Swiss Institute for Speleology and Karst Studies

Datalogger Guide: Usage, Setup, and Operational Insights

by

Nicolas Schmid

A report made during my civil service at ISSKA, to explain my work and make it easier for other employees or civilists to use and modify the dataloggers.

August 25, 2023

Contents

1 Datalogger General Overview

ISSKA requires a variety of sensors to collect data in caves or in outdoor locations. A Datalogger is needed for this to control the sensors and store their values with a given time step. Most commercial solutions are not suitable for these environments due to factors such as mud, dust, high air humidity, difficult accessibility, lack of internet connection and absence of external power sources. Some commercial solutions exist, but they're expensive and can't connect with just any sensor.

The institute manufactures its own dataloggers, which makes it possible to adapt these to almost any sensor. It has the advantage of being cheap and flexible, but it is not a plug-and-play solution. It requires basic knowledge about electronics and Arduino programming, which are explained in this report.

1.1 The TinyPico Microcontroller

The dataloggers utilize a TinyPico microcontroller, equipped with a cost-effective yet powerful ESP32 microprocessor. The ESP32 offers superior performance compared to similar microcontrollers like Arduino, thanks to its enhanced memory and higher clock frequency. This increased power is sufficient for any datalogger application.

The TinyPico can be placed in deep sleep mode, consuming minimal current (around 10 µA). This feature enables dataloggers to operate in caves for extended periods, utilizing smaller and more portable batteries. The microcontroller controls various datalogger components, including the screen, external clock, SD card and sensors. Programming the microcontroller with Arduino language is straightforward; connect its USB port to a computer and upload code using the Arduino IDE. Refer to Section 2.1 for details on programming a TinyPico using the Arduino IDE.

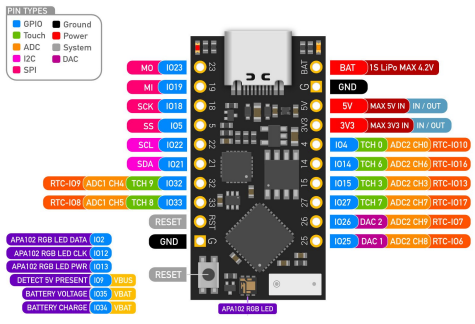


Figure 1: TinyPico microcontroller, featuring labeled ports. The microcontroller uses only 10 µA in deep sleep mode. [1]

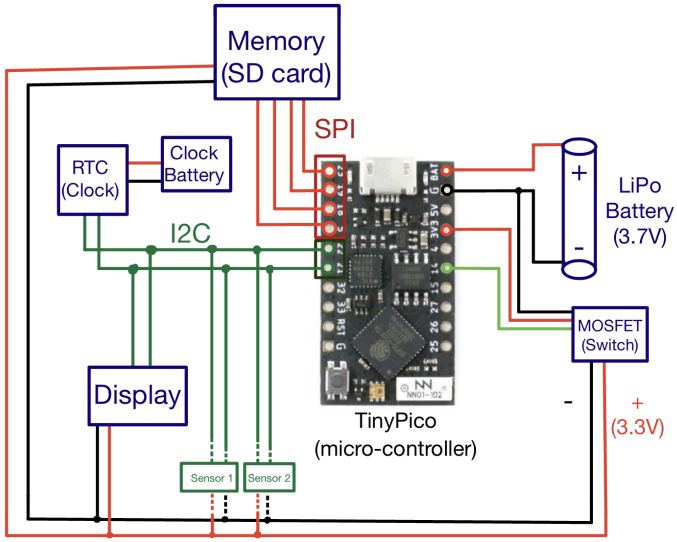


Figure 2: Schematic of a datalogger, illustrating its main components and connections.

1.2 Communication Protocols

Figure 2 illustrates how the TinyPico connects to other datalogger components. Diverse communication protocols are used for data exchange between components. These protocols vary in terms of cable usage (number of wires), maximum data transmission distance, transmission rate, device count, complexity, etc.

3 different communication protocols are used in our dataloggers:

- **I²C :** The datalogger employs I²C to communicate with most sensors, the external clock, and the display. This protocol requires only two wires and supports communication with up to 128 devices on the same bus. This is possible because each device has a unique 7-bit address stored in its memory. Before sending or receiving data from a device, the address of a device is sent on the I²C bus to inform all the devices connected to the bus if we communicate with them or with another device. Hardware implementation is straightforward; connect the 2 pins from the TinyPico (master device) called SCL and SDA to the SCL and SDA wires from the slave devices.

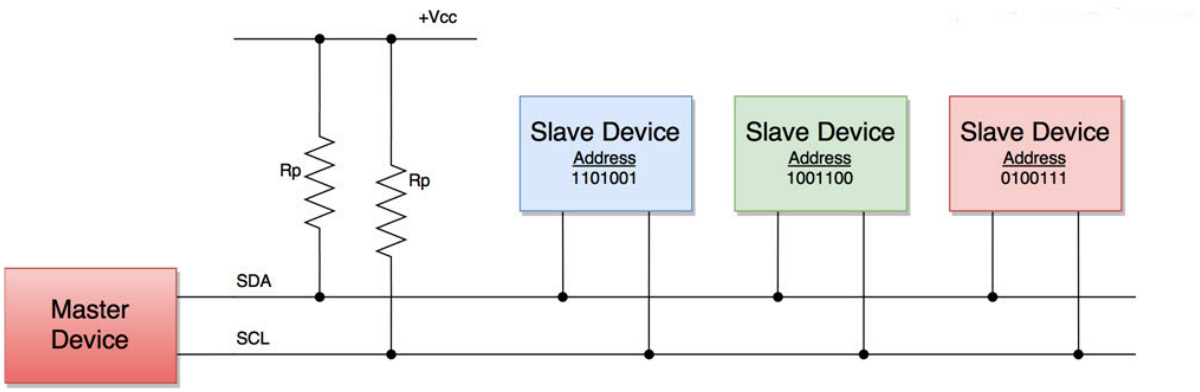


Figure 3: I²C communication protocol.

This protocol was originally designed to communicate over short distances of a few centimeters, but it actually also works over longer distances (up to 50 m in my experience) but the longer the distance is, the lower the communication frequency must be. The communication frequency is usually 100 kHz for distances of a few centimeters, but it must be reduced to about 1000 Hz for a communication cable length of 50 m.

To ensure proper communication, a pull-up resistor R_p should be inserted between the SCL/SDA lines and the 3.3 V power source as depicted in Figure 3. These pull-up resistors allow the voltage to rise faster in the communication cables. The appropriate value of R_p hinges on the total bus capacitance C_{bus} and the desired communication frequency. While a wide range of resistance values is effective, it's worth noting that lower R_p values enhance communication at the expense of increased power consumption. For extended communication cables, consider reducing the overall R_p value by adding pull-up resistors in parallel. Here is a formula for the range of R_p [2]:

$$R_p\text{min} = \frac{V_{cc}-0.4\text{V}}{3\text{mA}} = 1k\Omega \qquad R_p\text{max} = \frac{1000\text{ns}}{C_{\text{bus}}}$$

But in general, adding a 5 kΩ to 10 kΩ resistance in parallel for each sensor seems like a reasonable compromise for most users. For additional insights into selecting suitable pull-up resistor values for I^2C communication, you can refer to this stack-exchange thread.

- **SPI :** This communication protocol is used to communicate between the TinyPico and the SD card slot. This protocol employs 4 wires and supports multiple devices on the same bus, but unlike I^2C one extra wire is needed for each new sensor (see Figure 4). This makes the connections between devices more complicated since we need 4 wires instead of 2 with I^2C, but this protocol can communicate faster than I^2C and the hardware of a SPI device is simpler, so this is why it is used to communicate with the SD card.

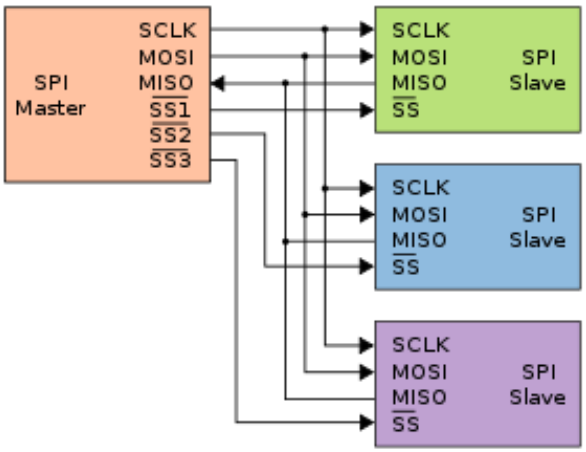


Figure 4: SPI communication protocol.

- **Serial :** Serial protocol is used for code upload and communication with some sensors. It is simple, but rather slow. This protocol also only requires 2 wires, but only 1 device can be connected to these 2 wires. This can be problematic if we want more than one sensor communicate with this protocol. For multiple devices, a serial multiplexer or I^2C-based sensors are preferred.

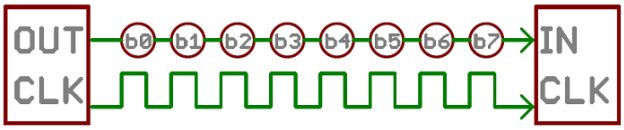


Figure 5: Serial communication protocol.

1.3 The Memory

A microSD card is used to store values measured by the sensors. For most applications, 64MB of storage is sufficient, but because microSD cards have become very affordable over time, we sometimes use ones with up to 8GB of storage.

Sensor values are stored in a file called *data.csv* with the format shown in Figure 6. The file always includes 3 columns: *ID*, *DateTime*, and *Vbatt*. Additional columns depend on the connected sensors. The file is created automatically on the microSD card when the datalogger boots up if the file doesn't already exist.

```
ID;DateTime;VBatt;temperature;pressure;box_humidity;SFMflow;
1;2023/06/15 14:00:00;4.05;28.390;906.16;34.55;0.01666667;
2;2023/06/15 14:05:00;4.06;28.410;906.17;34.12;-0.04166667;
3;2023/06/15 14:10:00;4.05;28.530;906.16;33.75;-0.03333334;
4;2023/06/15 14:15:00;4.06;28.520;906.16;33.79;-0.04166667;
```

Figure 6: Example of the data.csv file content.

Another file called *conf.txt* contains configuration settings for the datalogger. An example configuration file is shown in Figure 7. This file allows users to adjust the time step between measurements and set the correct time on the external clock without having to make changes in the code.

```
300; \\time step in seconds between measurements
1; \\boolean value to set the RTC when booting (1 or 0)
2023/08/25 14:05:00; \\time to set the RTC if the set value is 1
```

Figure 7: Example of the conf.txt file content.

1.4 Real Time Clock

The real time clock (RTC) keeps track of time and wakes up the datalogger at set intervals. Our dataloggers use the DS3231 RTC. This RTC has an accuracy of 2ppm, meaning it only drifts by 2 seconds every 1 million seconds, which is about 1 minute per year [3].

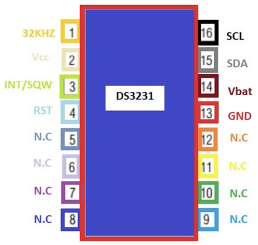


Figure 8: The DS3231 RTC pin layout.

This RTC communicates via I^2C with the TinyPico, hence the SCL and SDA pins of the DS3231 are connected to the TinyPico. The SQW pin is also connected to the TinyPico, because it is used to wake-up the TinyPico from a deep sleep mode. Before entering deep sleep mode, the TinyPico sends via I^2C the information to the clock to set an alarm at a given time in the future. When the time comes, the alarm triggers the SQW pin, waking up the TinyPico.

The RTC only keeps track of time when powered, so we have a small battery for it that's always connected. To set the RTC, put a battery in and write a future time on the *conf.txt* file along with "1" for the setRTC boolean value. Put the microSD card in, power on, and press the TinyPico's reset button when the current time matches the one on *conf.txt*. Then, set the boolean value to "0" in *conf.txt* to avoid setting the wrong time next time you power the datalogger.

1.5 Power Management

To minimise the power consumption of the datalogger, the TinyPico is put into deep sleep mode where it consumes almost no current. However, the sensors need to be powered off too. A MOSFET driver shown in Figure 9 acts as a switch for the sensors' power.

The GND pin is connected to the GND of the TinyPico, and VIN to its 3.3 V pin. INSENS is connected to the pin 14 of the TinyPico, hence the microcontrollers controls it. All sensors, display and SD card reader are connected to both the 3.3 V source and to GSENS.

When INSENS is low (below 1.5 V), GND and GSENS disconnect, but when INSENS is high (above 2.5 V), GND and GSENS connect [4], and the power can flow to the sensors, display and SD card reader. For additional insights about the design of the MOSFET driver and choice of its component you can refer to the MOSFET driver made by Adafruit.

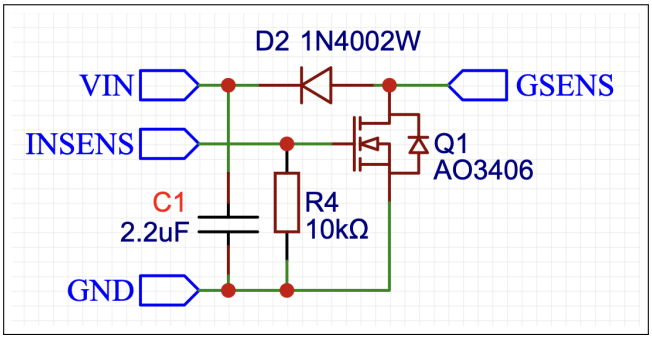


Figure 9: MOSFET driver schematics from EasyEDA.

