider circuit. Due to the 3.3v voltage limit of the analog to digital pins of the particle board. I had to create a new voltage divider circuit that was compatible with the board. A voltage divider is a resistive circuit that is used to stepdown DC voltage. In a voltage divider circuit, the

input voltage is directly proportional to the output voltage. **AND THE CIRCUIT MUST BE CONNECTED PARALLEL TO THE MEASURING TERMINALS, NEVER SERIES. FAILING TO DO SO WILL SHORT THE TERMINALS**

You can see in the image below, how it is calculated and the voltages we are working with. The solar panels will generate a max of 24v. So, with 25v as a margin, for 25 it will reach a voltage of 3.193v with a safety margin of .2 volts

higher voltage and converts it to a lower one by using a pair of resistors. The formula for calculating the output voltage is based on Ohms Law and is shown below.

$$V_{out} = \frac{V_s \times R_2}{(R_1 + R_2)}$$

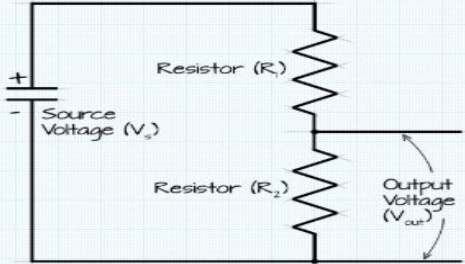
where:

- V_s is the source voltage, measured in volts (V).
- R_1 is the resistance of the 1st resistor, measured in Ohms (Ω).
- R_2 is the resistance of the 2nd resistor, measured in Ohms (Ω).
- V_{out} is the output voltage, measured in volts (V).

Enter any three known values and press "Calculate" to solve for the other.

Voltage Source (V_s)	<input type="text" value="25"/>	Volts (V)
Resistance 1 (R_1)	<input type="text" value="56000"/>	<input type="text" value="ohms (<math>\Omega</math>)"/>
Resistance 2 (R_2)	<input type="text" value="8200"/>	<input type="text" value="ohms (<math>\Omega</math>)"/>
Output Voltage (V_{out})	<input type="text" value="3.193"/>	Volts (V)

Click "Calculate" to update the field with orange border.



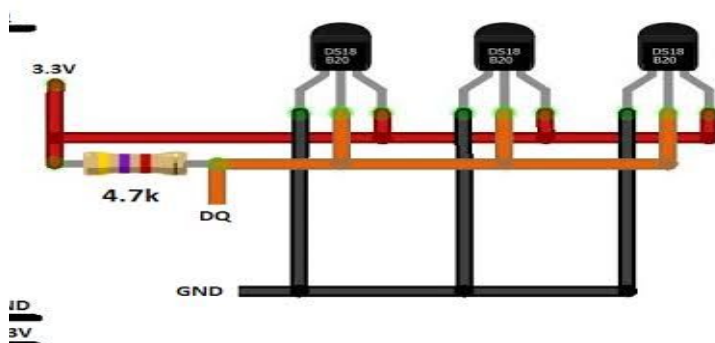
In this, V_{in} is the voltage to be measured and v_{out} is the voltage pin that goes to particle board. The resistance value I have used are $R_1 = 56000$ ohm and R_2 is 8200 ohm. This much I value of resistance will draw no current and won't close the circuit.

PROGRAM SIDE:

We can read the analog input in the pin B2. note that B2, B3, B4, B5 are analog pins too. So, we use this line `vout = (3.3/4095)*analogRead(pin);` to read the data and we calculate the V_{in} voltage using `vin += (vout / (R2/(R1+R2)))`; we take 100 samples and take an average of it. And send that data.

TEMPERATURE SENSOR:

The temperature sensor is an IC based sensor, NOT AN RTD. The IC is in the



Probe itself. This temperature sensor will be a bit complicated for you if you don't know the onewire communication protocol by Dallas Semiconductor group. The semiconductor works on both 3.3v and 5v. I'd recommend using 3.3v as the data line will be pulled-up to the voltage of VCC which might be more than 3.3v. The datasheet is given in the

project folder. You don't have to worry about onewire protocol as we are going to use a library B^). The sensor has three pin. Vcc (Red), Gnd (black), and data(yellow). The best part about this sensor is multiple sensors can be connected to a single pin and the data can be read with good speed. There must be a pull up resistor added between the data line and the vcc line as shown in the picture

The resistor value should be 4.7kohm. But I tested it with 1kohm resistor and it worked as expected.

You can see in the picture from what point that data line tapped and connected to the microcontroller. Here in the schematics provided I have used pin D1.

PROGRAM:

