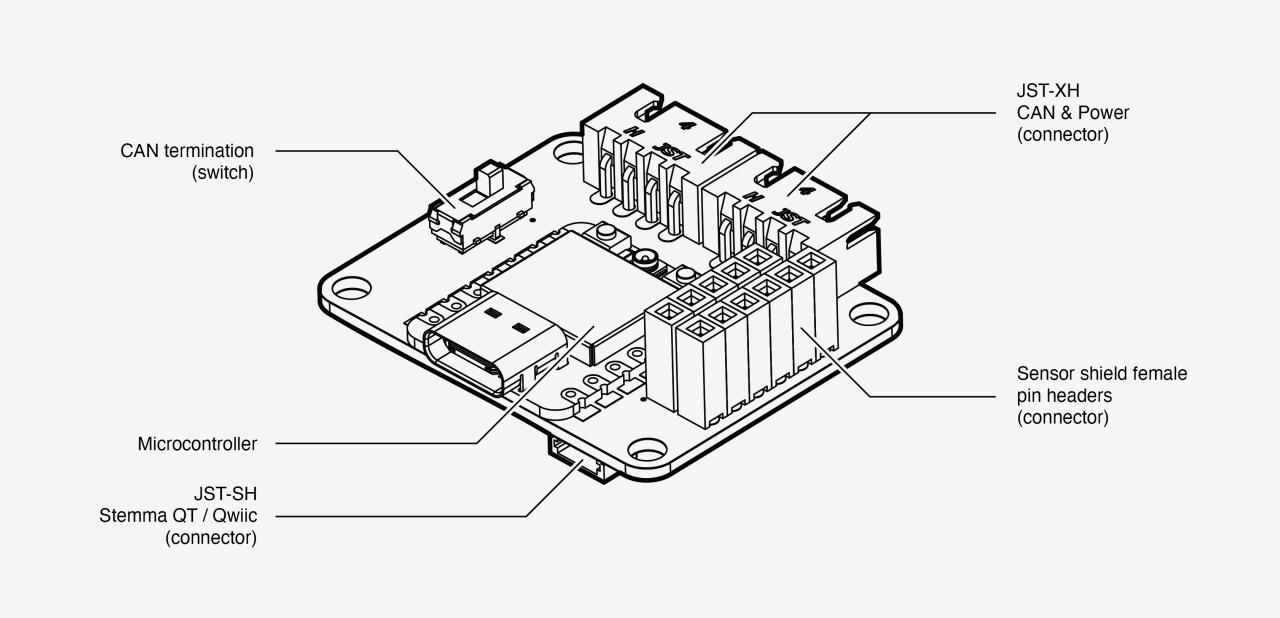
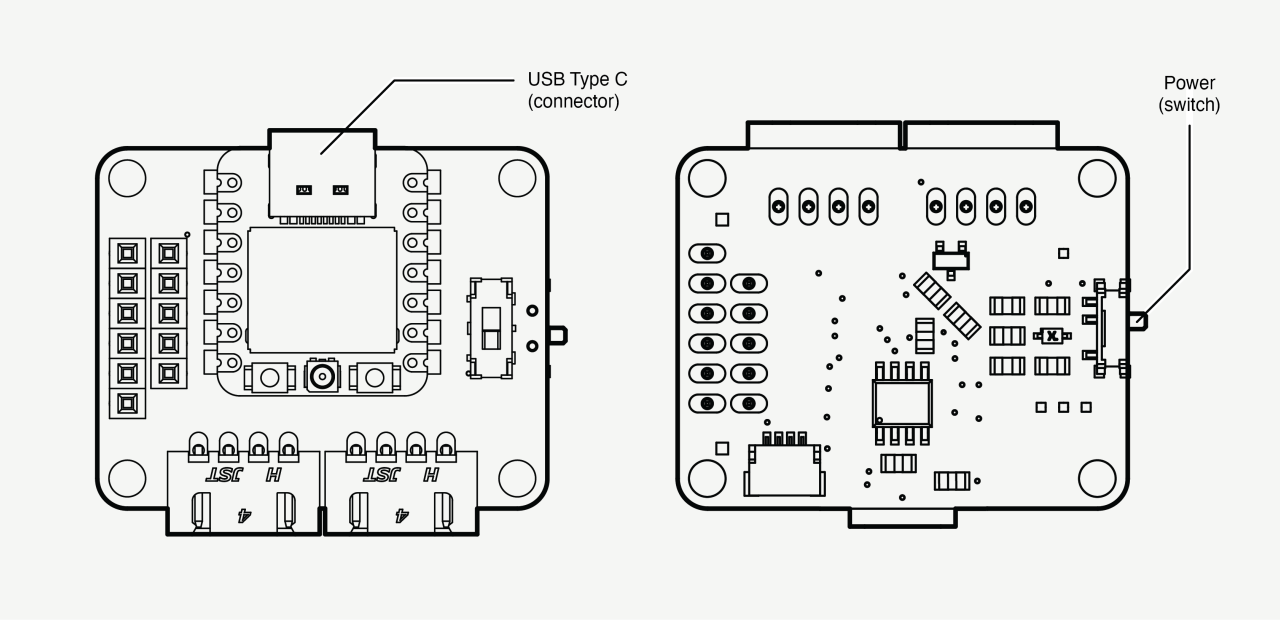
In the vehicle

Node

The sensors are configured as nodes on the CAN bus, meaning they each are paired with a small computer board. The latter acts as the middleman between the sensor's interfacing protocol (i.e. the language the sensor talks in) and the CAN bus (i.e. what the telemetry system talks in).

The node computer hardware diagram is presented in the figure below.