This brings the datalogger's power consumption to under 0.2 mA of current during sleep. For instance, if we use a 3.7 V LiPo battery with 2000mAh capacity, it would last $\frac{2000mAh}{0.2mA} = 10000h$ which is about 400 days. This is the main power consumption, while the rest depends on sensor types. If higher power sensors are needed or longer runtime desired, a battery with greater capacity can be used.

1.6 Multiplexer

We equip all our dataloggers with an I^2C multiplexer, even though it isn't drawn in Figure 2 to enhance the readability of the diagram. As depicted in Figure 10, the multiplexer is directly connected to the SCL and SDA lines, and its power is controlled by the MOSFET to avoid using power during deep sleep.

The multiplexer serves as a switch among eight separate I^2C lines. For example, if there are multiple I^2C devices with identical addresses, they can be connected to different channels on the multiplexer. To collect data from the sensors, we instruct the multiplexer to select the appropriate line, measure one sensor, switch to a different line, and then measure the next sensor, and so on.

In some instances, I^2C devices may interfere with each other, particularly when connected through

long cables. Using separate channels on the multiplexer effectively mitigates this issue. Given that we have eight channels available, it's a good practice to always use distinct multiplexer channels for different I^2C devices.

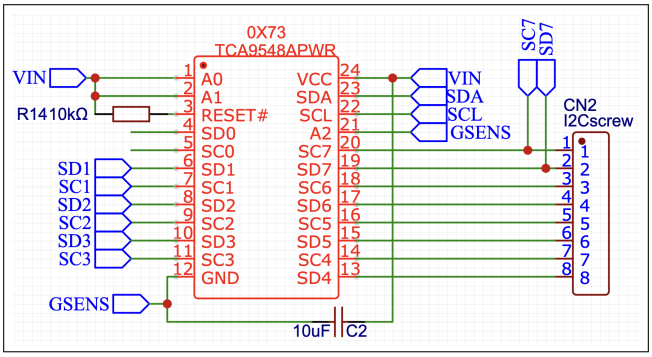


Figure 10: MOSFET driver schematic created using EasyEDA.

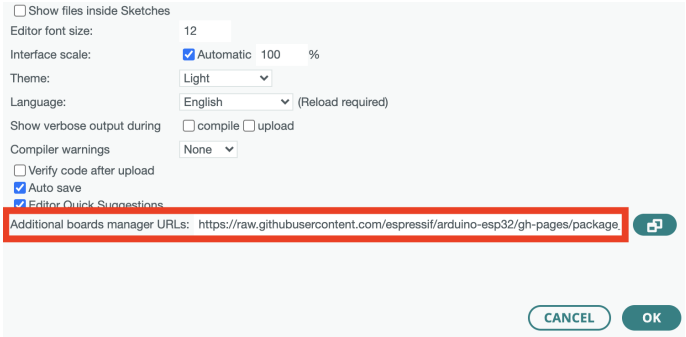
We employ the TCA9548APWR 1-to-8 multiplexer, which supports eight selectable I^2C addresses, ranging from 0x70 to 0x77. The addresses are determined by the states of pins A0, A1, and A2, which can either be connected to VCC (logical 1) or to ground (logical 0). The last digit of the I^2C address is derived from the binary value represented by A2-A1-A0. In our setup, as shown in Figure 10, A0 and A1 are set to logical 1, while A2 is set to logical 0. This configuration yields the last part of the I^2C address as $011 \rightarrow 3$ in hexadecimal notation. Consequently, the I^2C address for the multiplexer on our PCB is 0x73. For many commercially available boards that incorporate this multiplexer, the default address is set to 0x70.

2 Programming of Dataloggers

2.1 Setting up the Arduino IDE

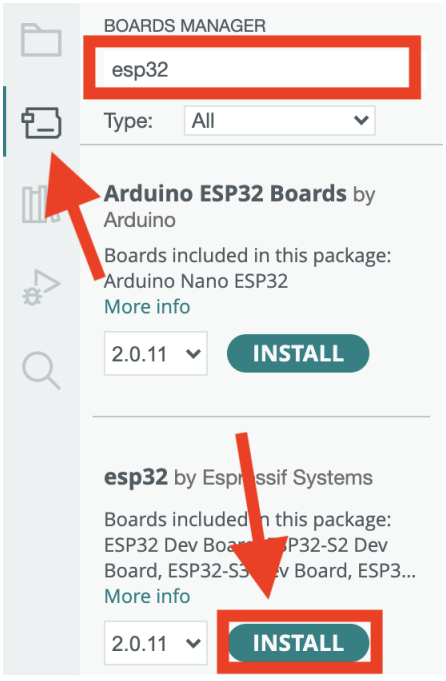
The first step is to download the Arduino IDE. I recommend using Arduino IDE 2.x as it offers a superior interface and an improved serial plotter for easier debugging. If you find my explanations unclear, consider following this tutorial.

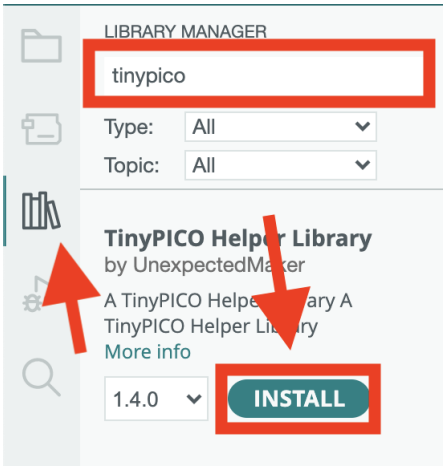
To work with an ESP32, you'll need to add the TinyPico board in the Board Manager. For this, first go to **File** → **Preferences** for Windows or to **Arduino IDE** → **Preferences** for macOS. At the bottom of the Preferences window, paste the following URL into the **Additional Boards Manager URLs** field and click **OK**:



https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json

Once this is done, click the **Board Manager** icon on the left-hand side of your window. Search for **esp32** and select *esp32 by Espressif Systems* by clicking on its **install** button. You'll also need to install additional libraries. To do this, navigate to the **Library Manager**, search for the required libraries (e.g., TinyPico Library), and click **Install**. A list of the necessary libraries is provided in Table 1.





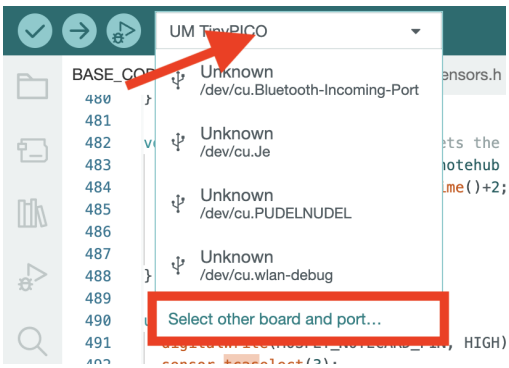
Library name	Developer name	Application of the library
TinyPICO Helper Library	UnexpectedMaker	This library allows us to simply get the voltage of the TinyPico and control its RGB led
RTClib	Adafruit	Library which facilitates communication with the DS3231 RTC
U8g2	oliver	Library used to communicate with a variety of displays including the U8X8 SSD1306 used in our dataloggers
BMP581 Arduino Library	Sparkfun	The BMP581 is a high precision pressure and temperature sensor which is by default on our dataloggers' PCB
SHT31	Rob Tillaard	This library works with the SHT35 sensor, which is by default on our dataloggers to measure humidity and temperature

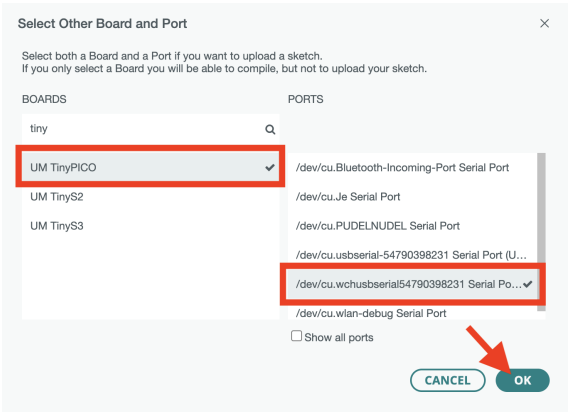
Table 1: Arduino libraries needed to compile the datalogger base code.

Once you've installed all the required libraries, you should be able to compile the base code for the dataloggers. The base code serves as the skeletal structure for any datalogger; only minor modifications are needed when adding a new sensor. For any new sensors, additional libraries may need to be installed. If you can't find the library in the Library Manager, refer to this tutorial to install the library manually.

The TinyPico V3 utilizes a CH9102F to connect the USB to the PICO-D4 ESP microprocessor, so ensure you have the appropriate driver installed [5]. You can grab the latest drivers from the WCH website. In my case I installed the WCH34xVCPDriver for macOS.

To upload your first program to a datalogger, download the base code for dataloggers from GitHub and open it in the Arduino IDE. Make sure all three files reside in a folder named after the Arduino code file, minus the ".ino" extension. Connect the TinyPico to your computer via USB, and then select the appropriate **port** and **board** from the IDE.





If the driver is successfully installed (you may need a reboot your computer) the TinyPico should appear in the Arduino IDE and as a device on your computer in the following formats:

- macOS (TinyPICO V2): **/dev/tty.SLAB_USBtoUART**
- macOS (TinyPICO V3): **/dev/tty.wchusbserialXXX** where XXX is the index of the USB device
- Linux (TinyPICO V2): **/dev/ttyUSBx** where x is the index of the USB device
- Linux (TinyPICO V3): **/dev/ttyACM0**
- Windows: **COMn** where n is the port number assigned by Windows

Once the port and board are selected, click on the arrow on the top left of your window to **upload** the code. If the upload fails, consider trying a different USB cable; some cables are designed only for power transfer and not for data transfer.

Once the code has been uploaded successfully, you should see output similar to what is shown in Figure 11.

```
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 633.8 kbit/s)...
Hash of data verified.
Compressed 420816 bytes to 246311...
Writing at 0x00010000... (6 %)
Writing at 0x0001b250... (12 %)
Writing at 0x00028e83... (18 %)
Writing at 0x0002e2c1... (25 %)
Writing at 0x00033f1b... (31 %)
Writing at 0x000394df... (37 %)
Writing at 0x0003f125... (43 %)
Writing at 0x000447d4... (50 %)
Writing at 0x00049f88... (56 %)
Writing at 0x0004ee93... (62 %)
Writing at 0x0005439a... (68 %)
Writing at 0x0005a9a4... (75 %)
Writing at 0x000639e4... (81 %)
Writing at 0x0006b944... (87 %)
Writing at 0x00070dbe... (93 %)
Writing at 0x000768d0... (100 %)
Wrote 420816 bytes (246311 compressed) at 0x00010000 in 4.2 seconds (effective 801.8 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

Figure 11: Upload output from the Arduino IDE.

To see what the dataloggers is sending to the computer via serial communication, open the serial monitor by clicking on the icon that resembles a magnifying glass in the top-right corner. Then, select a baud rate of 115200 on the bottom-right corner as depicted in Figure 12.

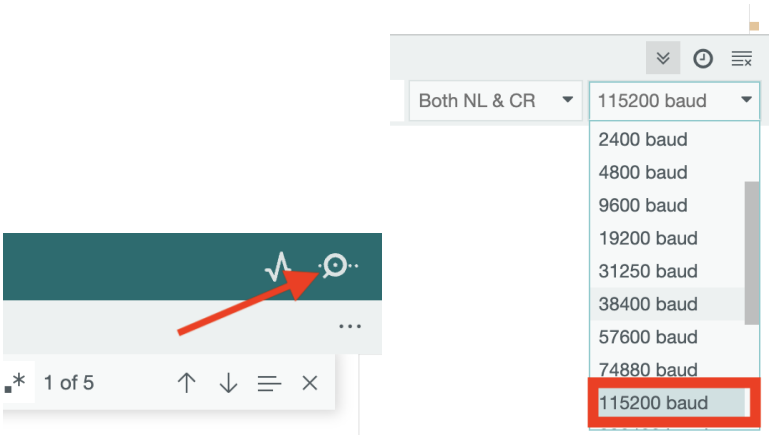


Figure 12: The selected baud rate must match that of the microcontroler.

2.2 Base Code Walk-through

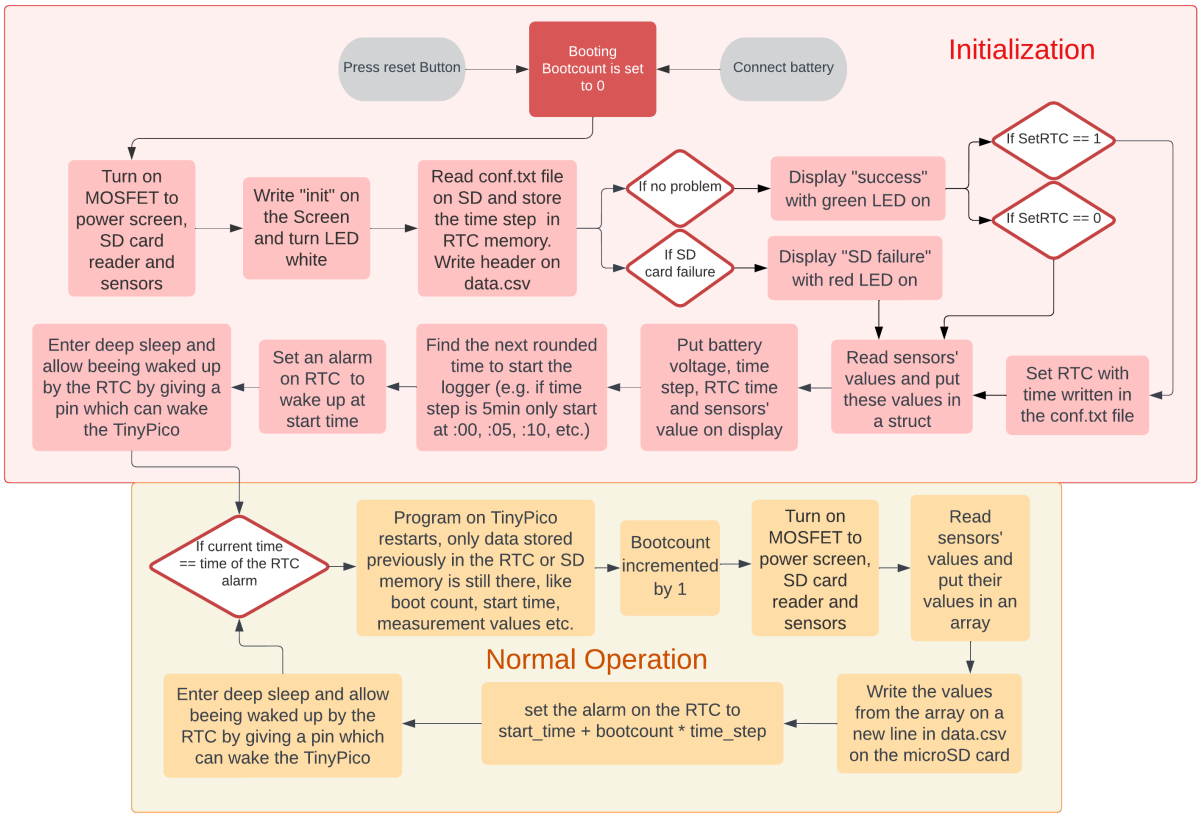


Figure 13: Datalogger base code diagram, created with Lucidchart.

Figure 13 explains how the base code of a datalogger works. When the datalogger is started, it first enters an initialization phase, where the boot count is set to 0, the header is written on the `data.csv` file and metrics such as battery voltage, RTC time and sensors' values are displayed. The datalogger is then put into deep sleep until a rounded time. Once awake, it enters a loop where it periodically wakes up, takes sensor readings, saves them to the SD card, and returns to deep sleep.

If you are familiar with Arduino coding, you might know that usually a program is made of a `setup()` function, which runs once at the beginning, followed by a `loop()` function that runs indefinitely, as shown in Figure 15. However in this datalogger program, the `loop()` function is not utilized. This is because every time the microcontroler wakes up from deep sleep, the `setup()` function is run again, so actually all the code is in this `setup()` function, and the device returns to deep sleep at the end of this function, as illustrated in Figure 14.

In the following sections, we will discuss each part of the code in greater detail. The complete code is available on Github.