This might be a little bit complicated but it is doable. First, we need two libraries. As you guessed from the above paragraph, we'll be using OneWire library and spark-dallas-temperature library. Spark-dallas-temperature depends on the onewire library. OneWire library handles the protocol and the sparkdallastemperature handles the temperature sensor readings. Please refer to the examples in sparkdallastemperature library in the lib folder of the project to know what the commands do. Let me walk you through the program. First, as it has been said earlier, all the three temperature sensor are connected to the same pin. Now, to identify which sensor's data belongs to which one, we need something called a device address. To find device address of each temperature sensor you can either use arduino or the particle board. The program to find the address of the sensor is available in the project folder "solardatalogger particle board"->functions->TemperatureAddressFinder.ino. Run this program by connecting to the mentioned pin. The pin number can be changed based on whether you are using arduino or particle board. Run the program, move it to src folder, hit compile and flash the program using the particle cli (command prompt) you can see the address on the serial monitor. Each device as a 64bit address. You will see it has 16 hexa decimal numbers in the address program. We will be using the last two digits of the address to identify the device and allot them to particular object. For instance in the program you can see this array declaration : `const int c[] = {0x33,0x81,0xF9};` This declaration has address in 8 bit hexa decimal numbers. 0x33 represents the least significant byte of the address. Now that address terminology is sorted, we can go to the program.

```
OneWire oneWire(Temp); //this is object function of temperature sensor
```

```
DallasTemperature sensors(&oneWire); //this is the object that uses the previous object
```

```
DeviceAddress bat,ucp,cp; //variable to store address of the temperature sensor
```

```
const int c[] = {0x33,0x81,0xF9}; // last two bit of the address
```

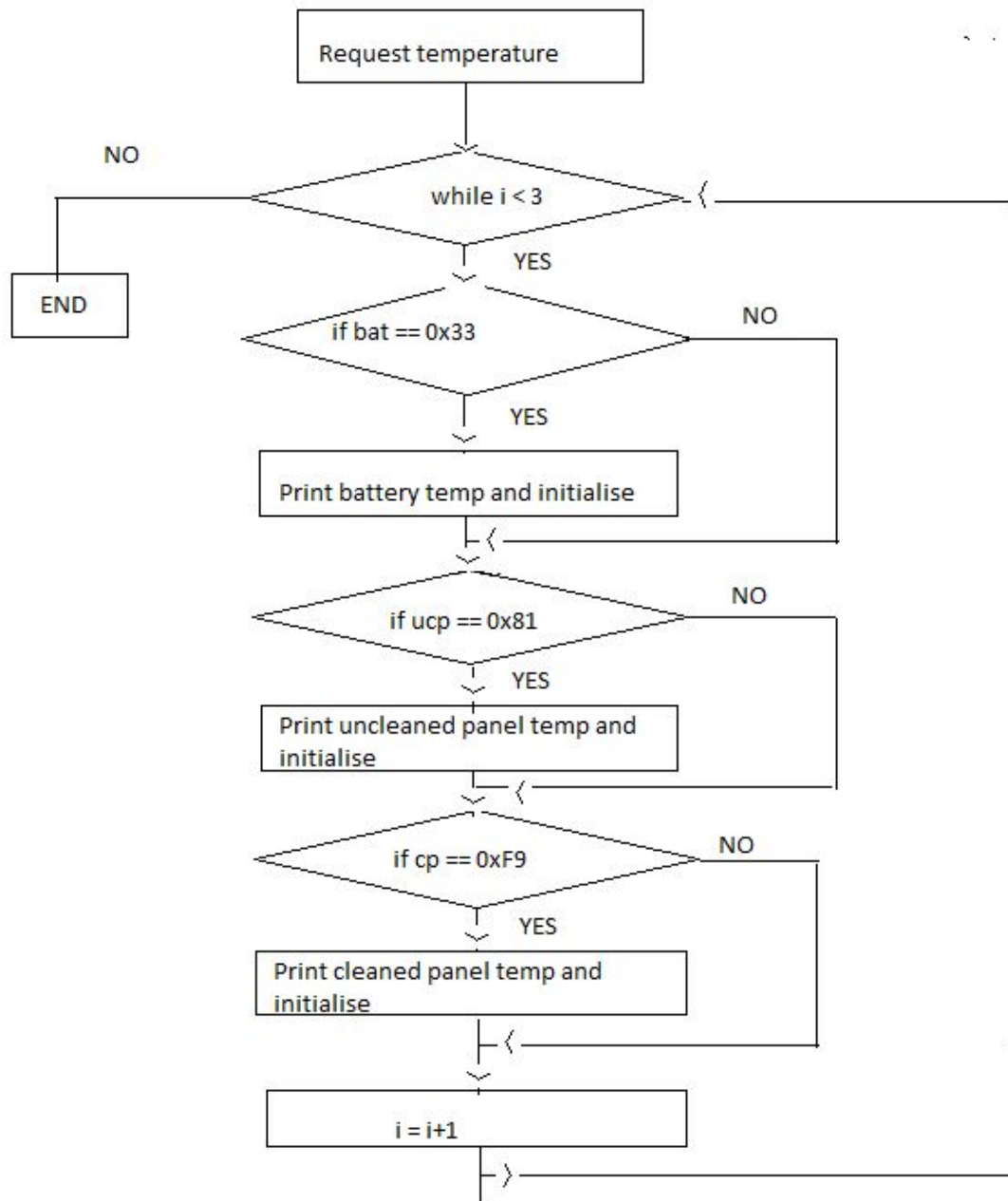
This is the initial stage of the sensor. We define a object, then we pass the address of the object to the dallaslibrary. The deviceAddress is a custom data type that can store the address of the sensor. It is basically an array of size 8. After this process, let's get to the tempFunc(); where the actual temperature is calculated.

Here I am taking an example of the existing address to explain how it works. It can change when different new sensors whose address are not registered are used.