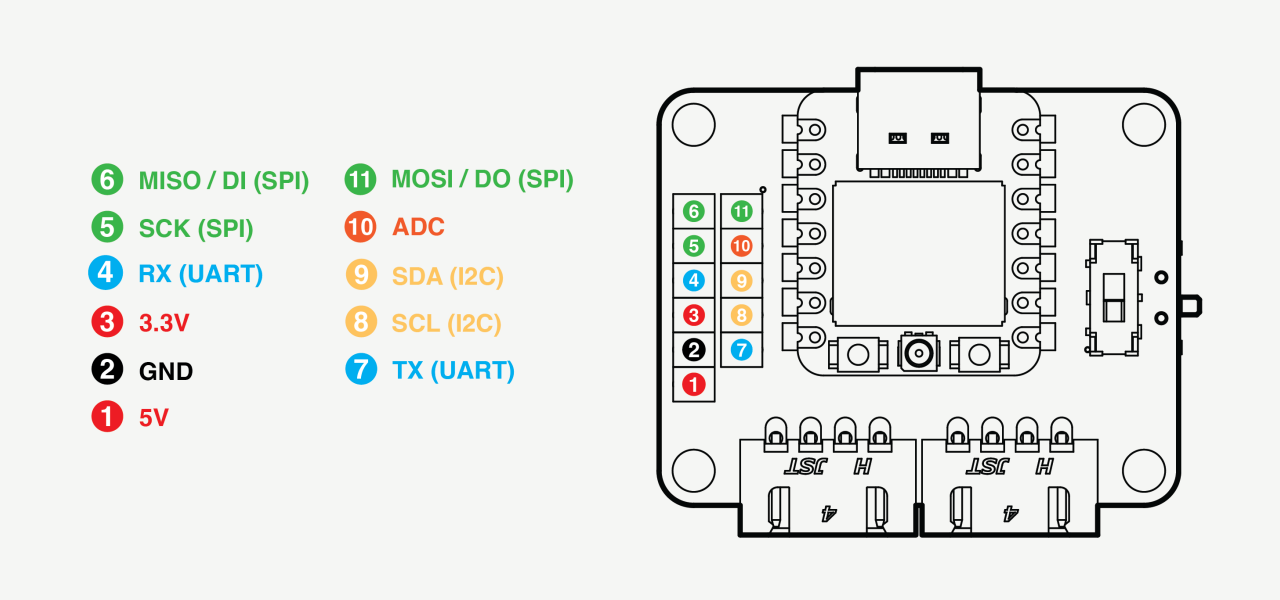
Technical diagrams of the node computer

Technical specs

{% tabs %} {% tab title="Tech specs" %}

- The nodes use a ESP32 microcontroller (S3 or C3 models) from Espressif, manufactured by Seeed (XIAO ESP32C3 / XIAO ESP32S3) (top surface).
- It uses two rows of 6 and 5 female headers (top surface) to connect to the sensor shield.
- A Qwiic / Stemma QT connector to connect to sensor breakout boards directly (no shield).
- They are equipped with TCAN1462V-Q1 CAN transceivers (bottom surface) that make use of the MCU's native TWAI controllers.
- They have two daisy-chained 4-pin JST XH connectors (top surface) for the data transmission CAN High / Low, and the power pins from the Receiver +5V and GND.
- The slide switch on the top surface is the CAN termination switch, connecting the CAN High and Low lines by a 120 Ohm resistor.
- The CAN lines also have footprint for capacitors in series and parallel (bottom surface) to help if filtering is needed on the bus. {% endtab %}

{% tab title="Pinout" %}



{% hint style="info" %} This pinout diagram is valid for the XIAO ESP32-C3 microcontrollers, but some pins also double as analog pins on the XIAO ESP32-S3 (see Table 1), marked as A<Number> (e.g. A5). {% endhint %}

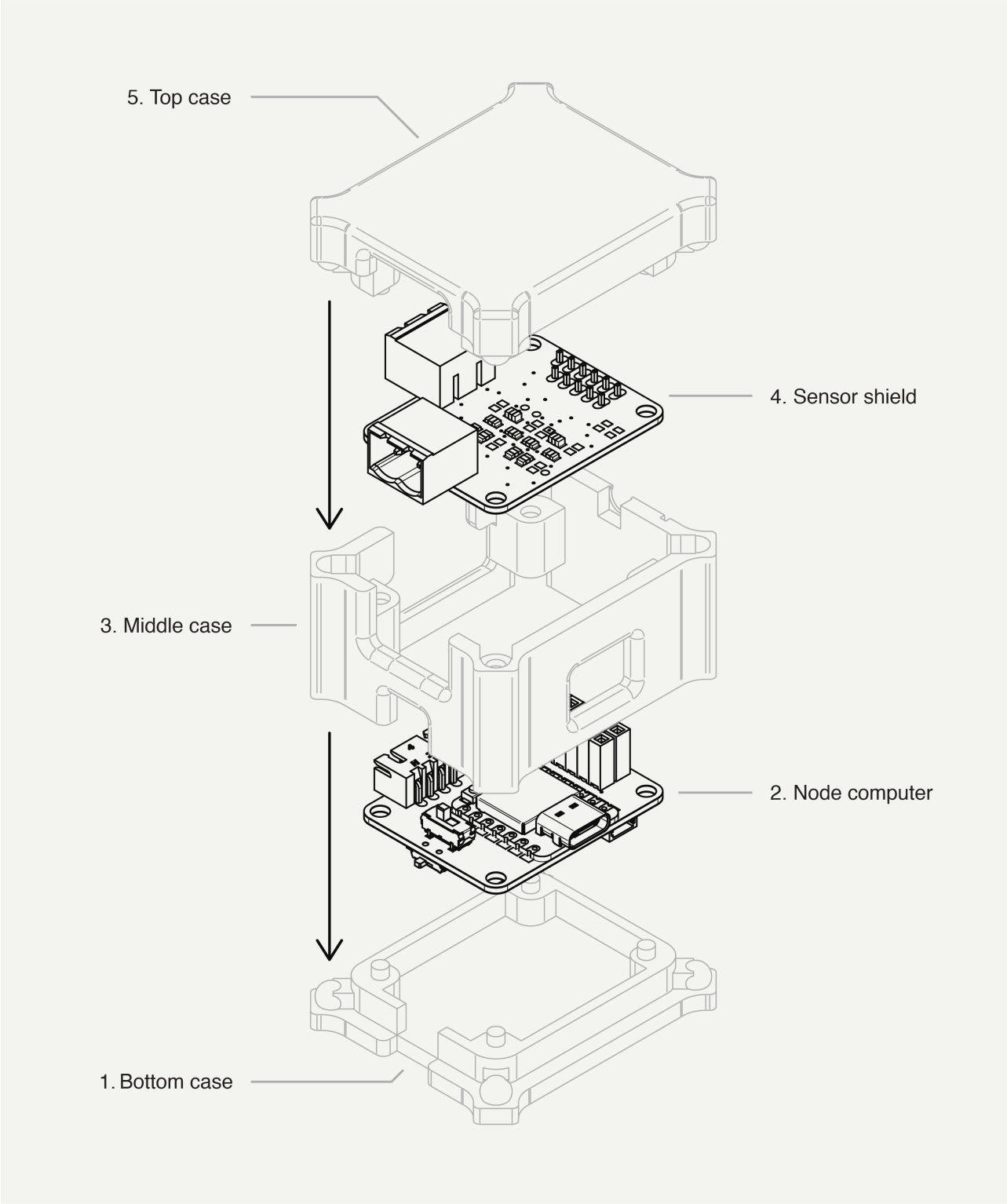
pin #	C3 pinout	S3 pinout
1	5V	5V
2	GND	GND
3	3.3V	3.3V
4	D7 / RX / GPIO20	D7 / RX / GPIO44
5	D8 / SCK / GPIO8	D8 / A8 / SCK / GPIO7
6	D9 / MISO / GPIO9	D9 / A9 / MISO / GPIO8
7	D6 / TX / GPIO21	D6 / TX / GPIO43
8	D5 / SCL / GPIO7	D5 / A5 / SCL / GPIO6
9	D4 / SDA / GPIO6	D4 / A4 / SDA / GPIO5
10	D3 / A3 / GPIO5	D3 / A3 / GPIO4
11	D10 / MOSI / GPIO10	D10 / A10 / MOSI / GPIO9
CAN_TX	GPIO_NUM_3	GPIO_NUM_2
CAN_RX	GPIO_NUM_4	GPIO_NUM_3
CAN_Standby	GPIO2	GPIO1