```ino
//variables stored in RTC memory
RTC_DATA_ATTR bool first_time=true;
RTC_DATA_ATTR int bootcount = 0;

void setup(){
    if(first_time){
        first_time=false;
        //do initialization once
        //...
        start_time = get_next_rounded_time();
        enter_deep_sleep(start_time);
    }
    else{
        bootcount++;
        //do normal operation in loop
        //...
        wakeup_time = start_time + bootcount*timestep;
        enter_deep_sleep(wakeup_time);
    }
}

void loop(){
    // do nothing
}
```

Figure 14: Arduino code structure with deep sleep.

```ino
void setup(){
    //do initialization once
    //...
}

void loop(){
    //do normal operation in loop
    //...
}
```

Figure 15: Typical Arduino code.

2.2.1 Powering the Devices

The first step is to power the various devices, which can be accomplished turning the sensors' MOSFET on. The MOSFET is generally connected to pin 14 of the TinyPico.

```ino
#define MOSFET_SENSORS_PIN 14 //pin to control the power of sensors, SD card reader and screen

void power_external_devices(){
  pinMode(MOSFET_SENSORS_PIN, OUTPUT);
  digitalWrite(MOSFET_SENSORS_PIN, HIGH);
}
```

2.2.2 Reading and Writing to the SD Card

The library *SD.h* is used to control the SD card reader. To use it, we must first proceed with an initialization. For this we have say which pin of the TinyPico is connected to the CS pin of the SD card reader.

```
#include <SD.h>
#define SD_CS_PIN 5 //Define CS pin for the SD card module
//...
if (!SD.begin(SD_CS_PIN)) {
    Serial.println("SD card initialization failed!");
}
else{
    Serial.println("SD card initialization failed!");
    //do something with the card
    //...
}
```

Figure 16: Initialization of the SD card reader.

We then can read values from the *conf.txt* file. Each value in this file are followed immediately by a ";" and preceeded by a "\n" (newline character). We use this information to parse the values. Obviously if there is no *contf.txt* file on the SD card, or if the content of this file doesn't respect the two above mentioned rules, this will result in an error.

```
#define CONFIG_FILENAME "/conf.txt"
bool setRTC;
RTC_DATA_ATTR int deep_sleep_time; //this value is stored in RTC memory to be able to access
//it after the TinyPico reboots
//...
File configFile = SD.open(CONFIG_FILENAME);
if (configFile.available()) {
    deep_sleep_time = configFile.readStringUntil(';').toInt();
    while (configFile.peek() != '\n'){ //if there are any black space in the config file
        configFile.seek(configFile.position() + 1);
    }
    configFile.seek(configFile.position() + 1); //to remove the \n character itself
    String str_SetRTC = configFile.readStringUntil(';');
    if (str_SetRTC=="0") SetRTC = false;
    else if (str_SetRTC=="1") SetRTC = true;
    //read setup time on conf.txt file
    //...
}
```

Figure 17: Reading and storing values from the *conf.txt* file.

Storing values on the SD card is straightforward, as shown in Figure 18. If the *data.csv* file doesn't already exist, it is automatically created. To write a header to the SD card, you can use the exact same code but replace the data message string with the header text.

```
#define DATA_FILENAME "/data.csv"
//...
String data_message = get_sensors_value_string();
File dataFile = SD.open(DATA_FILENAME, FILE_APPEND);
dataFile.print(data_message);
dataFile.println(); //go to a newline for each new data message
dataFile.close();
```

Figure 18: Storing sensor values onto the *data.csv* file.

2.2.3 Displaying Data

The U8x8 I^{2}C display is used to present specific data. Therefore, it's essential to initialize the I^{2}C communication protocol before using the display.This initialization is done using the *Wire.h* library. Figure 19 demonstrates how to initialize the U8x8 display and how to show values such as battery voltage, time step, and RTC (Real-Time Clock) time. The *TinyPICO.h* library is utilized to measure the battery voltage of the TinyPico.

```
#include <U8x8lib.h>
#include <Wire.h>
#include <TinyPICO.h>
U8X8_SSD1306_128X64_NONAME_HW_I2C u8x8(/* reset=*/ U8X8_PIN_NONE);//initialise the OLED display
TinyPICO tp = TinyPICO(); //used to measure the battery voltage and control RGB LED
//...
Wire.begin(); // initialize I2C bus
Wire.setClock(50000); // set I2C clock frequency to 50kHz
//...
u8x8.begin(); //initialize the display
u8x8.setBusClock(50000); //This frequency must be the same as the I2C frequency
u8x8.setFont(u8x8_font_amstrad_cpc_extended_r); //choose font
// Display battery voltage
u8x8.setCursor(0, 0);
u8x8.print("Batt: ");
u8x8.print(tp.GetBatteryVoltage(););
u8x8.print("V");
//Display current time on RTC
u8x8.setCursor(0, 2);
u8x8.print(readRTC());
//Display value of timestep read on conf.txt
u8x8.setCursor(0, 4);
u8x8.print("timestep: ");
u8x8.print(deep_sleep_time);
u8x8.print("s");
delay(8000); //wait 8 seconds so the user is able to read the screen
u8x8.clear(); //remove what is written on the display
```

Figure 19: Displaying value on the U8x8 I^{2}C display.

2.2.4 Setting and Reading the RTC

The RTC also communicates with I^2C. The communication protocol must be initialized before communicating with the clock. Communication with the RTC is facilitated through the *RTClib.h* library. To employ this library, an instance of the RTC_DS3231 class should be declared, followed by calling the `begin()` method.

```
#include <RTClib.h>
RTC_DS3231 rtc;
//...
rtc.begin();
rtc.disable32K(); // the default 32kHz output is disabled as it is not required
```

Figure 20: Initialization of RTC

The `now()` method retrieves the current time in DateTime format. DateTime format presents the advantage that we can add times together and easely transform them in human readable formats. Methods like `DateTime::second()`, `DateTime::minute()` and so on, can be used to obtain the seconds, minutes etc. of a given time. You can also get the unixtime (the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970) by calling the `now().unixtime()` method.

```
DateTime now = rtc.now(); //get the current time in DateTime format
String time_str = String(now.day()) + "/" + String(now.month()) + " " + String(now.hour())+ ":"
+ String(now.minute()) + ":" + String(now.second());
Serial.print("Current time: ");
Serial.println(time_str);
```

Figure 21: Reading RTC time.

When we first put the battery in a DS3231 clock, the time on the RTC defaults to January 2000. To update the RTC with the correct time, the `adjust()` method should be invoked.

```
int start_time_programm;
void setup(){
    start_time_programm = micros(); //get the start time of the programm in microseconds
    //...
    setRTC();
    //...
}

void setRTC(){
// write the precise time at which you will reboot the datalogger
DateTime time_for_booting = DateTime(year,month,day,hour,minute,second);
float adjustement_time = micros();
//compensate the delay between the booting time and the time at which we set the RTC.
rtc.adjust(SD_time+2+(adjustement_time-start_time_programm)/1000000);
Serial.print("rtc set to: ");
Serial.println(rtc.now().timestamp(DateTime::TIMESTAMP_FULL)); //print time on RTC
}
```

Figure 22: Set RTC time

We can put up to 2 alarms on the RTC. We only use alarm1 in our case. Since it might be that a previous user has put an alarm on the clock, it is prudent to clear any existing alarms before setting a new one. In our case, since we don't use alarm2 we can even disable it. To set the alarm1, we use the `setAlarm1()` method, with the argument `DS3231_A1_Hour` which means that the alarm1 is triggered when hours, minutes, and seconds match with the time given in the other argument. The TinyPico pin connected to the SQW pin of the RTC (typically pin 15 for our datalogger) needs to be configured in pull-up mode to detect the clock's trigger signal.

```
rtc.clearAlarm(1); //to clear a potentially previously set alarm set on alarm1
rtc.disableAlarm(2); //disable alarm2 to ensure that no previously set alarm causes any problem
rtc.setAlarm1(start_time+bootCount*sleeping_time,DS3231_A1_Hour); //set an alarm on alarm1
pinMode(GPIO_NUM_15, INPUT_PULLUP); //set the pin 15 of the TinyPico in pullup mode
```

Figure 23: Set alarm on RTC.

For additional information regarding the RTC library, please refer to [6].

2.2.5 Deep Sleep Mode

To minimize power consumption, the ESP32 is put into deep sleep mode. Before doing so, any source of power drain, such as the MOSFET and the LED on the TinyPico, must be deactivated. The RTC alarm is then set, as described in Section 2.2.4. We must specify the wake-up trigger (which pin of the TinyPico is connected to the RTC alarm to wake it up) and then call a built-in function which puts it into deep sleep.

```
void deep_sleep_mode(int sleeping_time){
    //turn off all things that might consume some power
    digitalWrite(MOSFET_SENSORS_PIN, LOW);
    tp.DotStar_SetPower(false); //LED of the TinyPico
    rtc.writeSqwPinMode(DS3231_OFF);
    //set alarm on RTC
    rtc.clearAlarm(1); //to clear a potentially previously set alarm set on alarm1
    rtc.disableAlarm(2); //to ensure that no previously set alarm causes any problem
    rtc.setAlarm1(start_time+bootCount_since_change*sleeping_time,DS3231_A1_Hour);
    pinMode(GPIO_NUM_15, INPUT_PULLUP);
    //specify wakeup trigger and enter deep sleep
    esp_sleep_enable_ext0_wakeup(GPIO_NUM_15, 0); //2nd param. is logical level of trigger signal
    esp_deep_sleep_start();
}
```

Figure 24: Putting the datalogger into deep sleep.

2.2.6 Compute Starting Time

At the end of the boot sequence, the datalogger calculates a starting time for data acquisition. For ease of analysis, it's often beneficial to align measurement times to rounded intervals like XX:00, XX:10, XX:30, etc. The computation of the starting time assumes that the time step defined is either less than 60 seconds or a multiple of 60; otherwise, unexpected behavior may occur.

If you choose less than 60 seconds, the datalogger will simply start aquisition the next time that the seconds are 0. If you choose a multiple of 60 (e.g. 300) the datalogger will start the next time that the minutes are a multiple of the time step chosen. For example if the time step is 300s (5 minutes) and you start the datalogger at 10:21:43, the datalogger will only start its measurements at 10:25:00.

```
void get_next_rounded_time(){
    DateTime current_time=rtc.now(); //get current time
    //compute the number of seconds left until the next rounded minute (where seconds is :00)
    int seconds_remaining = 60-current_time.second()%60;
    DateTime time_seconds_rounded_up = current_time+seconds_remaining;
    if(time_step<60){
        start_time = time_seconds_rounded_up;
    }
    else{ //time_step>=60
        int rounded_min = time_step/60; //gives the time step in minute
        if(time_seconds_rounded_up.minute()%rounded_min==0){
            start_time = time_seconds_rounded_up; //minute already multiple of time_step
        }
        else{
            //compute the number of minutes until the next rounded time (in minutes)
            int minutes_remaining = rounded_min - time_seconds_rounded_up.minute()%rounded_min;
            start_time = (time_seconds_rounded_up + 60*minutes_remaining);
        }
    }
    //store start time in RTC
    //...
}
```

Figure 25: Calculating the starting time for data acquisition.

2.2.7 Reading Sensor Values

The code for acquiring sensor data is encapsulated in an external C++ file named `Sensors.cpp`. Only the Sensors.cpp file should be modified when writing the code for a new sensor.

To import this external file, the C++ file must be in the same folder, and you must import its header file in the Arduino code. You must also create an instance of the Sensor class in the Arduino file to use it, as demonstrated in Figure 26.