There are two functions used to configure the temperature sensor. One is the tempcheck() function. And other is the tempFunc();

tempcheck() --: This function checks how many sensors are connected to it. And depending on the number of sensors, it stores the address, resolution and set the flag. `sensors.getDeviceCount() == 3` This line checks whether the number of sensors connected is 3 if not, it'll lower the flags, if yes. It will make the flags high. I will cover the flags part later. `sensors.getAddress(bat, 0)` This line gets the address. This stores the first sensor sending the address in a variable. Here, it stores in bat variable. The index 0 you are seeing is the order in which the data from the device is received. The order the data is received is always same as long the sensors are not changed to a different one. Now we get the address and store it in the variable. The work of tempCheck() function is done.

tempFunc() --: This is the function that read the temperature. The flowchart will give a better understanding.



This flowchart pretty much shows how the program works. The bat is the variable that stores the address of the temperature that is kept near battery and its address is stored in the tempCheck function. Similarly for other variables the same method is followed to check with address and assign value and determine which temperature sensor data is stored in what variable. Refer the code and examples in the sparkdallastemperature folder to have a better idea on what the certain line of code does. There is not need to assign pins for this in pinMode() function.

SD CARD:

This is not a sensor, but this is an external peripheral that we have connected. We have used a sparkfun microsd card adapter which has the following changed. The MISO pin is renamed as DO and the MOSI pin is renamed as DI. CS pin is the chip select, SCK is the clock and CD is the chip detection which we are not using. **The adapter works ON 3.3V ONLY.** Don't power it with more than 3.3V. **IT WILL FRY THE MICROSD IN THE ADAPTER.** If you have read the data sheet thoroughly, you

can see that there are three sets of pin for spi communication. SPI is a protocol that we will use to read and write data from and to the sd card. The pins used for this communication is written in the program. But for the sake of it, you can see it here too. // SCK => A3, MISO/DO => A4, MOSI/DI => A5, SS/CS => A2 (default) . The sd card wont work if even one pin is misconnected or not connected. So, take care of the connection. Don't worry about integrating, we'll be using libraries here so don't need to be familiar with the SPI protocol.

PROGRAMMING SIDE:

The program is handled by sdfat library from the particle dev. It's in the lib folder. Just read the comments near the program. It will be easy to understand. There is no flow, it just loads the data in a particular order. It doesnt open a new file a data is added. It opens an existing file and it stores the data there, data.csv is the file it stores the data. There is not need to assign pins for this

FTP SERVER:

There is no hardware needed for ftp server data sending and receiving data. Well, we will be doing only the sending part. What is a ftp server? Refer this link: https://en.wikipedia.org/wiki/File_Transfer_Protocol . If you have read the article, you'd know by now that there are two ways to communicate. Active mode and passive mode. In this project, we'll be using passive mode. Why? It doesn't need any manual authentication from the host server and can be done with just the credentials. This makes this mode easier to configure and work on. Also, since we are not receiving any data this mode becomes more ideal.

FTP server is not all flowers. It as a major drawback of ftp overhead. It is the extra bytes of data that is sent along with the data you are sending, for authentication, handshake purposes etc. This can be as much as few kilobytes for data as small as few bytes. This makes an ftp server less efficient way to work with. In my endeavour to save cellular data and to send the data we want, as many times as we want. It was a most inefficient. There is no way to cut the overhead as the File Transfer Protocol itself demands that many overhead bytes. The particle board has an e-sim. This means we have to use whatever sim that they have added during the manufacturing process. And the data charges are too much it's around 3\$ for 3MB of data per month: I know, pretty unfair.

PROGRAMMING SIDE:

We are going to, as many developers in the world, use a library to handle the ftp server data transmission. THE library is particleftplib. With this library it is not possible to host, but it is possible only to be a client (one that connects to a host).

The particleftplib is a simple and reliable library that doesn't offer much functionalities but it is more than sufficient for this application. Now, there is a certain process we have to follow in order to start the data transfer.

First, we have to open the port. We will passing `hostname` (who is hosting the server) what `port` used to communicate. Usually its 21, when it passed, the port number will be given by the host and then, that will be used for communication. This part is also handled by the library. The `timeout` is the time interval we will be waiting for the server to respond, if it doesn't respond, then we skip the connection.

```
ftp.open(hostname, port, timeout);
```


Second- once the port is open and the host responds, we pass the user name to login into ftp server. It is done by this line of code

```
ftp.user(userName)
```

This will send the username and make the ftp check if there is any username of that type, once it is accepted, we will pass password to the server-

```
ftp.pass(password);
```

Then viola, we are into the server. We have established connection to the server. NOTE THAT: We disconnect from the server if something goes wrong and skip the data sending for one-loop period (time taken for void loop() to execute on time- it depends on the contents of that function). It will throw error if it's not connected and will skip the connection for that minute. It is to be noted that, we have to establish connection every time we want to log data. I tried logging the data by opening the port just one time and continue logging the data. The port would close randomly and there is no way to check whether it is open or closed. So, sacrificing some data over reliability, I programmed to open every time we call the ftp function. This had less failures (at least 7/10) \o/. **All the functions in ftp accepts only string as input except for port and timeout.**

Now, we are inside the server, it is now time to log the data. We have to give the directory to which we have to send the data to. So, we use the cwd command to set the working directory. `ftp.cwd("Type address here")` the address to which the data is to be transferred must be given. (type the address between the double quotes, it only accepts string data type). Now, this function moves the address to the address we want in the ftp server. If the address is not available, it throws an error and the server is closed. The current address that is open in the ftp server can be checked using `ftp.pwd()` function. Once the address is given, it is time to upload the data.