{% endtab %} {% endtabs %}

Setup

Hardware

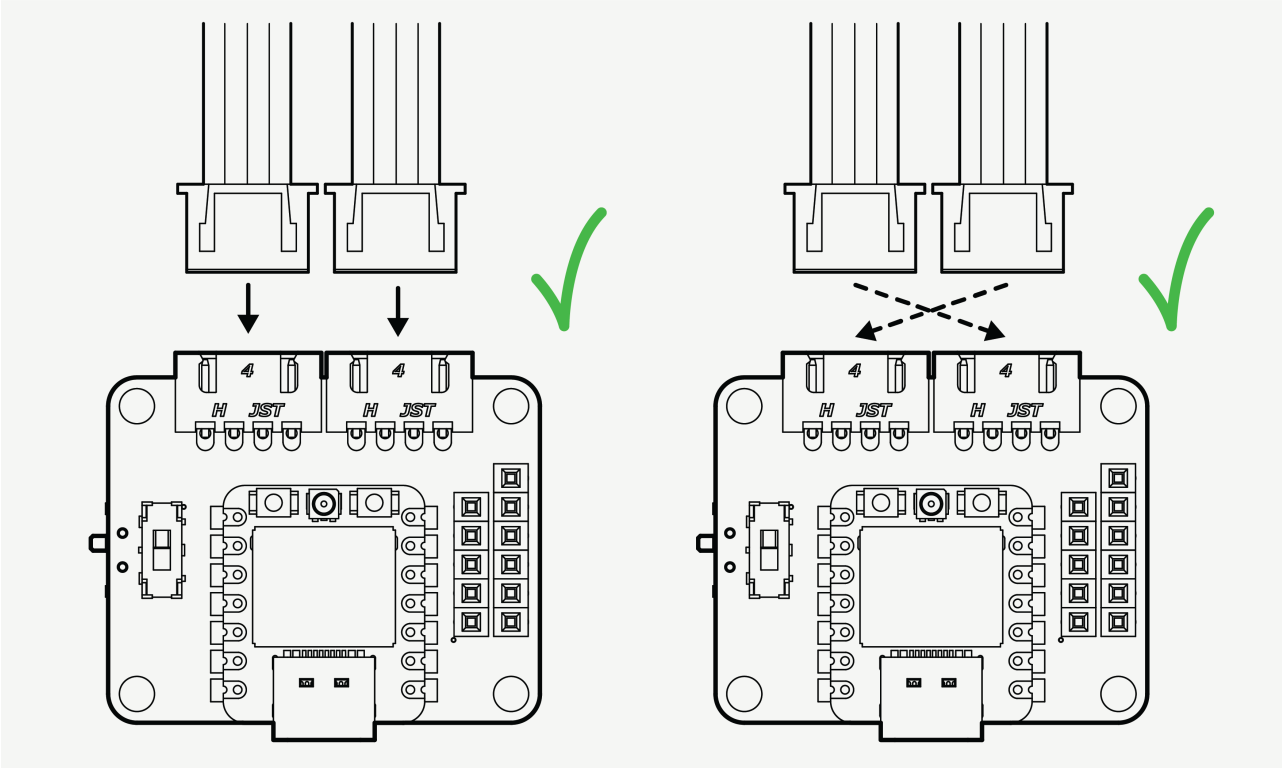
1. Assembling the node case (in ascending order)



{% hint style="info" %} You can also connect sensor breakout boards that feature a Qwiic or Stemma QT connector from Adafruit and Sparkfun directly to the similar connector on the bottom surface. This is the case for the GPS node, which uses a Sparkfun breakout board. {% endhint %}

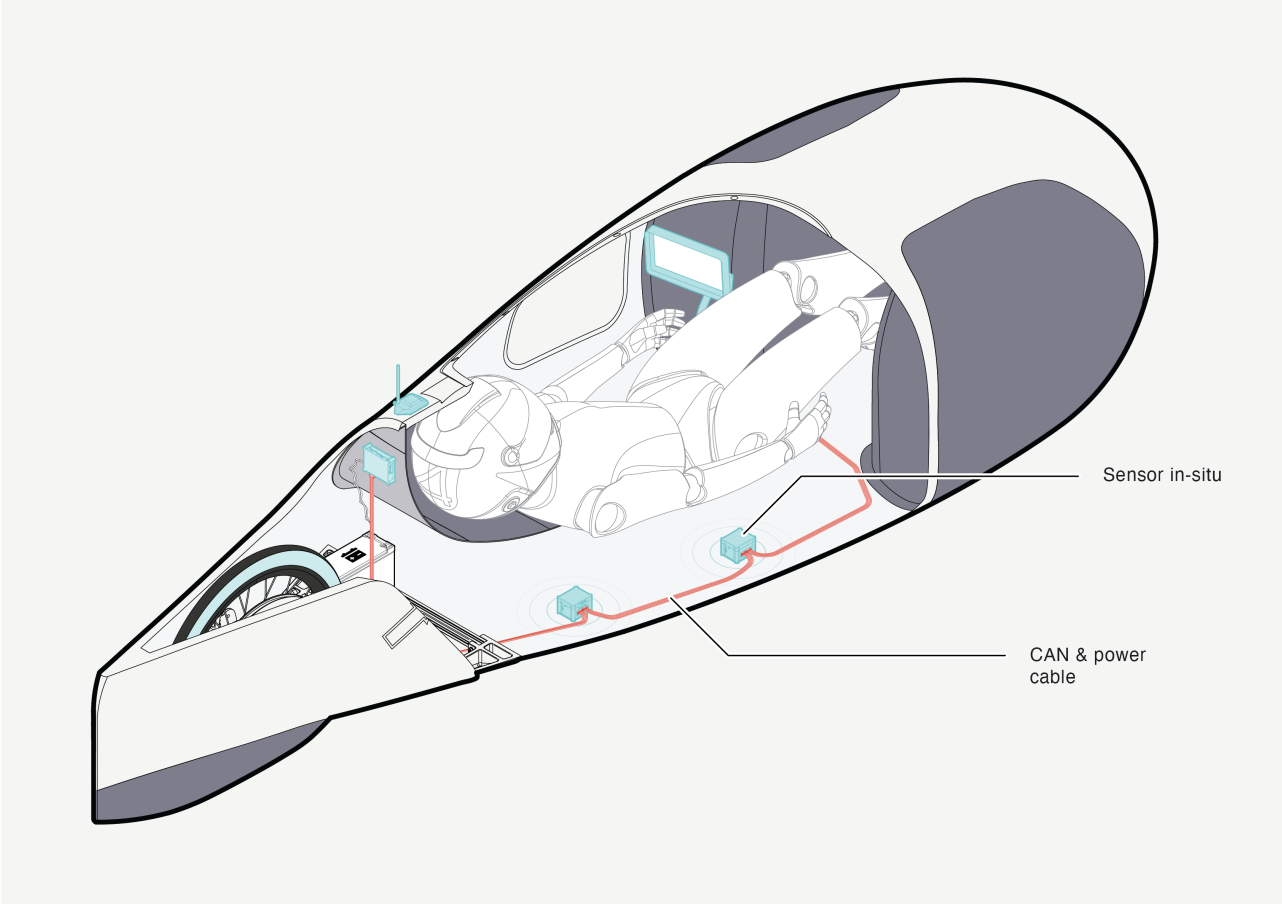
2. Attaching the node to the CAN/Power bus

Connecting to the CAN and Power bus is easy. There are two 4-pin JST XH connectors opposite to the USB-C connector, which are connected in parallel for you to daisy-chain the whole system. This means that there's no wrong way to connect it, just attach one cable to the bus, and if you want to add a new sensor, just attach another cable in the other connector and into the new node.