```
#include "Sensors.h"
Sensors my_sensors = Sensors();
//...
my_sensor.measure(); // measures all sensors and stores the values in a struct sensor.mesures
my_sensor.serialPrint(); //print all the values in the sensor struct onto the serial monitor
```

Figure 26: Import C++ file in the Arduino code.

The header file `Sensor.h` is a declaration of the Sensors class. This file must be in the same folder as both the Arduino and C++ files. This files tells which methods are available in the Sensors class, but the definition of these methods is done in the C++ file. You generally will not need to modify it when adding new sensors.

```
//declaration of the sensors' class. All the methods are defined in the Sensors.cpp file
#ifndef sensors_h
#define sensors_h
class Sensors {
    public:
        Sensors(){};
        String* get_names();
        int get_nb_values();
        String getFileHeader();
        String getFileData();
        String serialPrint();
        void measure();
        void tcaselect(uint8_t i);
};
#endif
```

Figure 27: `Sensors.h` file. It contains the declarations of the methods defined in the Sensor class in the `Sensors.cpp` file. The C++ language needs this to import a class in another file.

We will now explore the implementation of various methods within the `Sensors` class.

- **Sensors::tcaselect():** This method selects the multiplexer I^2C line which must be connected. It was copied from the Adafruit website.

```
#define I2C_MUX_ADDRESS 0x73 //adress in the hardware of the PCB V3.2
//...
void Sensors::tcaselect(uint8_t i) {
    if (i > 7) return;
    Wire.beginTransmission(I2C_MUX_ADDRESS);
    Wire.write(1 << i);
    Wire.endTransmission();
}
```

Figure 28: Implementation of the `tcaselect()` method for multiplexer control.

- **Sensors::measure():** The code in Figure 29 shows how to measure the BMP581 and the SHT35 sensors, and the battery voltage. But for general I^2C sensors, the steps of the code are almost always the same :
  - Import the sensor's Arduino library.
  - Create an instance of the sensor's class.
  - Invoke the `tcaselect()` method to select the multiplexer I^2C channel connected to the sensor.
  - Initialize I^2C connection with the sensor.
  - Acquire data from the sensor using a `read()` method.
  - Store the acquired data in a `values[]` array, using a `get()` method.

There are to arrays of the same length called `names[]` and `values[]` which contain the name and values of the measurements. When storing the values in the `float values[]` array, you must be careful that they get stored at the same index as the name of the values stored in the `String names[]` array.

Delay Guidelines:

Timing between function calls can be crucial. The following suggestions are based on my empirical findings and should be considered recommendations rather than definitive solutions:

- Add an initial delay in the `measure()` method to stabilize sensor power. A 5 ms delay is often sufficient.
- After each `tcaselect()` method invocation, include a minor delay, e.g., 3 ms, to ensure the multiplexer switches correctly.
- For sensors like BMP581, a delay of over 5 ms may be required after calling the `begin()` method. Some sensors, like CO_2 sensors, require extended warm-up times, necessitating a delay of several seconds (10 s). On the other hand, for the SHT35, no additional delay is needed.
- If the cables are long, the I^2C frequency must be set to a lower frequency. In this case then the above mentioned delay should be increased by a factor similar to the factor at which the I^2C frequency was reduced.
- While longer delays increase the likelihood of successful communication, they also consume more battery power. Optimize delays for a balance between reliability and energy efficiency.

```
#include "SHT31.h"
#include "SparkFun_BMP581_Arduino_Library.h"
#define BMP581_sensor_ADRESS 0x46 //the adress is set on the PCB hardware
#define SHT35_sensor_ADRESS 0x44 //the adress is set on the PCB hardware
SHT31 sht; //create an instance of the SHT31 class
BMP581 bmp; //create an instance of the BMP581 class
//declare arrays where the names and values of the sensors are stored
String names[]={"Vbatt","tempSHT","humSHT","tempBMP","pressBMP"};
const int nb_values = sizeof(names)/sizeof(names[0]); //length of names[]
float values[nb_values];
//...
//measure all sensors'values and store them in the values arrays
void Sensors::measure() {
  delay(5); //5ms delay to ensure that sensors are properly powered

  //measure the battery voltage of the battery which powers the datalogger
  values[0]=tiny.GetBatteryVoltage(); //Vbatt

  tcaselect(1); //connect the SHT35 I2C wire with the multiplexer
  delay(3); //3ms delay for the multiplexer to switch
  sht.begin(SHT35_sensor_ADRESS);
  sht.read();
  values[1]=sht.getTemperature(); //tempSHT
  values[2]=sht.getHumidity(); //humSHT

  tcaselect(2); //connect the BMP581 I2C wire with the multiplexer
  delay(3); //3ms delay for the multiplexer to switch
  bmp.beginI2C(BMP581_sensor_ADRESS);
  delay(10); //10ms delay for communication initialization with the sensor
  bmp5_sensor_data data = {0,0};
  int8_t err = bmp.getSensorData(&data);
  values[3]=data.temperature; //tempBMP
  values[4]=data.pressure/100; //pressBMP (in millibar)
}
```

Figure 29: The measure method.

- **The methods to get the data measured :** To access the data measured with the `Sensors::measure()` method, there are several methods which output the data in different formats:
  - `Sensors::getFileHeader()` returns a String with the sensor names separated with a ";".

```
// Output format: "<sensor 1 name>;<sensor 2 name>;..."
String Sensors::getFileHeader () {
  String header_string = "";
  for(int i = 0; i< nb_values; i++){
    header_string = header_string + names[i] + ";";
  }
  return header_string;
}
```

  - `Sensors::getFileData()` returns a String with the sensor values separated with a ";".

```
// Output format: "<sensor 1 value>;<sensor 2 value>;..."
String Sensors::getFileData () {
  String datastring = "";
  for(int i = 0; i< nb_values; i++){
    datastring = datastring + String(values[i],2) + ";";
  }
  return datastring;
}
```

– `Sensors::serialPrint()` returns a String with the sensor names and values in the format "name1: value1\n name2: value2\n ..."

```
// Output format: "<name1>: <value1>\n <name2>: <value2>\n...
String Sensors::serialPrint() {
  String sensor_display_str = "";
  for(int i = 0; i< nb_values; i++){
    sensor_display_str = sensor_display_str + names[i]
    +": "+String(values[i],1) + "\n";
  }
  Serial.print(sensor_display_str);
  return sensor_display_str;
}
```

2.3 Modifying the Code for New Sensors

Only minor changes are required in the C++ file to accommodate new sensors. Below, we present examples for adding the MS5837 pressure sensor, the SDC41 CO_2 sensor, and the PT100 temperature sensor. You can compare these examples with the base code displayed in Figure 29.

2.3.1 MS5837 Pressure Sensor with a 10 m Cable

The MS5837-02BA is a pressure sensor which measures pressure up to 2 bars. It is often used to measure water level, for example to measure the water flow with an associated calibration curve. It can hence measure water level up to 10 m and its cable must hence be long to avoid having the datalogger in the water. This cable length implies that we put a lower I^2C frequency to communicate with the device.

To ensure that there is no communication problem after changing the I^2C frequency, I stop and then restart the communication protocol with the `Wire.end()` and `Wire.begin()` methods. For 10 m of cable, I set the frequency to 5000 Hz, although higher frequencies might also work. After reading the sensor data, I switch to another multiplexer channel to minimize interference with other I^2C devices like the RTC. I then again restart the I^2C protocol but with the original clock frequency of 50 kHz.

```
//include all libraries
//...
#include "MS5837.h"
#include "Wire.h"
//instantiate all sensors classes
//...
MS5837 ms5837_sensor;
//declare arrays where the names and values of the sensors are stored
String names[]={"Vbatt","tempSHT","humSHT","tempBMP","pressBMP","pressMS","tempMS","htWat"};
//...
void Sensors::measure() {
  //code from the base code measure method
  //...
  //change the I2C frequency for the long cable
  Wire.end();
  Wire.begin();
  Wire.setClock(5000);
  tcaselect(7);
  delay(5); //delay for the multiplexer
  unsigned long start_time_connection = micros();
  bool timeout = false;
  while (!ms5837_sensor.init() && !timeout) {
    Serial.println("pressure sensor not connected");
    delay(50);
    timeout = (micros()-start_time_connection)>2000000;
  }
  if (timeout) Serial.println("Init pressure sensor failed!");
  else Serial.println("Init pressure sensor ok!");
  ms5837_sensor.setModel(MS5837::MS5837_02BA);
  ms5837_sensor.read();
  values[5]=ms5837_sensor.pressure();
  values[6]=ms5837_sensor.temperature();
  if(values[5]>4030){values[5]=0;}
  if(values[6]<-50){values[6]=0;}
  tcaselect(0); //select another i2c channel to avoid interferences
  delay(5);
  //change back to the original I2C frequency
  Wire.end();
  Wire.begin();
  Wire.setClock(50000);
  delay(50);
  //water height in cm is (water pressure-air_pressure)*10/g if pressure is in millibar
  values[7] = (values[5]-values[4])*10/9.81; //htWat
}
```

Figure 30: Arduino code for the MS5837 pressure sensor.

2.3.2 SCD41 CO2 Sensor

The SCD41 sensor measures CO_2 from 0 to 40000 ppm but with accurate measurements reading between 400 ppm and 5000 ppm. In this range it has an accuracy of $\pm$ 50 ppm, and a repeatability of $\pm$ 10 ppm . This sensor communicates with I^2C. For single shots measurements, the sensors needs

a delay of 5000 ms between measurements. This sensor also measures humidity and temperature, to compensate their effects on the CO_2 sensor (the compensation is done in the sensor chip).

```
//include all libraries
//...
#include "SparkFun_SCD4x_Arduino_Library.h"
//instantiate all sensors classes
//...
SCD4x co2_sensor(SCD4x_SENSOR_SCD41);
//declare arrays where the names and values of the sensors are stored
String names[]={"Vbatt","tempSHT","humSHT","tempBMP","pressBMP","CO2","tempSCD","humSCD"};
//...
void Sensors::measure() {
  //code from the base code measure method
  //...
  tcaselect(7); //connect CO2 sensor to channel 7 of the multiplexer
  delay(5);
  co2_sensor.begin();
  for(int i = 0; i<4;i++){ //take 4 single shot meas. with 5s interval, store the last one
    co2_sensor.measureSingleShot();
    delay(5000); //delay needed specified in the datasheet for single shot measurements
    co2_sensor.readMeasurement();
    values[5]=co2_sensor.getCO2();
    values[6]=co2_sensor.getTemperature();
    values[7]=co2_sensor.getHumidity();
    Serial.print("CO2:");
    Serial.println(values[5]); //only last value printed is stored
  }
}
```

Figure 31: Arduino code for the SCD41 CO2 sensor.

2.3.3 Pt100 Temperature Sensor

The Pt100 is a resistance temperature sensor made using a Platinum element with a resistance of 100 Ω at 0 °C and typically with a 38.5 Ω change in resistance over the range 0 °C to 100 °C. Pt100 Sensors are used for a wide variety of temperature measurement applications.

To measure the resistance of this analog sensor, we use the Qwiic PT100 - ADS122C04 board from Sparkfun. It is based on the ADS122C04 chip, which provides both a stable current source and a precise analog-to-digital converter essential for measuring the Pt100 resistance.

Most Pt100 probes have 4 wires, with 2 wires connected at both sides of the 100 Ω resistance. You might think that 2 wires should be enough to connect a resistor, but this is actually a pretty smart feature to reduce the error due to the resistance of the wires; 2 wires are used to provide current to the resistance and the other 2 are used to measure the voltage. Since there is no current flowing into the wires used for voltage measurement, there will be no voltage drop in these wires, enabling optimal measurement of the Pt100 resistance [7].

To connect the Pt100 to the Sparkfun board, measure which 2 wires are connected together (0 Ω between the 2 wires) and check that there is about a 100 Ω between two connected wires and the two others. Take 2 wires which are connect together (no matter which) and connect them to the ports 1 and 2 of the Qwiic Pt100. Connect the 2 other wires to the ports 3 and 4 of the Qwiic Pt100.

```
//include all libraries
//...
#include <SparkFun_ADS122C04_ADC_Arduino_Library.h>
//instantiate all sensors classes
//...
SFE_ADS122C04 PT100_sensor;
//declare arrays where the names and values of the sensors are stored
String names[]={"Vbatt","tempSHT","humSHT","tempBMP","pressBMP","PT100"};
//...
void Sensors::measure() {
  //code from the base code measure method
  //...
  tcaselect(7);
  delay(5);
  PT100_sensor.begin();
  PT100_sensor.configureADCmode(ADS122C04_4WIRE_MODE);
  values[5] = PT100_sensor.readPT100Centigrade(); //PT100
  PT100_sensor.powerdown();
}
```

Figure 32: Arduino code for the PT100 temperature sensor.

2.3.4 FS3000 Sparkfun Air Velocity Sensor

The FS3000 air velocity sensor measures the air flow in only one direction. If air flows in 2 opposite directions, two sensors placed in opposite directions are needed. Its measurement range is either from 0 to 7 m/s or from 0 to 15 m/s, but the accuracy is typically 5% of this range, we hence use the range from 0 to 7 m/s in our applications, with an accuracy of ± 0.35m/s, but with a resolution exceeding 0.01 m/s.

The sensor comprises a heated sensor element and a temperature sensor. It calculates air velocity by analyzing the rate of heat dissipation.