We have to first say what type of data is the host expecting. So, we set the type of the data we are sending using this command. `ftp.type("A")` A for ascii character for transferring numbers, texts etc, B for binary character for transferring images, music etc. After setting up the file type, we have to create a file in the ftp server.

`ftp.stor(name)` this command is used to create the file with name we provide. In the place of name, we can pass a string/character for the name. Since, we want to have a unique name. So, we have taken the universal time and converted it into a string and then passed into the stor function. This will create a file with a unique name. We have to now write to the file. So we can use this function called `ftp.data.write()`; As any other function in ftp library, it only accepts string. The data we get from the sensors are float and integer. We will first convert the data into a string and then pass it into the ftp.write function. This will write the data into the server. Once the data is written, it's time to close the server.

The data is being sent, to prevent stopping the server when the data is being sent and prevent corrupted data. We first flush any data that is being sent. `ftp.data.flush()`; this will flush any unsent data that is on the data line. `ftp.finish()` ends the ftp connection and clears out any stray data. It stops the file upload. If you are wondering why we are creating a new file, every minute. Because, we are limited by memory. To send the data to the ftp server, we need to store the data in the RAM of the ublox module (handles cellular connectivity) in the particle board. The RAM of UBLOX module may overflow without our knowledge, so we cannot make the system reliable. The RAM size is small and we cannot simply store large amount of data. Any ways, we consulted with the CTT team and they said they receive most of their data in different files. So, it is fine to send in a single file. Once the file upload

has been stopped by `ftp.finish()` command. `ftp.quit()` command is used to come out of the server, thus we end a successful transaction of the data.

As you can see, so many steps to send a small data. Each command sends a packet of data which we define as ftp overhead. Anyways, apparently, this is the best/secure thing we have.

ADDITIONAL FUNCTIONS:

Apart from the functions that read data from the sensor, we have used some additional functions. They are as follows:

`int sdcheck();` this checks whether the sdcard is connected or not. This function tries to establish a connection with the sd card. If it cannot get a connection, it will make the flag low.

`void tempcheck();` // This checks whether all the 3 temperature sensor is connected or not. This checks by seeing how many sensors are connected to it. **NOTE:** Because, we want it to adapt it to new sensors that we connect in the future, this will set the flags of all three temperature sensor to low even if it doesn't detect even on of the sensor. All the flags of temperature sensor will only be high when all three sensors are present.

`int currentCheck(int);` // This checks the presence of current sensor. This just checks whether the current sensor is connected or not. It does this by checking the voltage across the Vout pin of the current sensor and gnd pin. If you have read the current sensor section, you can remember that when we power the sensor up and there is no current flowing, the voltage across that will be half of the whatever the power supply is able to provide. Now, if it is not connected at all, there won't be any power at all between the Vout pin and gnd. This will also set the reference value for the current sensor. Making it changeable for any voltage type used

`int voltageCheck(int);` // This checks if the voltage sensed is not zero. If it is zero, which means there is no voltage sensor connected. So, it is better to connect the voltage sensing components and then power up the board. Even if you dont, its fine. The program will adapt from the loop and it will change the flag as soon as it detects the presence of voltage between the voltage sensors

`int irrCheck(int);` // I never had the opportunity to see the sensor, but according to the people who used it, it outputs voltage. So, if it is not connected, the sensor will output zero volts.

`printAddress();` // This is to print the address of the temperature sensor. This is just here for debugging purposes. (:

PROGRAM FLOW: A VIEW OF HOW IT'S DONE (:

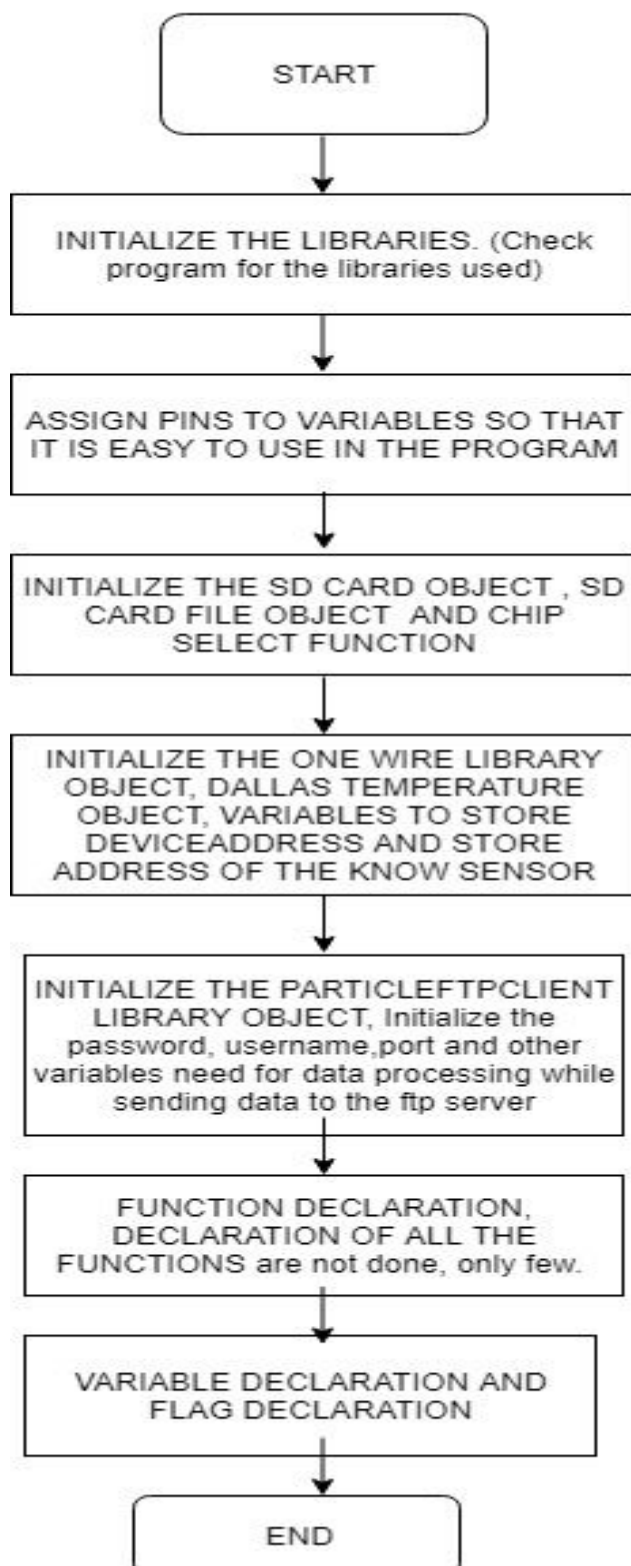
This will be easier to explain using a flow chart, there are two main functions. One that runs as soon as the particle board is switched on. And other is that runs every other time except when switching it on, (i.e) It is the part where actual program runs and loops through. These two are the official loop. There is one called preprocessor. It will before the first function. This is where we initialize everything