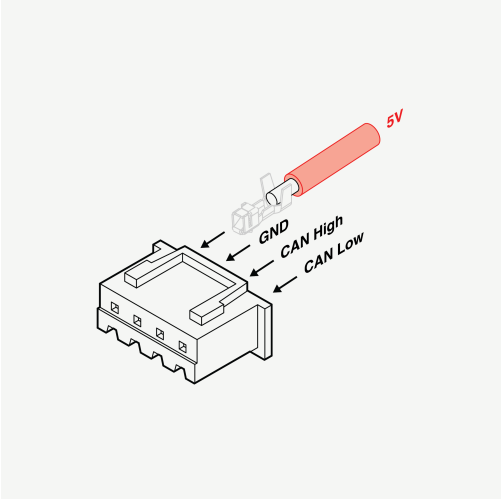


3. Installing in the vehicle

Place the node wherever it is needed in the vehicle.



{% hint style="info" %} If you need to make a new cable, make sure to use the proper connections as seen in the figure below. {% endhint %}



CAN & Power cable wiring diagram

Software

1. Installing the integrated development environment (IDE)

The system works similar to most Arduino projects, as it has been coded in C++ using the Arduino IDE.

Any IDE will do the job, however we recommend using the following:

Arduino IDE is the easiest to use, has plenty of documentation and forum help and is sufficient almost 99% of the time.

Platform IO (through VSCode) is essentially a pro version, where you have more control but requires a manual setup.

ESP-IDF is the MCU's native IDE, which gives more liberty in using all its functions.

2. ESP board definitions

The ESP boards (MCU) are not defined by default in the Arduino IDE. You will have to add them initially to be able to connect to the nodes and others.

To do this in the Arduino IDE, navigate to **File > Preferences**, and fill **"Additional Boards Manager URLs"** with the url below:
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json

Navigate to **Tools > Board > Boards Manager...**, type the keyword "**esp32**" in the search box, select the latest version of **esp32**, and install it.

3. Installing the node library

Download the two files `Node.h` and `Node.cpp` from the Github repository in the Node folder.

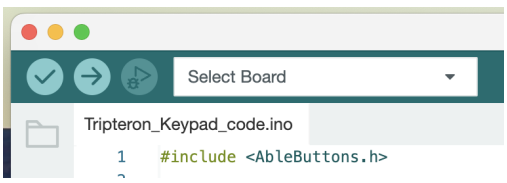
To add the node library to your arduino sketch[^1], simply add the two files in your sketch folder.

▶ Details
Example folder organisation

```
example_sketch (folder)
-> Node.cpp
-> Node.h
-> example_sketch.ino (arduino sketch)
```

4. Connecting to the node computer board

To connect the node computer to the arduino sketch, select the top-left text box **Select Board**.



For C3 models - XIAO_ESP32C3

For S3 models - XIAO_ESP32S3

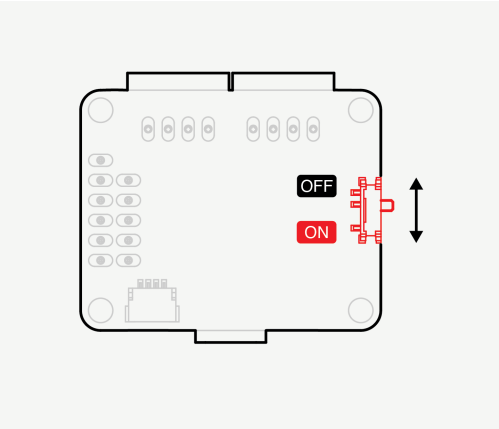
How to use

Find the quick guide in the overview section.

{% hint style="info" %} You can add up to 70+ nodes onto the CAN bus, however limiting the number of sensors to <15 is recommended (i.e. limit the different messages competing on the CAN bus). {% endhint %}

Steps

1. Switch the node ON.



2. Assemble the node (sensor, computer and case).
3. Connect a CAN cable to one of the JST sockets to add the node on the CAN bus.
4. If the node is added at the end of the bus, switch the CAN termination ON.

{% hint style="info" %} After the system is powered, the node is powered if you see a LED lighting up. {% endhint %}

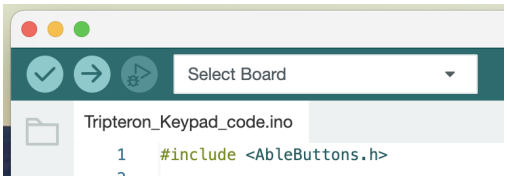
{% hint style="warning" %} Later versions of the node computer boards have a slide switch on the bottom, which need to be activated for the power rail to go through the board.

Be careful because earlier versions have reversed ON/OFF writings by mistake, so if in doubt test it outside the vehicle, and if you see a blinking red/yellow light coming from the node, it is powered. {% endhint %}

Troubleshooting

If it seems as though the Receiver is not seeing the sensor messages, or something else is not working as it should:

1. Disassemble the node.
2. Connect the node computer to your laptop through USB.
3. Open Arduino IDE, and check that you have the correct board definition in the top-left text box **Select Board**.



▶ Details
No board appearing

If you don't see a board appearing, then the MCU is not powered, or the serial connection has been severed. This can be because,

- The USB cable is not properly connected either to the node or laptop
- The MCU is not receiving power, which can be checked with a multimeter on the MCU pins to see if it detects a voltage. This can be due to a severed power connection on the circuit.
- The serial connection is severed, which can be because
  - the USB connector on the MCU has been damaged,
  - or because something (e.g. static electricity) has caused your laptop to disconnect the USB port (safety mode). It happens that connection is re-established after some time.
    - You can first try to reset the board by clicking the **RESET BUTTON** once while the board is connected to your PC. If that does not work, hold the **BOOT BUTTON**, connect the board to your PC while holding the **BOOT** button, and then release it to enter **bootloader mode**.

▶ Details
The wrong board is appearing

If the wrong board is appearing, that is a good sign because it recognises something.