```
//include all libraries
//...
#include <SparkFun_FS3000_Arduino_Library.h>
//instantiate all sensors classes
//...
FS3000 air_velocity_sensor;
//declare arrays where the names and values of the sensors are stored
String names[]={"Vbatt","tempSHT","humSHT","tempBMP","pressBMP","airSpeed"};
//...
void Sensors::measure() {
  //code from the base code measure method
  //...
  tcaselect(7);
  delay(5);
  air_velocity_sensor.begin();
  air_velocity_sensor.setRange(AIRFLOW_RANGE_7_MPS);
  values[5]=air_velocity_sensor.readMetersPerSecond();
}
```

Figure 33: Arduino code for the FS3000 Sparkfun anemometer.

3 Wireless Transmission

3.1 The Notecard

We use a Qwiic cellular notecarrier (see Figure 34) to send the data from our dataloggers wirelessly. It is made of a notecard developed by Blues Wireless mounted on a notecarrier developed by Sparkfun which allows us to communicate easily via I^2C with the notecard.

The Qwiic notecarrier costs about 80CHF but this price includes a 10 year subscription and 500MB of data.

This notecard uses significantly more power than the rest of our devices connected to the datalogger (between 100 mA and 300 mA) but since we only use it for dataloggers placed near from the surface, it is often possible to put a wire for a solar charger for the battery. But to minimize the power consumption, we have another MOSFET which controls the power supply of the notecard, to allow us to only power it when we need it.

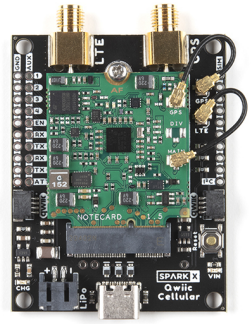


Figure 34: Qwiic cellular notecarrier (in black) with the notecard (in green) mounted on it.

Connecting the Notecard to our datalogger is straightforward: the SCL and SDA pins on the notecarrier are linked to an I^2C channel on the multiplexer. The 3V and GND pins are connected to a MOSFET driver, usually controlled by pin 4 on the TinyPico. An antenna must be connected to the MAIN port of the notecard, or screwed to the LTE port of the notecarrier.

The type of antenna can have a big impact on the quality of the communication. The length of the cable between the notecard and the antenna, and the number of connections can also impact negatively the strength of the communication signal. The choice of antenna depends on the frequencies required. The notecard communicates with LTE-M, NB-IoT, and Cat-1 networks globally [8]. According to [9], Switzerland uses bands 20, 3, 1, 7, 8 for LTE-M network coverage, which frequencies are given in [10]. With this information, I would either select an antenna with frequency range between 791 MHz and 960MHz or with a frequency range between 1710 MHz and 2170 MHz.

For our dataloggers, we used for example the TLS.01.305111 antenna, which supports 5G,4G,3G,2G,NB-IoT,CAT-M1 and Wi-Fi networks.

3.2 How to Send and Receive Data

As illustrated in Figure 35, the TinyPico (acting as the product control unit) communicates with the Qwiic notecarrier (serving as the customer's MCU application) via I^2C. This notecarrier communicates with the notecard which wirelessly connects to notehub.io (the Notehub data router). Notehub provides an API that enables forwarding of the received data to various cloud services; in our case,

iotplotter.com.

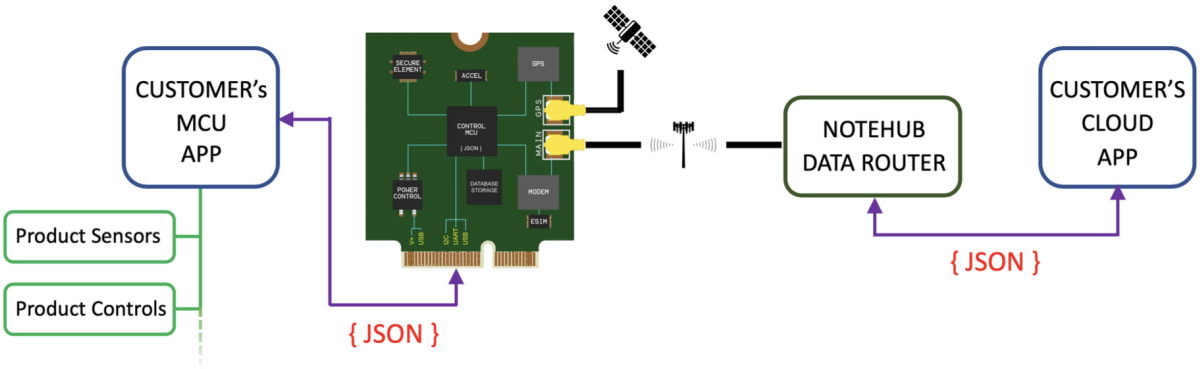


Figure 35: Data transfer using a notecard.

3.2.1 Controlling the Notecard with JSON commands

All commands sent to the Notecard utilize JSON (JavaScript Object Notation) formatting. JSON employs specific symbols such as { }, :, ", [ ], and adheres to the following syntax:

- Data is represented in key/value pairs.
- The colon (:) assigns a value to key.
- Key/value pairs are separated with commas (,).
- Curly brackets hold objects ({ }).
- Square brackets hold arrays ([ ]).

A typical JSON representation of key/value pairs would be:

```
{"key1":"value1", "key2":"value2", "key3":"value3"}
```

In JSON, values can be strings, integers, bools or even nested JSON objects and arrays. For example:

```
{
    "name": "Rui",
    "sports": {
        "outdoor": "hiking",
        "indoor": "swimming"
    },
    "pets": [
    {"name":"Max","weight":25.2},
    {"name":"Dique","weight":5.6}
    ]
}
```

Table 2 lists some common JSON commands for controlling the notecard. More commands and detailed information can be found in the notecard quickstart tutorial. You can find tutorials about almost every aspect of the data transmission in their notecard guides & tutorials, and there is a detailed and comprehensible explanation about all the possible JSON requests which you can send to the notecard in the notecard api documentation.

JSON Command for the notecard	**Description**
{"req":"hub.set", "product":"com.gmail.ouaibeer:datalogger_isska"}	Configure the Notecard so it knows from where to send and receive data.
{"req":"note.add","body":{"temp":35.5,"humid":56.23}}	Queue a note to the Notecard. The "body" of this note is a JSON which contains the data to send .
{"req":"hub.sync"}	Initiate a synchronization between the Notecard and Notehub. If a note is queued, this note will be uploaded to the notehub. If a note from Notehub is queued, this note will be downloaded by the notecard.
{"req":"hub.sync.status"}	Check on the state of the synchronization with Notehub. The response is a JSON file containing all the needed information about the synchronization.

Table 2: Example of usefull JSON commands to control the notecard.

3.2.2 Sending JSON Commands with Arduino Code

To send JSON commands to the notecard, we use the `Notecard.h` library developed by blues wireless. Figure 36 shows the methods needed to send JSON commands to the notecard and to receive JSON responses from the notecard. There is either the `sendRequest()` method which simply sends a JSON command and the `requestAndResponse()` method which sends a JSON command and return its JSON response from the notecard.

```
#include <Notecard.h>
Notecard notecard;
//...
notecard.begin();
//create the JSON command files req1 and req2
//...
notecard.sendRequest(req1); //send a JSON command which requires no response
J *my_JSON_response = notecard.requestAndResponse(req2); //send a command and store the response
//do something with the JSON response
//...
notecard.deleteResponse(my_JSON_response); //delete the JSON file if not needed anymore
```

Figure 36: Arduino code to send and receive JSON files.

To create a JSON command, the `newRequest()` method is used. The argument of this method say which request to send, and we can then add options to the request by adding Booleans, Strings, Floats or Integers with the `JAddTypeToObject()` methods. The JSON file type is `J*`.

```
//create the following JSON requests:
//{"req":"hub.set","product":"com.gmail.ouaibeer:datalogger_isska","mode","minimum"}
J *req1 = notecard.newRequest("hub.set");
JAddStringToObject(req1, "product","com.gmail.ouaibeer:datalogger_isska");
JAddStringToObject(req1, "mode", "minimum");

//{"req":"hub.sync"}
J *req2 = notecard.newRequest("hub.sync")

//{"req":"note.get", "file":"data.qi", "delete":true}
J *req3 = notecard.newRequest("note.get");
JAddStringToObject(req3, "file", "data.qi");
JAddBoolToObject(req3, "delete", true);
```

Figure 37: Create some simple JSON requests.

For more complex JSON structures, the `JCreateObject()` method is used to generate a JSON object. You can add strings to it, or integers, boolean and even other JSON objects or arrays of JSON objects with the `JAddArrayToObject()` method or add items to the array with the `JAddItemToArray()` method.

3.3 Base Code Modifications for Dataloggers with a Notecard

We will now see what changes are made to the Arduino base code explained in section 2.2 to enable wireless data transmission via the notecard. A schematic representation of the modified Arduino code is presented in Figure 38, where alterations specific to wireless communication are highlighted in purple. The full source code incorporating these wireless capabilities can be accessed on Github.

The code which is specific to the sensors added to the datalogger is exactly the same whether you add a notecard or not.

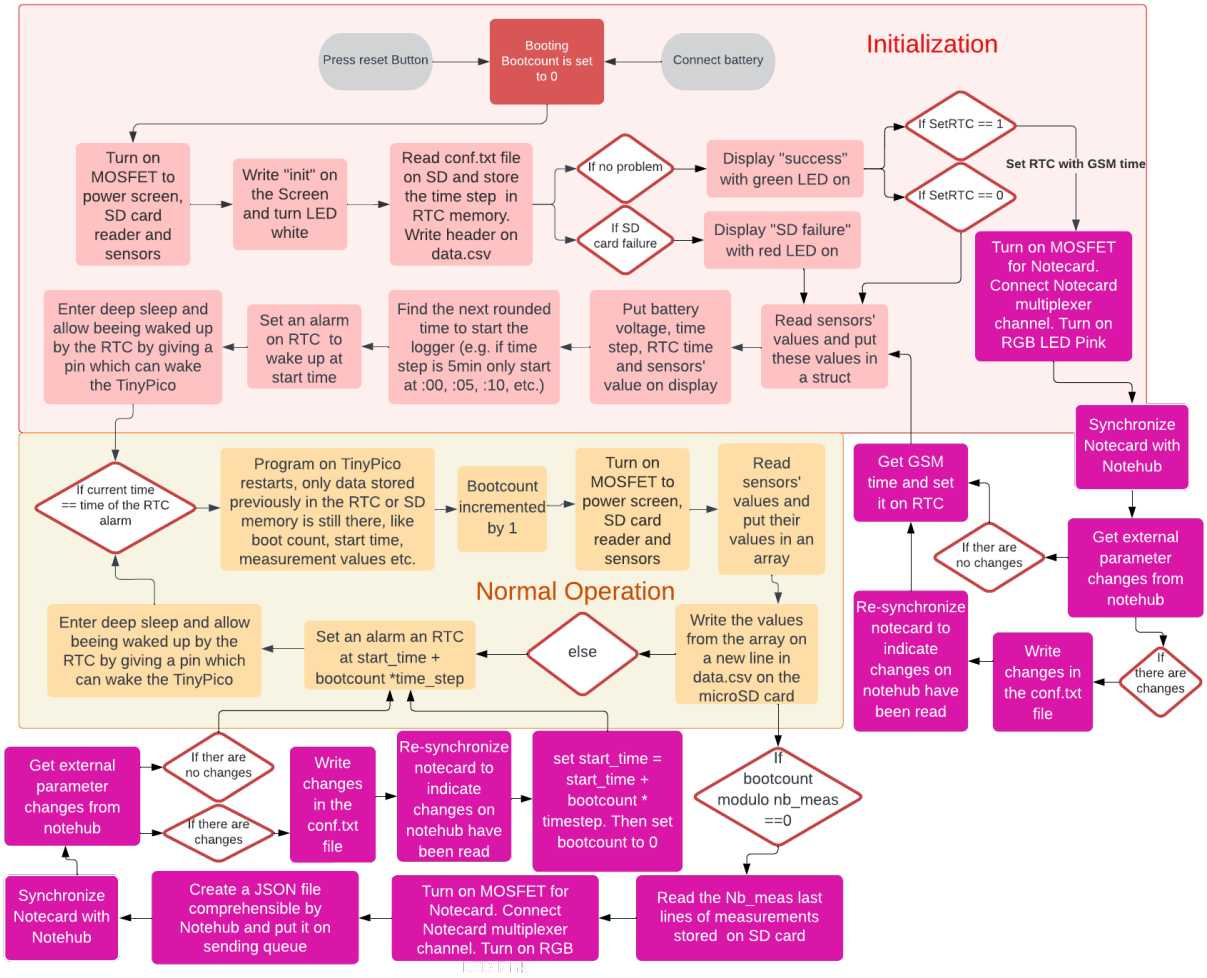


Figure 38: Datalogger base code diagram. Functions adapted for wireless communication are highlighted in purple. Made with Lucidchart.

The following subsections will offer a detailed exploration of how these adaptations are implemented, particularly focusing on:

- Synchronizing the notecard with notehub
- Retrieving the current time from the cellular network
- Formulating a JSON file containing the gathered sensor data
- Transmitting sensor data to notehub
- Importing external parameters from notehub

3.3.1 Synchronizing the Notecard with Notehub

Synchronizing the notecard with notehub is the way to either transfer data from notehub (a server) to the notecard (the datalogger) or from the notecard to notehub or even both at the same time.

On both the notecard or on notehub, you can queue data which must be sent, and when you do the synchronization, the data in the queue is sent from one device to the other.

To initiate synchronization with notehub, you must already have set up a notehub project. Each notehub project is associated with a unique productUID which will be used in the code for the notecard to link the notecard with your notehub project. In our case the productUID is : `com.gmail.ouaibeer:datalogger_isska`. The first command that we send to the notecard tells the notecard which productUID and which synchronizing mode is used. The JSON command is the following:

```
{"req":"hub.set", "product":"com.gmail.ouaibeer:datalogger_isska", "mode":"minimum"}
```

The minimum mode disables periodic connection. The notecard will not synchronize until it receives an explicit `hub.sync` request.

A `hub.sync` request must then be sent to start the synchronization. Given that synchronization may take between 1 to 2 minutes, it is essential to wait for its completion. The synchronization status can be checked using the following command:

```
{"req":"hub.sync.status"}
```

Among various details provided in the synchronization status, the `completed` parameter indicates the time passed (in seconds) since the synchronization was finished. Therefore, checking whether `completed` is non-zero allows us to determine if the synchronization is complete or not. The Arduino code that implements these commands is illustrated in Figure 39.