that is to be used. The flowchart below are in order of the times there are executed when the particle board is switched on.

PREPROCESSOR:

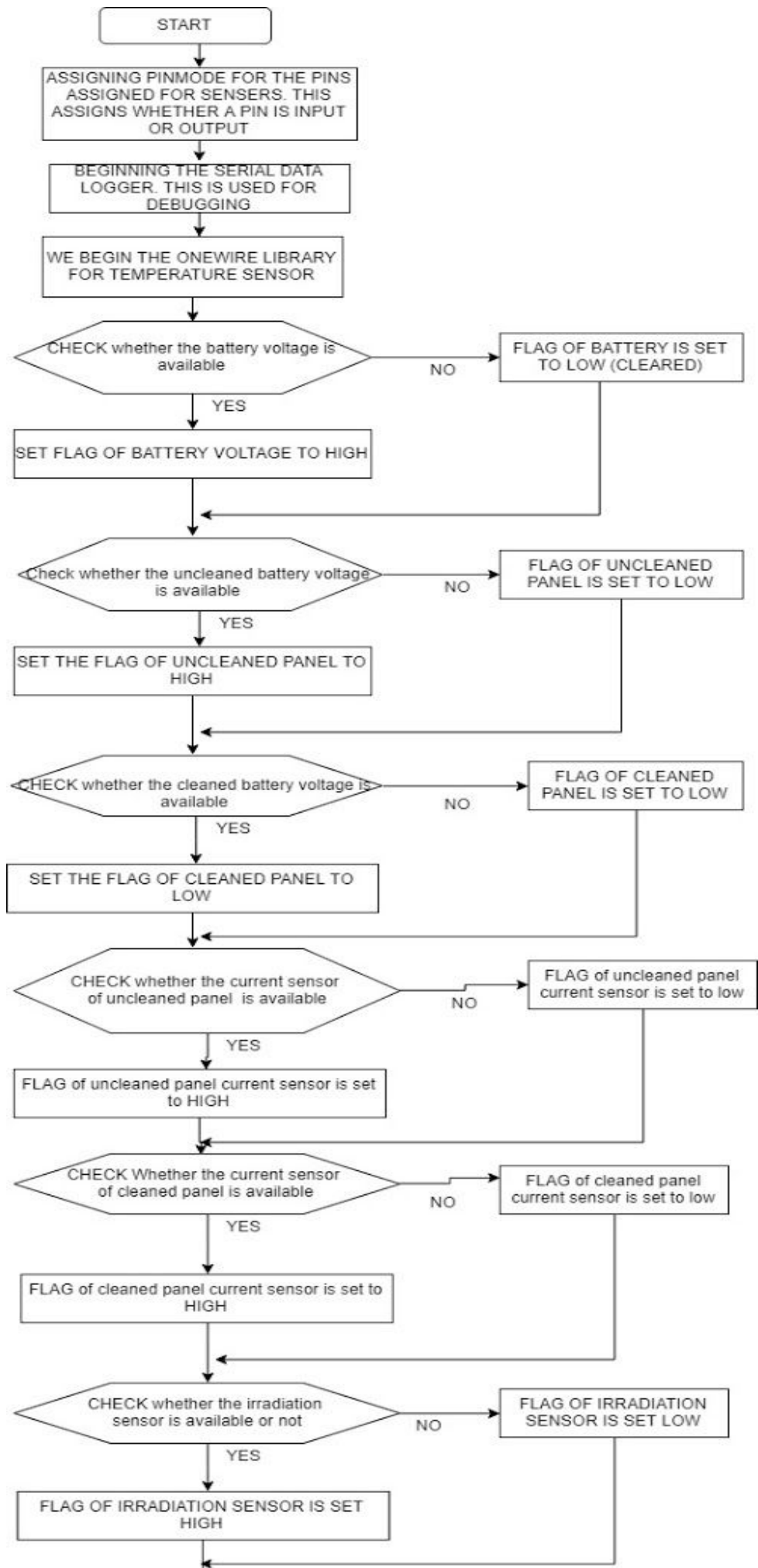
As one can see in the flowchart below, It is where most of the initialization is done. Function declarations and everything. The order of declaration is not important except for some like, initializing the `#include` at first in a mandatory. Some proper C programming language rules must be followed.