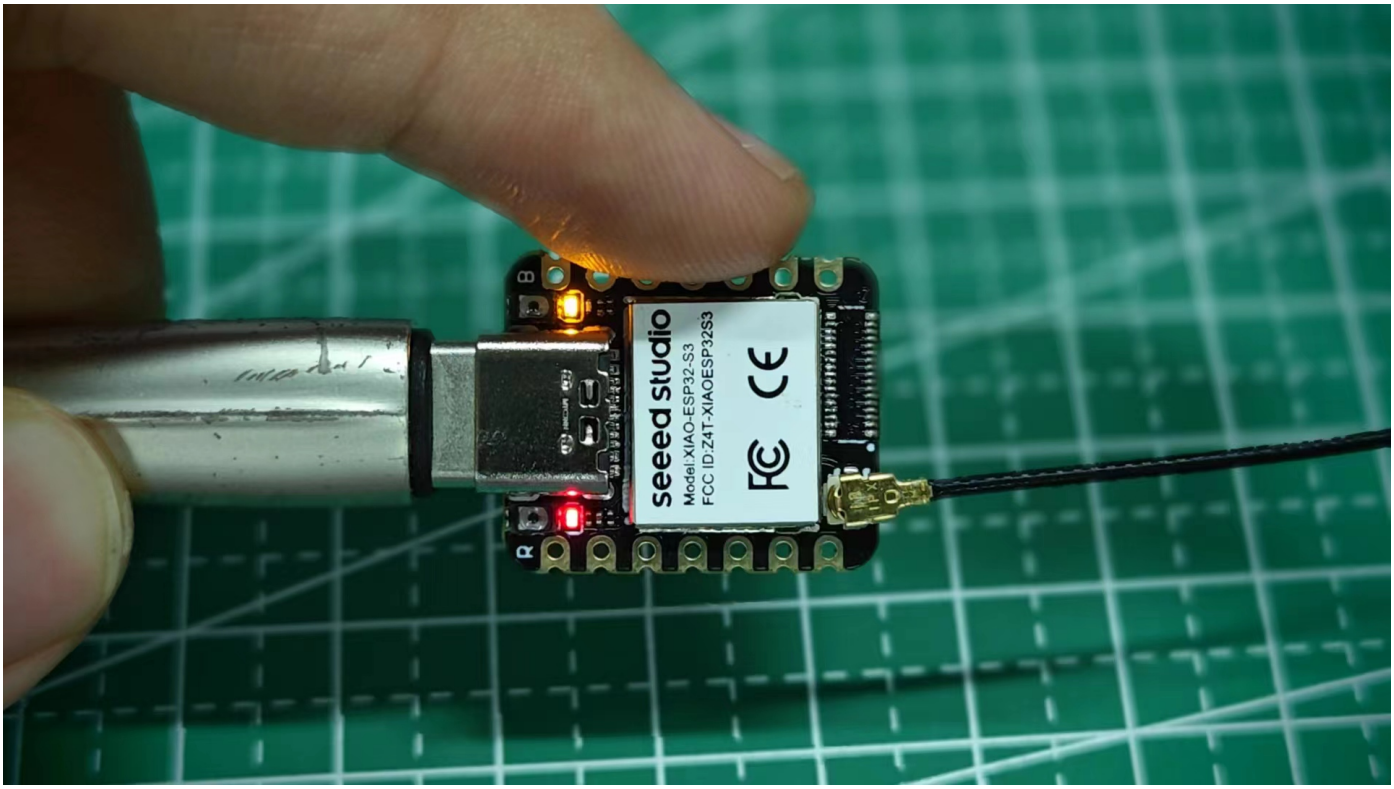
You can open the text box, click on select **other board and port...** and search for the appropriate board.

The boards are,

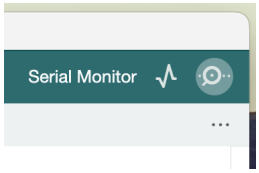
For C3 models - XIAO_ESP32C3

For S3 models - XIAO_ESP32S3

S3 models are identifiable by the writing on the sticker and the presence of solderable pins on the bottom, as seen on the right side of the image below.



4. Open the serial monitor, in the top-right corner.



This will present what is output on the serial port by the board, which you can use to deduce where the issues are coming from.

e.g. it can display "sensor not found on I2C bus" even when the sensor board is connected meaning that there is a connection issue between the node computer and sensor board).

{% hint style="info" %} The serial output will only output things that it has been coded to output, so it is your responsibility to leave serial print commands in your code for debugging. {% endhint %}

Development

Node Library - to code a node

{% tabs %} {% tab title="API reference" %}

- The `Node` class methods provide a high-level interface for interacting with a CAN bus network, abstracting away the complexities of direct hardware manipulation.
- The `VariableType` enumeration and `ExpectedMessage` structure facilitate the definition and handling of expected messages within the CAN network.

Usage Notes

- `uint32_t id`: The identifier of the expected message.
- `VariableType var1Type`: The type of the first variable in the message payload.
- `VariableType var2Type`: The type of the second variable in the message payload.

A structure representing an expected message with its identifier and variable types:

`ExpectedMessage`

- `INT32`: 32-bit signed integer.
- `UINT32`: 32-bit unsigned integer.
- `FLOAT`: Floating-point number.
- `NONE`: No variable (used when there is only one variable).

An enumeration defining the possible types of variables that can be included in a CAN message:

`VariableType`

12. `void initializeCANBus(gpio_num_t canRxPin, gpio_num_t canTxPin, uint8_t canStdbyPin, bool listenMode)`
    - Initialises the CAN bus with the specified pins and mode.
13. `template <typename T> void convertToBytes(T value, byte *buffer)`
    - Converts a value of type `T` to a byte array.

Private Methods

1. `void begin(gpio_num_t canRxPin = GPIO_NUM_NC, gpio_num_t canTxPin = GPIO_NUM_NC, uint8_t canStdbyPin = 0, bool listenMode = false)`
    - Initialises the CAN bus with the specified pins and mode.
2. `void initializeMessage(uint32_t id, uint8_t dataLength)`
    - Initialises a message container with the specified ID and data length.
3. `void deleteMessage(uint32_t id)`
    - Deletes a message from the `messages` container.
4. `template <typename T1, typename T2 = std::nullptr_t> void updateMessageData(uint32_t id, T1 var1, T2 var2 = nullptr)`
    - Updates the data of a message with the given ID in the Node's message map.
5. `void transmitAllMessages(bool lowPowerMode = false, uint32_t sleepDurationMs = 1000)`
    - Transmits all messages in the Node's message map to the CAN bus.
6. `void transmitMessage(uint32_t id, TickType_t ticks_to_wait = pdMS_TO_TICKS(1000))`
    - Transmits a specific message with the specified ID and waits for the specified number of ticks.
7. `void addExpectedMessage(uint32_t id, VariableType var1Type)`
    - Adds a single expected message with only one variable.
8. `void addExpectedMessage(uint32_t id, VariableType var1Type, VariableType var2Type)`
    - Adds an expected message with two variables.
9. `void addExpectedMessages(const std::vector<ExpectedMessage> &messages)`
    - Adds a collection of expected messages to the internal map.
10. `std::pair<uint32_t, std::pair<String, String>> parseReceivedMessage()`
    - Parses the received CAN message and returns the message ID and string representations of the first and second variables in the message payload.
11. `void displayLatestMessageData()`
    - Displays the data of the latest message in the Node's message map.