```
/* before using the notecard the following functionalities must be called to initialize it:
  digitalWrite(MOSFET_NOTECARD_PIN, HIGH);
  sensor.tcaselect(NOTECARD_I2C_MULTIPLEXER_CHANNEL);
  delay(100);
  notecard.begin();
  delay(400);
  //set communication parameters for the notecard
  J *req = notecard.newRequest("hub.set");
  JAddStringToObject(req, "product", myProductID);
  JAddStringToObject(req, "mode", "minimum");
  notecard.sendRequest(req);
*/

bool synchronize_notecard(){ //returns true if notecard is synchronized with notehub
  //send command to sync the notecard
  notecard.sendRequest(notecard.newRequest("hub.sync"));
  //wait for syncronisation
  bool time_out = false;
  unsigned long start_time_connection = micros();
  int completed=0;
  do{ //this loop wait until then connection is completed or until we reach 4 minutes (timeout)
    J *SyncStatus = notecard.requestAndResponse(notecard.newRequest("hub.sync.status"));
    completed = (int)JGetNumber(SyncStatus,"completed");
    notecard.deleteResponse(SyncStatus); //delete the response
    if((micros()-start_time_connection)/1000000>240){ //after 240s stop waiting for connexion
      time_out = true;
      return false;
    }
    delay(500); //check connection status every 500ms
  } while (completed==0 && !time_out);

  if (completed!=0){
    Serial.println("The notecard is syncronized!");
    return true;
  }
  else{
    Serial.println("The notecard failed to synchronize!");
    return false;
  }
}
```

Figure 39: Code snippet for synchronizing the notecard with notehub.

3.3.2 Retrieving Network Time and Network Signal Strength

During synchronization, the notecard also synchronizes its time with the time from the cellular network. After the synchronization we can get the time of the notecard with the following JSON command:

The notecard synchronizes its internal clock with the cellular network during the synchronization process. After the synchronization, the current time can be queried from the notecard using the

following JSON command:

```
{"req":"card.time"}
```

Executing this command returns a JSON object formatted as follows:

```
{
    "time": 1599769214,
    "area": "Beverly, MA",
    "zone": "CDT,America/New York",
    "...":"..."
}
```

To receive this JSON file we use the `requestAndResponse()` method. We only need the `time` information from the JSON file. To select it, since the time is a number we use the `JgetNumber()` method, with `time` as argument.

```
J *rsp = notecard.requestAndResponse(notecard.newRequest("card.time"));
unsigned int received_time = (unsigned int)JGetNumber(rsp, "time");
notecard.deleteResponse(rsp);
```

Figure 40: Retrieving the notecard's current time in Unix format.

To get information about the quality of the network signal, the following JSON command can be used:

```
{"req":"card.wireless"}
```

This returns a complex JSON object that includes a `net` object with the attribute `bars`, which represents the signal strength:

```
{
    "status": "{modem-off}",
    "count": 1,
    "net": {
        "band": "LTE BAND 2",
        "bars": 1,
        "updated": 1599225076,
    }
}
```

To access the `bars` information of this JSON file, we must first get the `net` JSON object with the `JgetObjectItem()` method and then get the `bars` value from this object.

```
J *wireless_info = notecard.requestAndResponse(notecard.newRequest("card.wireless"));
J *net_infos = JGetObjectItem(wireless_info,"net");
int bars = JGetNumber(net_infos, "bars");
```

Figure 41: Retrieving the network signal strength.

3.3.3 Sending Sensor Data from the Notecard to Notehub

We don't send the sensors data after each measurement, because it uses a lot of battery power, which should be minimized. The number of measurements to be sent at once can be set on the `conf.txt` file.

Whenever the boot count is a multiple of the configured number of measurements to send, we package these measurements into a single JSON file and send them via the notecard.

```
void setup(){
    bootCount_since_change++;
    //...
    measure_all_sensors();
    if (bootCount_since_change%nb_meas_sent_at_once == 0){
        send_data_wirelessly();
    }
    //...
}
```

Figure 42: Send data only after accumulating a certain number of measurements.

To route the JSON file containing the sensor data correctly, it must have the following format:

```
{
    "sensor_name_1": [{
            "value": "4.46",
            "epoch": "1693817760"
        },{
            "value": "4.47",
            "epoch": "1693818360"
        },{
            "value": "4.44",
            "epoch": "1693818960"
        }],
    "sensor_name_2": [{
            "value": "26.75",
            "epoch": "1693817760"
        },{
            "value": "26.81",
            "epoch": "1693818360"
        },{
            "value": "26.80",
            "epoch": "1693818960"
        }]
}
```

For each sensor, we create a JSON array containing all the measurements and their corresponding timestamps (in Unix time, UTC+0). In the example above, we have 2 sensors, and we send 3 measurements of these sensors at once.

The Arduino code in Figure 43 shows how to create this JSON file using arrays that contain the sensors' names, values, and measurement times.

```
J *data = JCreateObject();
for(int i=0; i<sensor.get_nb_values();i++){
    J *sensor = JAddArrayToObject(data,sensor_names[i]);
    for(int j=0; j<nb_lines_to_send;j++){
        J *sample = JCreateObject();
        JAddItemToArray(sensor, sample);
        JAddStringToObject(sample, "value", data_matrix[j][i].c_str());
        JAddStringToObject(sample, "epoch", time_array[j]);
    }
}
```

Figure 43: Create a JSON file containing the sensors data.

Once the data JSON file is created we can add it to to an outgoing queue which will be sent when we synchronize the notecard. This is done with the `note.add` request, to which we specify the `body` which is in this case our sensors' data JSON file.

```
void send_data_wirelessly(){
    //initialize notecard and create data JSON file
    //...
    J *body = JCreateObject();
    JAddItemToObject(body, "data", data);
    J *req_queue_data = notecard.newRequest("note.add");
    JAddItemToObject(req_queue_data, "body", body);
    notecard.sendRequest(req_queue_data);
    //synchronize notecard and turn it off
    //...
}
```

Figure 44: Send sensor data using a notecard.

We actually don't have the `data_matrix` and `time_array` but only have the data stored in `data.csv` on the SD card. We must hence create these arrays before sending the data over cellular network.

The code in Figure 45 does the following; it takes the last x lines from the csv file, puts the different sensors' values in a matrix called `data_matrix` and the measurement times in an array called `time_array`.

The time stored on the SD card is a string of the time of the measurement in UTC+1 in the format `YYYY/MM/DD hh:mm:ss`, but we need the Unix time in UTC+0 for the notecard. We hence must first extract the year, month, day etc. from this string, convert it in DateTime format, use built-in functionalities from the DateTime format to convert it into Unix time and then subtract 1 hour (3600 seconds) from it.

```
//find the position in data.csv where the first x lines to be sent begin
File myFile = SD.open(DATA_FILENAME, FILE_READ);
int position_in_csv = myFile.size(); //get the position of the last character in the csv file
for (int i = 0; i < nb_meas_sent_at_once;i++){
    position_in_csv = position_in_csv -20; //a line is certainly longer than 20 characters
    myFile.seek(position_in_csv);
    while (myFile.peek() != '\n'){
        position_in_csv = position_in_csv - 1;
        myFile.seek(position_in_csv); //go backwards until we detect the previous line separator
    }
}
myFile.seek(position_in_csv+1);

//read the the x lines in the csv and store the values in the arrays.
//Each element in a line is followed by a ";" and each line by a "\n" char
String data_matrix[nb_meas_sent_at_once][sensor.get_nb_values()];
int time_array[nb_meas_sent_at_once];
int counter=0; //counts the lines in the matrix
while (myFile.available()) {
    for (int i =0; i<2+sensor.get_nb_values();i++){ //the +2 is for datetime and bootcount
        String element = myFile.readStringUntil(';');
        if (element.length()>0 &&i==1){ //convert time string to unixtime
            String day_stored = element.substring(0,2);
            String month_stored = element.substring(3,5);
            String year_stored = element.substring(6,10);
            String hour_stored = element.substring(11,13);
            String minute_stored = element.substring(14,16);
            String second_stored = element.substring(17,19);
            DateTime datetime_stored = DateTime(year_stored.toInt(),month_stored.toInt(),
            day_stored.toInt(),hour_stored.toInt(),minute_stored.toInt(),second_stored.toInt());
            time_array[counter] = datetime_stored.unixtime()-3600; //convert to UTC+0
        }
        if (element.length()>0 &&i>1) data_matrix[counter][i-2] = element;
    }
    myFile.readStringUntil('\n');
    counter++;
}
myFile.close();
```

Figure 45: Create data matrix and measurement time array from SD card data.

3.3.4 Sending Parameters from Notehub to the Notecard

To send data from notehub to the notecard, you must have already synchronized your notecard with a notehub project once.

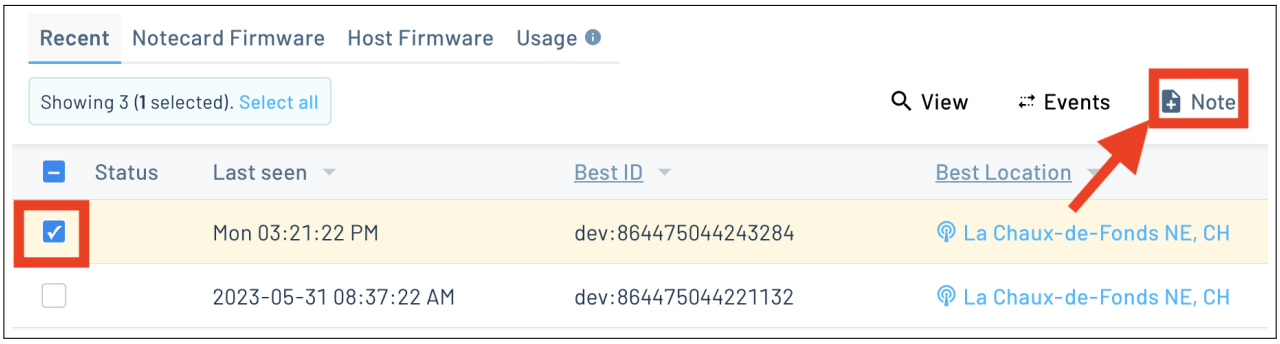


Figure 46: Selecting a device on Notehub.

On your Notehub project, select your device by ticking the small square, which should turn blue. Next, click on the "+ Note" button. A pop-up will appear where you can select a notefile and write a JSON note. For the notefile, always select `data.qi`.

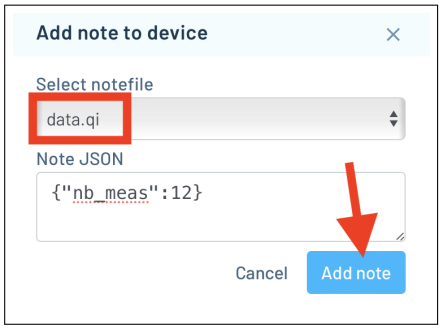


Figure 47: Adding a note in Notehub.

For the JSON message, you can specify either the number of measurements to be sent at once or the time step between two measurements in seconds, as shown by the JSON examples below:

```
{"time_step":600}
```

Figure 48: Setting the time step between two measurements to 600 seconds.

```
{"nb_meas":12}
```

Figure 49: Setting the number of measurements to be sent at once to 12.

The changes made on notehub are automatically sent to the notecard when we do a synchronization. To check if there were changes made on notehub, we send the following JSON command to the notecard:

```
{"req":"file.change"}
```

The JSON response for this command provides a number within `info` → `data.qi` → `total`. This number indicates the total changes made on notehub. If this number is zero, no action is taken. However, if the number is greater than zero, it implies that changes were made on notehub. To retrieve these changes, the following JSON command is used:

```
{"req":"note.get", "file":"data.qi", "delete":true}
```

This command return the first note added on `data.qi` on a queue and deletes it from the queue. We send this JSON command as many times as there are changes made on `data.qi` to get the latest changes at the end. We check if the `body` of these notes contains either a value labelled `nb_meas` or `time_step`. If this is the case we adjust the time step and the number of measurements to be sent at once with this number. If the spelling is wrong the changes just won't be applied.