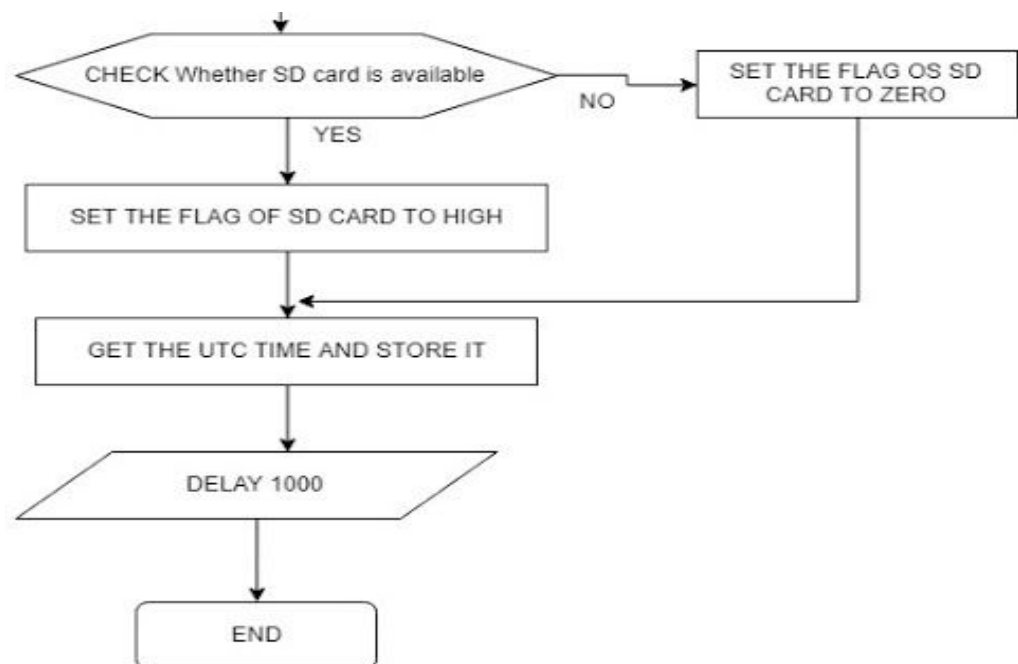
PREPROCESSOR LOOP



AFTER THE ABOVE FLOWCHART PROCESS IS EXECUTED, THE BELOW FLOW CHART IS EXECUTED. This is the void setup() function you see in the program. All the program initial settings are set here.