Public Methods

Creates an instance of the `Node` class.

```
Node();
```

Constructor

`Node`

Classes and Structures

The `Node` library provides a framework for interacting with a CAN bus network using the ESP-IDF framework on ESP32 devices. It includes functionalities for initializing the CAN bus, sending and receiving messages, and managing message expectations.

The `Node` library provides a framework for interacting with a CAN bus network using the ESP-IDF framework on ESP32 devices. It includes functionalities for initializing the CAN bus, sending and receiving messages, and managing message expectations.

Classes and Structures

`Node`

Constructor

```
Node();
```

Creates an instance of the `Node` class.

Public Methods

1. `void begin(gpio_num_t canRxPin = GPIO_NUM_NC, gpio_num_t canTxPin = GPIO_NUM_NC, uint8_t canStdbyPin = 0, bool listenMode = false)`
    - Initialises the CAN bus with the specified pins and mode.
2. `void initializeMessage(uint32_t id, uint8_t dataLength)`
    - Initialises a message container with the specified ID and data length.
3. `void deleteMessage(uint32_t id)`
    - Deletes a message from the `messages` container.
4. `template <typename T1, typename T2 = std::nullptr_t> void updateMessageData(uint32_t id, T1 var1, T2 var2 = nullptr)`
    - Updates the data of a message with the given ID in the Node's message map.
5. `void transmitAllMessages(bool lowPowerMode = false, uint32_t sleepDurationMs = 1000)`

    - Transmits all messages in the Node's message map to the CAN bus.
6. `void transmitMessage(uint32_t id, TickType_t ticks_to_wait = pdMS_TO_TICKS(1000))`
    - Transmits a specific message with the specified ID and waits for the specified number of ticks.
7. `void addExpectedMessage(uint32_t id, VariableType var1Type)`
    - Adds a single expected message with only one variable.
8. `void addExpectedMessage(uint32_t id, VariableType var1Type, VariableType var2Type)`
    - Adds an expected message with two variables.
9. `void addExpectedMessages(const std::vector<ExpectedMessage> &messages)`
    - Adds a collection of expected messages to the internal map.
10. `std::pair<uint32_t, std::pair<String, String>> parseReceivedMessage()`
    - Parses the received CAN message and returns the message ID and string representations of the first and second variables in the message payload.
11. `void displayLatestMessageData()`
    - Displays the data of the latest message in the Node's message map.

Private Methods

12. `void initializeCANBus(gpio_num_t canRxPin, gpio_num_t canTxPin, uint8_t canStdbyPin, bool listenMode)`
    - Initialises the CAN bus with the specified pins and mode.
13. `template <typename T> void convertToBytes(T value, byte *buffer)`
    - Converts a value of type `T` to a byte array.

`VariableType`

An enumeration defining the possible types of variables that can be included in a CAN message:

- `INT32`: 32-bit signed integer.
- `UINT32`: 32-bit unsigned integer.
- `FLOAT`: Floating-point number.
- `NONE`: No variable (used when there is only one variable).

`ExpectedMessage`

A structure representing an expected message with its identifier and variable types:

- `uint32_t id`: The identifier of the expected message.
- `VariableType var1Type`: The type of the first variable in the message payload.
- `VariableType var2Type`: The type of the second variable in the message payload.

Usage Notes

- The `Node` class methods provide a high-level interface for interacting with a CAN bus network, abstracting away the complexities of direct hardware manipulation.
- The `VariableType` enumeration and `ExpectedMessage` structure facilitate the definition and handling of expected messages within the CAN network. {% endtab %}

{% tab title="Example sketch" %} This example code initialises the CAN messages in the setup, and then in the loop updates them and sends them onto the bus.

```
#include "Node.h"

/**
 * Pin Configuration for XIAO ESP32-C3 and XIAO ESP32-S3
 *
 * This section defines the GPIO pins used for TX, RX, and controlling
 * the sleep mode of the CAN transceiver chip based on the specific
 * board variant being used. Ensure that you select the correct pin
 * configuration for your hardware setup.
 *
 * TX_GPIO_NUM: Transmit Pin
 * - XIAO ESP32-C3: GPIO_NUM_3
 * - XIAO ESP32-S3 (e.g. Screen Node): GPIO_NUM_2
 *
 * RX_GPIO_NUM: Receive Pin
 * - XIAO ESP32-C3: GPIO_NUM_4
 * - XIAO ESP32-S3 (e.g. Screen Node): GPIO_NUM_3
 *
 * sleepPin: Sleep Control Pin for the CAN Transceiver
 * - XIAO ESP32-C3: 2
 * - XIAO ESP32-S3: 1
 *
 * Note:
 * - Make sure to define the correct board type before setting these pins.
 * - These configurations are critical for proper communication over the CAN bus.
 */

Node node;
void setup()
{
  // Begin serial communication at a baud rate of 115200.
```

```
  // This is necessary to print messages to the serial monitor.
  Serial.begin(115200);

  // Delay for 1 second to allow the serial monitor to initialize properly.
  // This is necessary to avoid interference with serial printing.
  delay(1000);

  // Initialize the node with the specified RX and TX pins, a standby pin, and a listen mode flag.
  // The node will use the specified pins for CAN communication.
  // The standby pin is used to control the CAN transceiver.
  // The listen mode flag determines whether the node will listen for messages on the CAN bus.
  node.begin(GPIO_NUM_4, GPIO_NUM_3, 2);

  // Initialize a message container with the specified ID and data length.
  // This function is called once per message ID that is used (i.e. to be sent or received).
  // The message container is used to store the message ID and the data associated with the message.
  node.initializeMessage(10, 8);

  // Initialize another message container with another ID and data length.
  node.initializeMessage(14, 4);

  // Display the data of the latest message in the Node's message map.
  // This function iterates through the messages in the Node's message map to find
  // the message with the highest ID, which is assumed to be the latest message.
  // It then prints the ID and data of the latest message.
  node.displayLatestMessageData();

  // Add multiple expected messages.
  // This vector contains ExpectedMessage objects that represent the expected messages received by the node.
  // Each ExpectedMessage object has an ID and two variable types.
  std::vector<ExpectedMessage> messages = {
      {20, INT32, FLOAT}, // Example of a message with two variables
      {24, UINT32, NONE}  // Example of a message with only one variable
  };

  // Add the expected messages to the Node's internal map of expected messages.
  node.addExpectedMessages(messages);
}

void loop()
{
  /*
   * In the example loop, we update and transmit messages.
   * We update message 10 with sine values of i and -i.
   * We update message 14 with the value -14.0.
   * We transmit all messages.
   * We delay for 1 second before the next iteration.
   */
  for (int i = 0; i < 314; i++)
  {
    // Update message 10 with sine values of i and -i.
    node.updateMessageData(10, (sin((float)i / 100)), (sin((float)-i / 100)));

    // Update message 14 with the value -14.0.
    node.updateMessageData(14, (float)-14);

    // Transmit all messages.
    node.transmitAllMessages(false);

    // Delay for 1 second before the next iteration.
    delay(1000);
  }
}
```