```
J *notecard_changes = notecard.requestAndResponse(notecard.newRequest("file.changes"));
J *changes_infos = JGetObjectItem(notecard_changes,"info");
J *data_changes = JGetObjectItem(changes_infos,"data.qi");
int nb_changes = (int) JGetInt(data_changes,"total");
notecard.deleteResponse(notecard_changes);
int new_time_step = 0;
int new_nb_meas_sent_at_once = 0;
while (nb_changes>0){
    //the parameters changed come in a FIFO queue
    J *req = notecard.newRequest("note.get");
    JAddStringToObject(req, "file", "data.qi");
    JAddBoolToObject(req, "delete", true);
    J *parameter_changed = notecard.requestAndResponse(req);
    J *parameter_changed_body = JGetObjectItem(parameter_changed,"body");
    //adjust the correct parameter depending on its name
    int temp_time_step = (int) JGetInt(parameter_changed_body,"time_step");
    if(temp_time_step != 0){
        new_time_step = temp_time_step;
    }
    int temp_nb_meas_sent_at_once = (int) JGetInt(parameter_changed_body,"nb_meas");
    if(temp_nb_meas_sent_at_once != 0){
        new_nb_meas_sent_at_once = temp_nb_meas_sent_at_once;
    }
    notecard.deleteResponse(parameter_changed);
    nb_changes-=1;
}
```

Figure 50: Get changes from notehub on the notecard.

3.4 Data Routing on IoTplotter

When data is sent to notehub, you can see its JSON file in the **Event** section on notehub. However, this is not convenient to visualize data and the received messages are deleted after a few days. But notehub allows us to route the data to another online service like Google Cloud, Amazon Web Storage etc.

In our case we use a service called IOTplotter. IoTplotter, created by Joseph Hewitt, is a free service designed for the maker community. It collects data from IoT devices for long-term graph plotting and storage.

To create a route from notehub to IoTplotter, just follow these steps:

- Go on notehub, on your project from which you took the productUID in your datalogger code. Open the project and click on **Event**. Here you can verify that your datalogger sends data to notehub. You should receive notefiles called `data.qo` which have a JSON body which corresponds to the data you sent with the notecard.

Figure 51: Check events on notehub.

- If the data is successfully sent to notehub, check in which fleet your notecard belongs. For this click on **Device**. You will see the last devices which sent something, but also to which fleet they belong. You can obviously change the fleet of a device by clicking on the device and selecting the fleet in its settings.

Figure 52: Check the device's fleet.

- Then go on IoTplotter (you need an account). On the main page, select an unused feed which you can modify, or create a new feed.

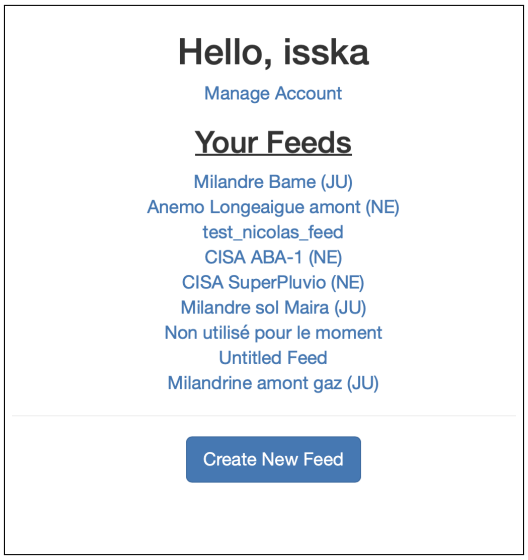
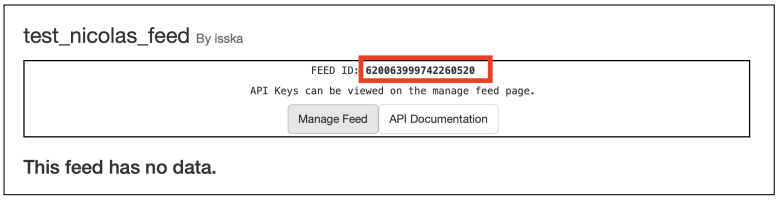
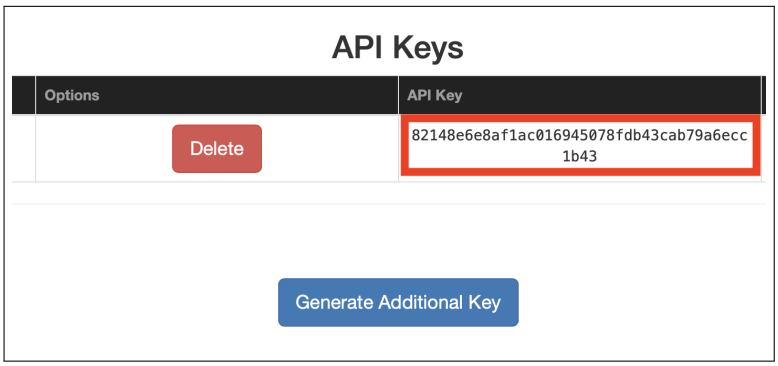


Figure 53: Select or create feed on IoTplotter.

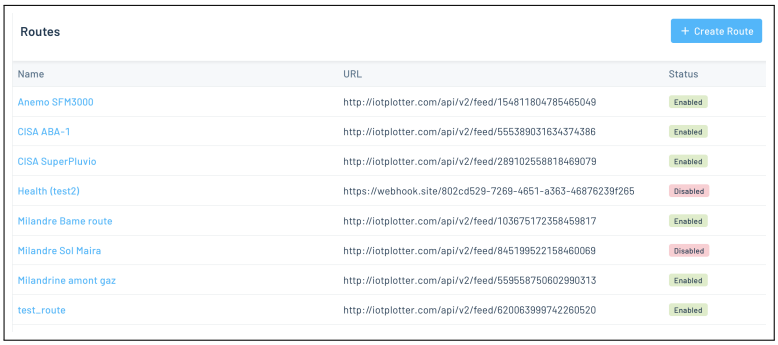
- Notice that you have a feed ID which you will need to copy on notehub, so store it somewhere. Then click on **Manage Feed**. There you will be able to give a name to your feed.



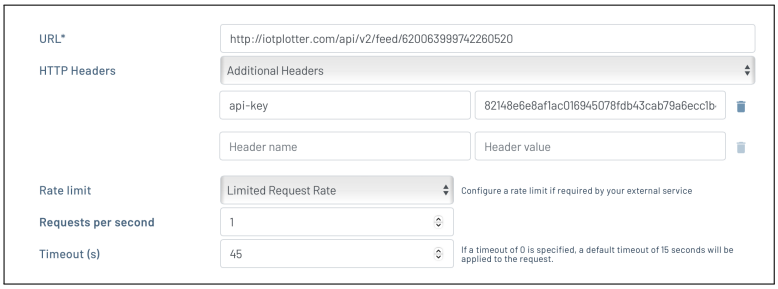
- On the left corner, click on **Keys**. There you will see the API-key which you will also need to copy on notehub.



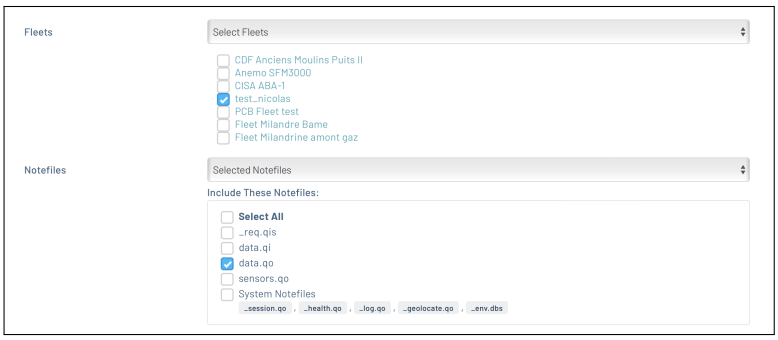
- Once you have the API-key on the feed ID, go back to notehub. Click on **Route**. On the corner right, click on **New Route** or on an existing unused route to modify it. There you will be able to enter a name for your route.



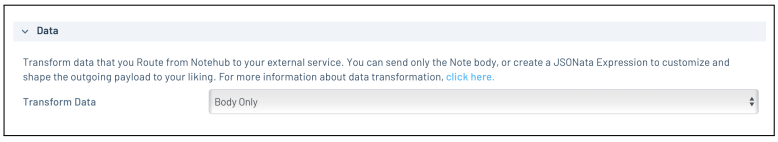
- For the route configuration, you must enter as URL the following link: `http://iotplotter.com/api/v2/feed/ + your feed ID`.
  Choose the option **AdditionalHeaders** and write on the left `api-key` and on the right copy your API-key from IoTplotter. Select the option **Limited Request Rate** as rate limit and choose similar settings as shown below.



- Choose the options **select fleet** and **select notefile**. For the fleet, select the fleet of your device. For the notefile, always use `data.qo`.



- In the **Transform Data** section, choose **body Only**. This will only send the body of your notefile to IoTplotter.



- Save your changes by clicking on **Apply changes**. You are done. After data is sent from your datalogger on notehub, go on IoTplotter, on the feed which you chose. You should see the data and if you click on one sensor, a graph should appear. After a few days, it could look like this:

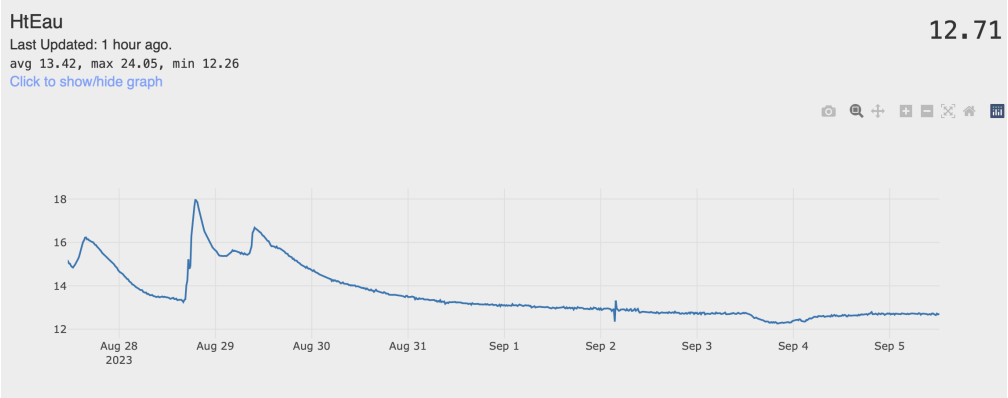


Figure 54: Example of a water height graph made with IoTplotter.

4 PCB Design

To connect the different devices of the datalogger, we designed printed circuit boards (PCB). Manufacturing PCBs has become very affordable and offers greater reliability than soldering on a breadboard. We design our PCB with EasyEDA and order them on JLCPCB.

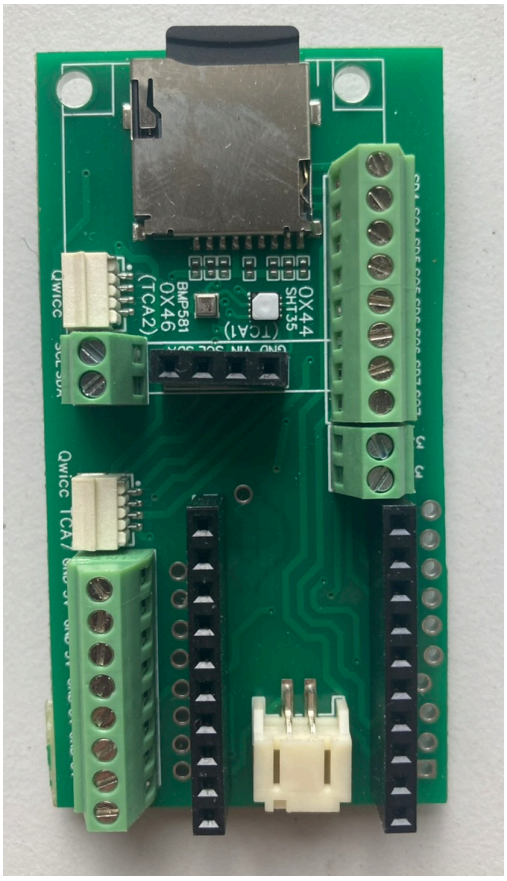


Figure 55: Top view of a datalogger PCB.

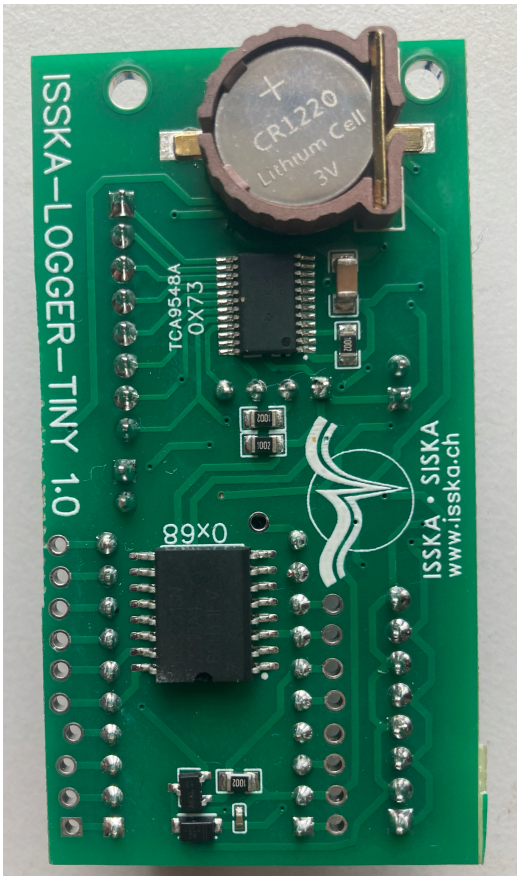


Figure 56: Bottom view of a datalogger PCB.

JLCPCB is the largest PCB prototype enterprise, based in Shenzhen, China. They offer quick and cost-effective PCB printing services. JLCPCB collaborates with LCSC, China's leading electronic components distributor, to provide components for PCB assembly. They offer a robust and user-friendly PCB design software called EasyEDA, which is free and compatible with all operating systems. They obviously made it super easy to get from a PCB design on EasyEDA to a PCB order at JLCPCB, which is why the software is free.

There are two main stages in the PCB design:

- **The schematic design**: This is the first part in the design process. You select all the components that you need on your PCB and mark all the electrical connections between the components.
- **The board design**: Once the design is complete, you can start the board design. You will see all the parts you selected in the schematic with their footprints. You must manually place

these parts in a way that minimizes crossings between electrical connections. After all the parts are placed, you can trace the wires between the components. If two wires must cross, you'll need to add vias, which are holes in the PCB that allow the wire to pass to the other side. Finally, you can add text that will be printed on the PCB and trace the board outline.

4.1 Design Tips

This video provides an entertaining introduction to PCB design with EasyEDA. Additionally, here are some tips based on my experience with EasyEDA:

- **Component Selection:** When you select the parts for your PCB, always begin by searching for your parts in JLCPCB's parts library. Important considerations include:
  - Check for equivalent JLCPCB basic parts, which are generally cheaper and always in stock. For this, search for your part on the JLCPCB parts library, and use a filter to only select basic parts.