SETUP function flowchart

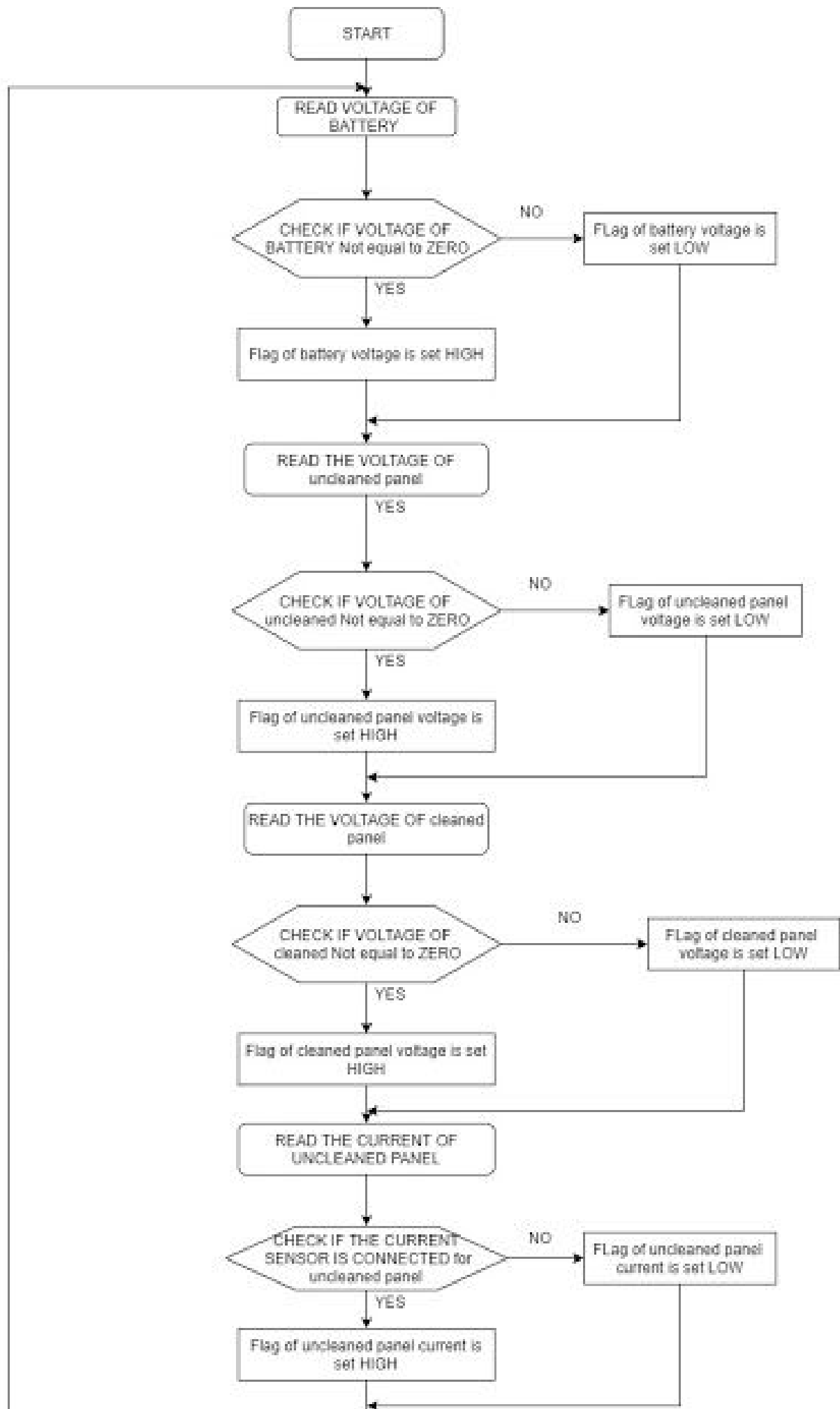


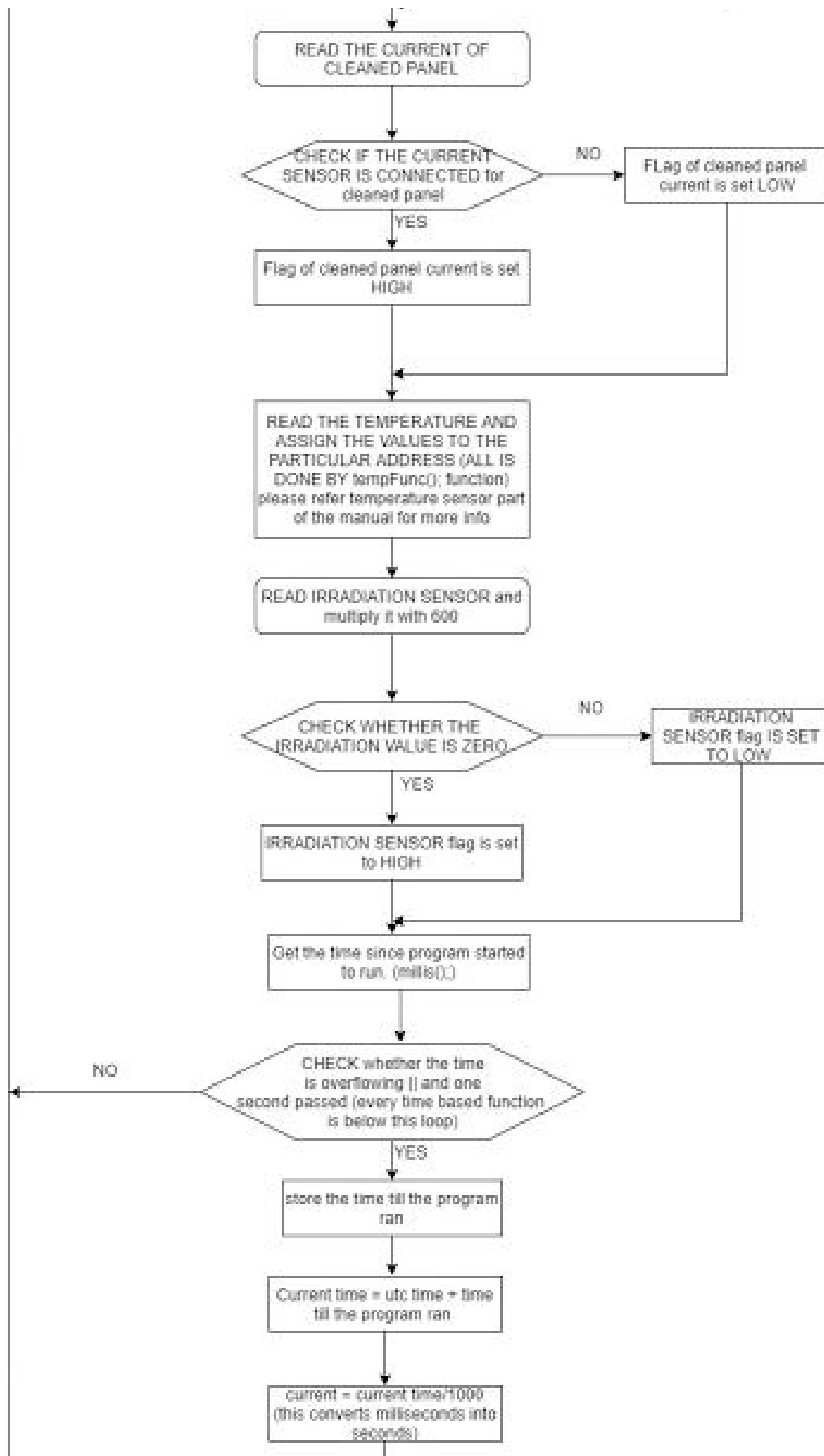


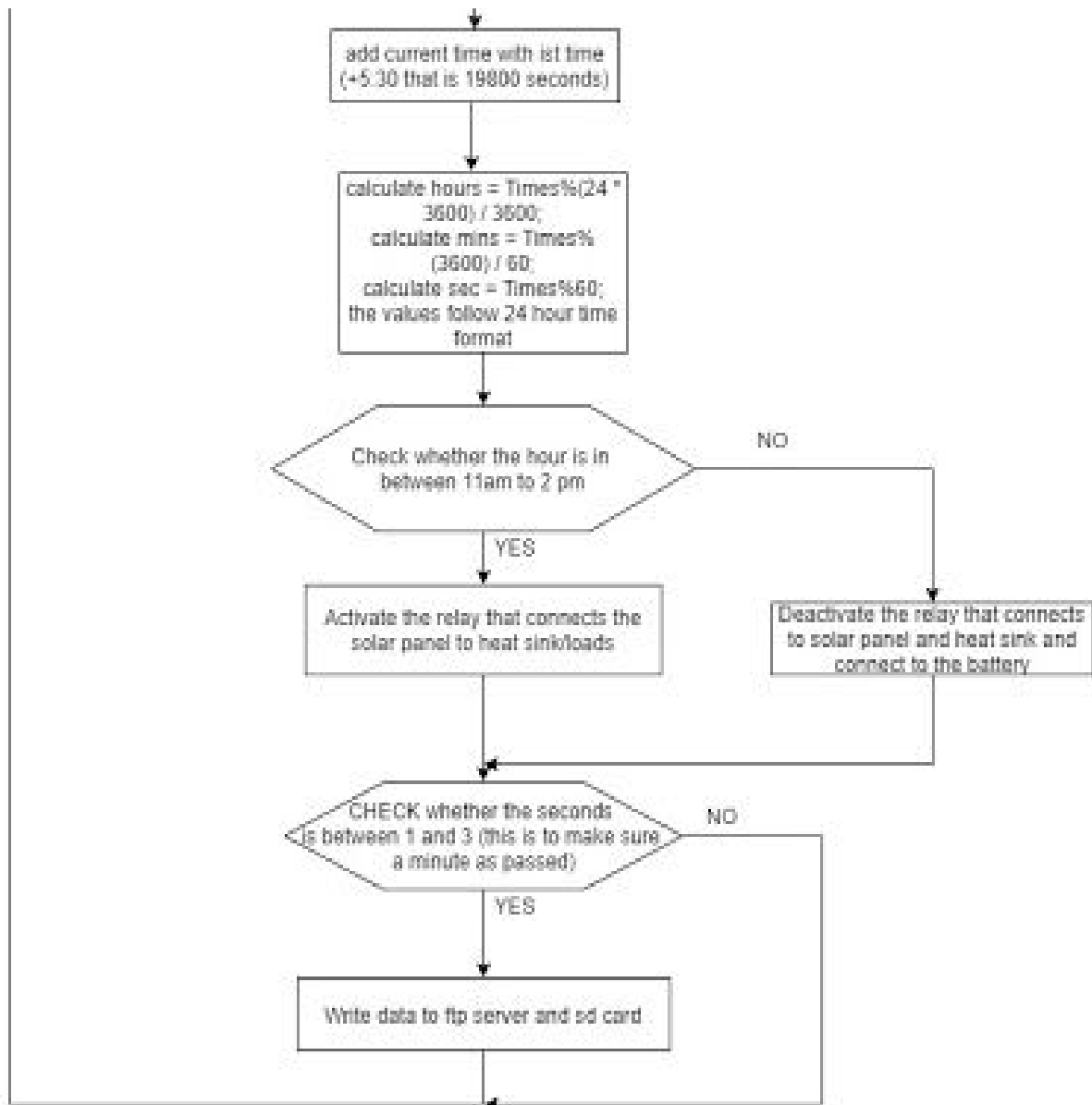
The above flowchart shows all the process done in the void setup function. Note that, the ftp server is not checked in the setup function. This is not needed as it is checked in the loop functions. The above flowchart process is executed only once when the device is first switched on.

VOID LOOP() FUNCTION:

This is the function that runs all the time except when starting. This is where the real time calculation are done, functions are called, data is gathered, etc.







This flowchart should have a proper representation of what the program does when it is work. Please note that, i have used flags to set which data to send and store in ftp server. The flags is a direct result of functionality of a particular sensor or event. **NOTE:** Please don't change any values in the following lines of code.

```
if ((now - lastTime) >= 1000) { //this is to prevent timer overflow.. it should, in theory... the timer will overflow every 59days
```

```
lastTime = now; //this will make it into a loop and prevent overflow
```

```
Times = utc+(now/1000);
```

```
Times+=19800; //utc+ist
```

```
hrs = Times%(24 * 3600) / 3600; //calculate hrs from utc+ist
```

```
mins = Times%(3600) / 60; //calculate mins from utc+ist
```

```
sec = Times%60; //calculate sec from utc+ist
```

This takes care of time and any change in value will ruin the time calculating feature of the board and will not give accurate and this resulting in untimely connection with ftp server or worse, completely stopping the connection due to time mismatch. It is not that ftp server needs proper time to get connected, it is just that, the data will be sent to ftp server every one minute, changing might result in not satisfying the condition that makes the data to be sent every minute.

DEBUGGING METHOD :

The line `//#define DEBUG 1` uncomment this line and compile. Flash the firmware after compiling. This will start to display debugging codes and texts. You can use this to check the operation, operation flow, flags etc.