{% endtab %}

{% tab title="Example screen" %} This example code receives CAN messages on the bus and hypothetically updates the screen accordingly.

```
#include "Node.h"

/**
 * Pin Configuration for XIAO ESP32-C3 and XIAO ESP32-S3
 *
 * This section defines the GPIO pins used for TX, RX, and controlling
 * the sleep mode of the CAN transceiver chip based on the specific
 * board variant being used. Ensure that you select the correct pin
 * configuration for your hardware setup.
 *
 * TX_GPIO_NUM: Transmit Pin
```

```
 * - XIAO ESP32-C3: GPIO_NUM_3
 * - XIAO ESP32-S3 (e.g. Screen Node): GPIO_NUM_2
 *
 * RX_GPIO_NUM: Receive Pin
 * - XIAO ESP32-C3: GPIO_NUM_4
 * - XIAO ESP32-S3 (e.g. Screen Node): GPIO_NUM_3
 *
 * sleepPin: Sleep Control Pin for the CAN Transceiver
 * - XIAO ESP32-C3: 2
 * - XIAO ESP32-S3: 1
 *
 * Note:
 * - Make sure to define the correct board type before setting these pins.
 * - These configurations are critical for proper communication over the CAN bus.
 */

Node node; // Create an instance of the Node class

// The setup function is called once at startup
void setup() {
  // Initialize serial communication at a baud rate of 115200.
  Serial.begin(115200);

  // Introduce a delay of 1 second to allow for serial printing.
  // This delay is necessary to avoid issues with the serial output.
  delay(1000);

  // Initialize the CAN node with the specified pins and configuration.
  // The CAN_RX pin is GPIO_NUM_4, CAN_TX pin is GPIO_NUM_3,
  // CAN_STBY pin is 2, and CAN_LISTEN_MODE is set to false.
  // The CAN_RX and CAN_TX pins are specific to the XIAO ESP32-C3 and XIAO ESP32-S3.
  // Ensure to configure the correct pin configuration for your hardware setup.
  node.begin(GPIO_NUM_4, GPIO_NUM_3, 2, false);

  // Define multiple expected messages.
  // Each message is represented by an instance of the ExpectedMessage struct.
  // The struct contains the message ID and the variable types of the expected message payload.
  // In this example, we have two messages:
  // - Message ID: 10, Variable Type: FLOAT, FLOAT
  // - Message ID: 14, Variable Type: FLOAT, NONE (no second variable)
  std::vector<ExpectedMessage> messages = {
    { 10, FLOAT, FLOAT },
    { 14, FLOAT, NONE }
  };

  // Add the expected messages to the node.
  // This informs the node about the expected messages to receive.
  node.addExpectedMessages(messages);
}

// Main loop
void loop() {
  // Parse the received message and get the message ID, variable 1, and variable 2
  // The first element of the pair is the message ID, and the second element is a pair of strings for variable 1 and variable 2
  auto parsedMessage = node.parseReceivedMessage();

  // Check if a valid message ID was returned
  if (parsedMessage.first != 0) {
    // Get the message ID, variable 1, and variable 2
    uint32_t messageId = parsedMessage.first;  // The ID of the received message
    String var1 = parsedMessage.second.first;  // The value of the first variable in the message payload
    String var2 = parsedMessage.second.second;  // The value of the second variable in the message payload

    // Print the message ID and the values of the variables on the serial monitor
    printf("Message ID: %u\n", messageId);
    printf("Var1: %s\n", var1.c_str());
    printf("Var2: %s\n", var2.c_str());

    // ADD THE CODE TO UPDATE THE SCREEN HERE
  }
}
```

{% endtab %}

{% tab title="Example sensor" %} This code was made for the voltmeter node, to show the implementation of the node library with a specific sensor (i.e. TI ADS1115 ADC chip).

```
#include "Node.h"
#include <Adafruit_ADS1X15.h>
```

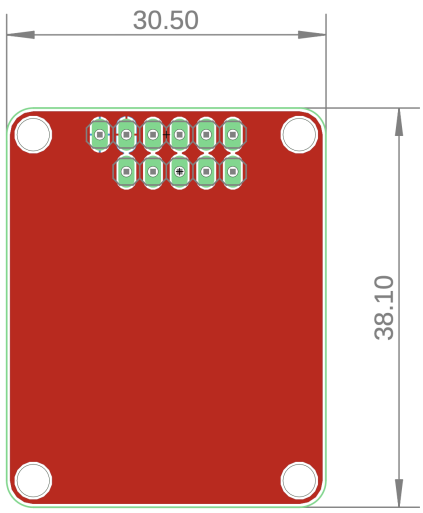
```
/**
 * Pin Configuration for XIAO ESP32-C3 and XIAO ESP32-S3
 *
 * This section defines the GPIO pins used for TX, RX, and controlling
 * the sleep mode of the CAN transceiver chip based on the specific
 * board variant being used. Ensure that you select the correct pin
 * configuration for your hardware setup.
 *
 * TX_GPIO_NUM: Transmit Pin
 * - XIAO ESP32-C3: GPIO_NUM_3
 * - XIAO ESP32-S3 (e.g. Screen Node): GPIO_NUM_2
 *
 * RX_GPIO_NUM: Receive Pin
 * - XIAO ESP32-C3: GPIO_NUM_4
 * - XIAO ESP32-S3 (e.g. Screen Node): GPIO_NUM_3
 *
 * sleepPin: Sleep Control Pin for the CAN Transceiver
 * - XIAO ESP32-C3: 2
 * - XIAO ESP32-S3: 1
 *
 * Note:
 * - Make sure to define the correct board type before setting these pins.
 * - These configurations are critical for proper communication over the CAN bus.
 */

Adafruit_ADS1115 ads; /* Use this for the 16-bit version */

Node node;
void setup()
{
  // Initialize serial communication
  Serial.begin(115200);

  // Delay to prevent issues with serial printing
  delay(1000);

  // Initialize the Node with specific GPIO pins and standby pin
  node.begin(GPIO_NUM_4, GPIO_NUM_3, 2);

  // Initialize messages with specific IDs and data lengths
  node.initializeMessage(10, 8);
  node.initializeMessage(11, 4);

  // The ADC input range (or gain) can be changed via the following
  // functions, but be careful never to exceed VDD +0.3V max, or to
  // exceed the upper and lower limits if you adjust the input range!
  // Setting these values incorrectly may destroy your ADC!
  //                                                                ADS1015  ADS1115
  //                                                                -------  -------
  // ads.setGain(GAIN_TWOTHIRDS);  // 2/3x gain +/- 6.144V  1 bit = 3mV      0.1875mV (default)
  // ads.setGain(GAIN_ONE);        // 1x gain   +/- 4.096V  1 bit = 2mV      0.125mV
  ads.setGain(GAIN_TWO); // 2x gain   +/- 2.048V  1 bit = 1mV      0.0625mV
  // ads.setGain(GAIN_FOUR);  // 4x gain   +/- 1.024V  1 bit = 0.5mV    0.03125mV
  //  ads.setGain(GAIN_EIGHT);      // 8x gain   +/- 0.512V  1 bit = 0.25mV   0.015625mV
  //  ads.setGain(GAIN_SIXTEEN);    // 16x gain  +/- 0.256V  1 bit = 0.125mV  0.0078125mV

  // Initialize the ADS1115 ADC
  if (!ads.begin())
  {
    Serial.println("Failed to initialize ADS.");
    while (1)
      ;
  }

  // Delay after initialization
  delay(500);

  // Set the data rate for the ADS1115 ADC
  ads.setDataRate(RATE_ADS1115_8SPS);
}

void loop()
{
  // Read ADC values for differential inputs 0-1 and 2-3
  int32_t results01 = ads.readADC_Differential_0_1();
  int32_t results23 = ads.readADC_Differential_2_3();

  // Map the ADC values to the desired range
  float mapped01 = map(results01, 5100, 32640, 500, 3200) / 1E2;
  // float mapped23 = map(results23, 5100, 32640, 500, 3200) / 1E2;
```

```
    // float mappedSOC = map((int32_t)(mapped23 * 10), 0, 327, 0, 100);

    // printf("0-1: %d  2-3: %d\n", results01, results23);
    printf("%d,%d,\n", results01, results23);
    printf("%f,\n", mapped01);

    // Update the message data
    // node.updateMessageData(10, mappedSOC, mapped23);
    // node.updateMessageData(11, mapped01);
    // node.transmitAllMessages();

    delay(500);
  }
```