  - Prices can vary for similar parts. Make sure to opt for the more economical yet reliable options.
  - After confirming the part and reviewing its datasheet, you can add it to your EasyEDA project by copying the JLCPCB part number, entering it in the search bar which appears after clicking on the "Library" icon in EasyEDA.
- **Use typical application circuits:** The datasheets often provide a "Typical application circuit" for complicated pieces like sensors, multiplexers etc. If you are not 100% sure about how to use a component, copying the "Typical application circuit" in your circuit is often the best choice to make.

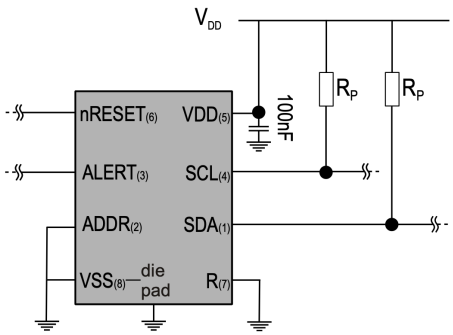


Figure 57: Typical application from the SHT35 datasheet.

- **Connect I²C address pins:** For I²C devices, carefully check that all pins used for the address choice of the device are connected or you won't detect the device. Be also very careful if the device has for example a RESET pin. Check in the datasheet how this pin must be connected and do not simply leave it floating, hoping for the best.
- **Use net ports:** For the schematic, use "Net ports" in the "wiring tools" to label wires and to connect components without tracing a wire between them. It makes your schematic way more

clean and comprehensible. It will also greatly help during the PCB design, since the labels will appear on your connections, which allow you to know which is which.

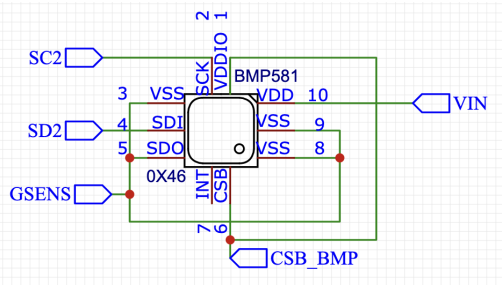


Figure 58: Use net ports to connect the components in the EasyEDA schematic.

- **Finalize schematic first:** Be certain of your schematic before moving on to part placement to avoid subsequent issues.
- **Assembly sides:** If you have space, don't put pieces on both side of the PCB, this costs significantly more to assemble (still pretty cheap) but this is a great way to save space if needed. Simply select the part and then click on "BottomLayer" in the "Layers and Objects" window to put it on the bottom.
- **Verify placement with 3D viewer:** To check that you don't have overlapping components, use the 3D viewer tool to visualize all the components mounted on your board.
- **Don't use autorouter:** I would recommend not using the autorouter. You often can find better paths for your wires than the autorouter, so it actually can take more time to correct the paths from the autorouter.
- **Avoid sharp turns in your circuits:** Two 45° turns are always better than one 90° turn.

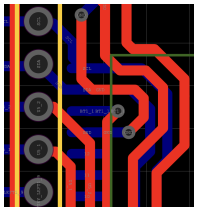


Figure 59: Avoid sharp curves on PCB.

- **Make wires large:** If you have space, make your wires, the vias and the space between wires as thick as you can, it doesn't cost more.
- **Version:** Always write the PCB version on the board.

There are two datalogger PCB designs made on EasyEDA on Github, as well as a PCB made for the MS5837 pressure sensor. The one called `isska-logger-tiny` has pieces on both sides to optimize space, but has no MOSFET driver for the nocard. It is thought for applications where the datalogger must be in a confined space and where it doesn't need to send data online. The one called simply `isska-logger` is bigger and has a Qwiic port to directely connect a notecard, as well as screw to

screw the notecard on the logger. Since the pieces are only printed on one side of the PCB, it is a bit cheaper to manufacture as the `isska-logger-tiny`.

You can open them with EasyEDA. But if you can't use EasyEDA, there is also a svg of the schematics as well as a svg of the PCB design, as shown in Figure 60 and Figure 61.

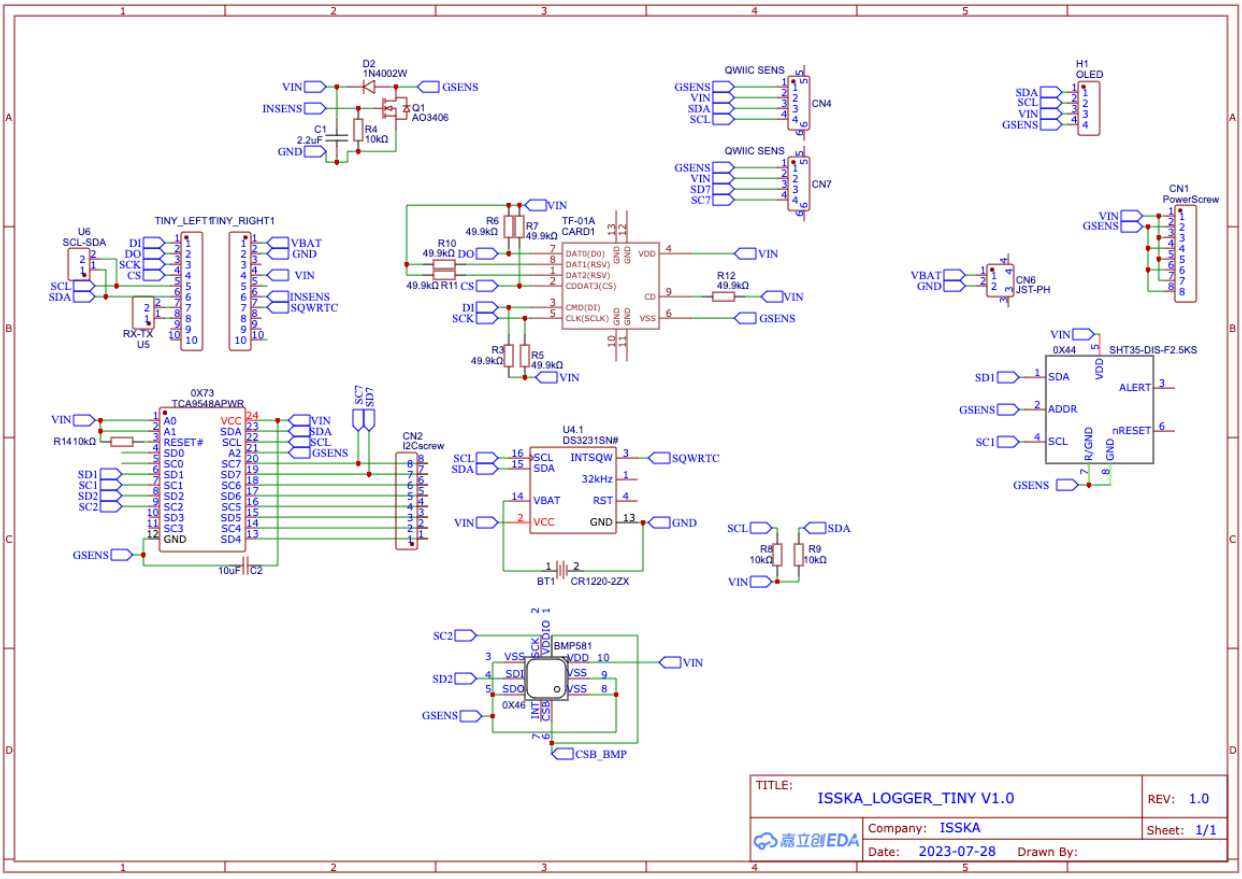


Figure 60: Schematic of the isska-logger-tiny on EasyEDA.

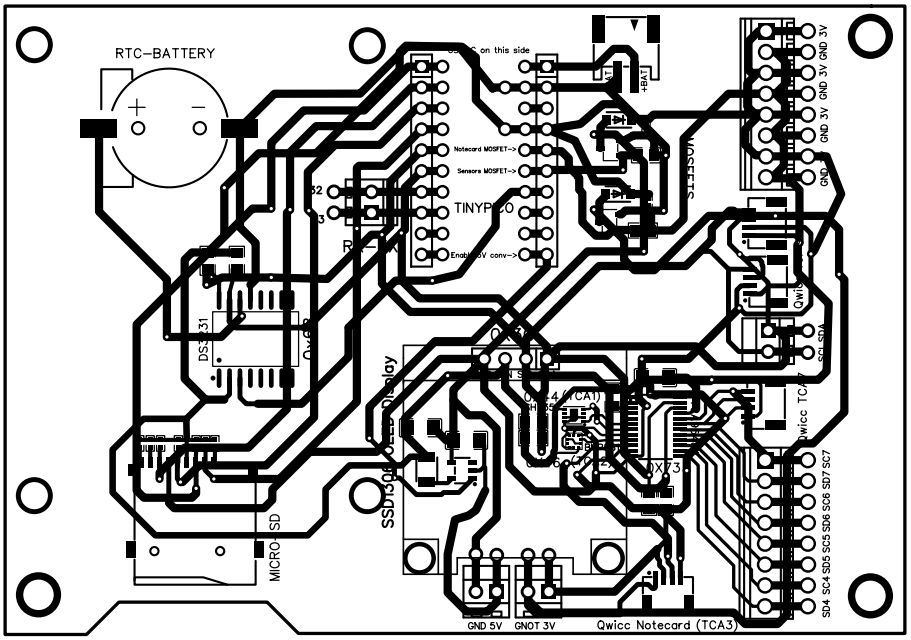


Figure 61: PCB design of the iskka-logger 1.2 with top and bottom layers.

4.2 Order a PCBA on JLCPCB

To order an assembled PCB (PCBA) from JLCPCB, you'll first need to generate three specific files in EasyEDA:

- **Gerber file** : Contains all the PCB layout information but not the components used.
- **Bill of Materials (BOM) file:** A csv file listing the components and their identifiers on the PCB.
- **Pick and Place file:** A csv file indicating the placement of each part on the PCB.

You have one button to generate each of these 3 files on EasyEDA on the top right. Use the default setting when generating them.

You can find these 3 files for the PCB that I made on Github.

Once they have been generated, go here on JLCPCB to place your order. Upload the Gerber file and specify the quantity. Recommended settings to modify are:

- Choose "ENIG" as surface finish. It costs a bit more than the default option, but the default option used lead which is highly toxic.
- Set "Confirm production file" to "Yes". It only costs a few cents for an engineer to check that what you order makes sense and correct it if you did an obvious mistake. If you chose this, you will receive a mail when your file has been reviewed and you must confirm it as soon as possible when it is ready to confirm.
- Select the PCB Assembly option.
- Set "Confirm parts placement" to "Yes".
- If you plan on assembling pieces on both sides of the PCB, you must chose the "Standard" PCB type and select "Both sides" under "Assembly side".

Then click on "Next", upload your "Bill of Material" and "Pick and Place" files, click on "process BOM and CPL".

A list of your components will appear. For each part that you already have in your parts library, click on the part and then on the "Search part" icon. If the parts are available in the right quantity in your parts library, they will show up and you simply can press on "select" to use them.

If you want to order 10 PCBs you often need more than 10 of each piece. You often need 1 or more spare pieces in case one gets broken or lost during the assembly process. Each part has an attrition quantity which exactly specifies how many spare pieces you need. To find the attrition quantity, search for your part on the the JLCPCB parts library, select your part and then click on the "Parts calculators" button which stands right next to the price of the piece. But in general, it is faster to directly contact the assistance on JLCPCB.

References

[1] Seon Rozenblum, *TinyPico, A Tiny Mighty ESP32 Development Board*, August 2023.

[2] Texas Instrument, *I2C Bus Pullup Resistor Calculation*, February 2015

[3] Maxim Integrated Products *Extremely Accurate I2C-Integrated RTC/TCXO/Crystal*, 2015

[4] Alpha and Omega semiconductor, *AO3406 30V N-Channel MOSFET*, 2011

[5] Seon Rozenblum, *Getting satrted, TinyPico FAQ*

[6] Adafruit, *RTC_DS3231 Class Reference*

[7] All about circuits, *Kelvin (4-wire) Resistance Measurement*

[8] Blues, *The Notecard*

[9] Emnify, *LTE-M Connectivity*

[10] RF Wireless World, *LTE-M Frequency Bands*