{% endtab %} {% endtabs %}

Design a new sensor

The sensors come in all forms, but only one shape. They all have a standardised footprint that you can build on to make new sensors while still remaining compatible with the node computer.



Template sensor shield footprint (Unit: mm)

The footprint has four M3 sized holes for support, and two rows of pin headers placed just like the node computer pinout diagram.

The fusion PCB file can be found on the Github repository under **Hardware**, and has the pin header sensor connections already pre-defined to simplify the design.

Some general help and thoughts on selecting a sensor:

- Think about what you want to know performance-wise (e.g. fuel consumption etc.), and search up what kind of sensor best suits your requirements.

You can have multiple sensors that can accomplish the same task, but have different pros and cons.

e.g. to measure velocity, we could have had an RPM sensor or a GPS. GPS was a safer, more absolute solution, and in this case we already needed it for position.

{% hint style="info" %} On big retailers websites (e.g. Mouser, Farnell, DigiKey, etc.) look for:

- the price (unit or bulk?),
- availability (is it end of life?),
- size (does it fit on the board?),
- functionalities (handy reliability mechanisms, etc.) {% endhint %}
- What communication protocol does it use?

Connecting to your sensor depends on the protocol the sensor uses to communicate. These can be Serial-Peripheral-Interface (SPI), Inter-Integrated Communication (I2C), UART or you have some general-purpose input/output (GPIO) pins that you can reprogram based on the ESP32-C3 or ESP32-S3 capabilities. The board also has an input for simple analog voltage reading up to 2.5V, however it is recommended to use the ADC shield instead of the ADC pins as they have lower resolution and impedance (essentially just worse, check out Espressif's documentation).

You can see the pinout of the node computer in the technical specifications.

- Does it have an ECAD footprint available?

Often you are able to find the footprint ready-made by a retailer on their page (Mouser, etc.) or by a hobbyist. This simplifies the design process, however it is recommended to always double check the pinout matches that of the datasheet, as well as the dimensions.

{% hint style="info" %} **Tip:** Find sensors with extensive online coverage (arduino tutorials, pre-existing code, etc.) as well as from breakout board manufacturers (e.g. Adafruit, Sparkfun, Seed). They often open source their electrical diagrams and code. {% endhint %}

When it comes to wiring the sensor on the board and adding the necessary components around the chip, it is best to follow the datasheet given by the manufacturer (e.g. Texas Instruments). They often present example schematics showcasing the most common setups.

Most of the sensors developed this year were chosen because they had been covered in online tutorials, and thus had already proven code and easy-to-use libraries for them. This makes it easy to pair with our node library, as you can see in the sensor example.

Build a node computer

If you need to order and assemble a new node computer, you can use the gerber files provided in the Github repository, and upload them to JLCPCB.

{% tabs %} {% tab title="Schematics" %} **Electrical Diagram**

{% file src=".gitbook/assets/Node computer schematic.pdf" %}

PCB footprint

{% file src=".gitbook/assets/Node PCB Footprint.pdf" %} {% endtab %}

{% tab title="3D Viewer" %} {% embed url="https://a360.co/3WZYaDd" %} {% endtab %}

{% tab title="Parts list (BOM)" %}

Qty	Value	Device	Package	Parts	Description	Manufacturer	Link
1		PINHD-1X5	1X05	JP3	PIN HEADER	n/a	n/a
1		PINHD-1X6	1X06	JP1	PIN HEADER	n/a	n/a
2	0	R_CHIP-0805(2012-METRIC)	RESC2012X65	R3		n/a	n/a
3	0.1 uF	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C1		n/a	n/a
2	60	R_CHIP-0805(2012-METRIC)	RESC2012X65	R1		n/a	n/a
1	CDSOT23-T24CAN	CDSOT23-T24CAN	SOT95P230X117-3N	D1	CAN TVS Diode	Bourns	https://www.mouser.co.uk/ProductDetail/Bour
1	JS102011JCQN	JS102011JCQN	JS102011JCQN	S1	Slide Switches .3A 6VDC SPDT VERT MNT SMT J BEND	C & K COMPONENTS	https://www.mouser.co.uk/ProductDetail/CK/J
2	JST-XH-04-PIN-LONG-PAD	JST-XH-04-PIN-LONG-PAD	JST-XH-04-PACKAGE-LONG-PAD	X1		n/a	n/a
3	NC	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C3		n/a	n/a
1	PCM12SMTR	PCM12SMTR	PCM12SMTR	S3	Slide Switches 0.3A SPDT ON-ON	C & K COMPONENTS	https://www.mouser.co.uk/ProductDetail/CK/P qs=mfFuHy8STfL3qrPSfCHA7w%3D%3D&_gl=
1	PMEG1020EJ	PMEG1020EJ	SOD-323	D4	Schottky Rectifier, 10 V, 2 A, Single, SOD-323, 2 Pins, 460 mV	NEXPERIA	https://www.mouser.co.uk/ProductDetail/Nexp

Qty	Value	Device	Package	Parts	Description	Manufacturer	Link
1	QWIIC_RIGHT_ANGLE	QWIIC_CONNECTORJS-1MM	JST04_1MM_RA	J1	SparkFun I2C Standard Qwiic Connector	n/a	n/a
1	TCAN1462VDRQ1	TCAN1462VDRQ1	SOIC127P600X175-8N	IC2	CAN FD transceiver	Texas Instruments	https://www.mouser.co.uk/ProductDetail/Texas
1	XIAO-ESP32C3	XIAO-ESP32C3	XIAO-ESP32C3-MODULE14P-2.54-21X17.8MM	U1	Seeed Studio XIAO ESP32C3	Seed	https://thepihut.com/products/seeed-xiao-esp

{% endtab %} {% endtabs %}

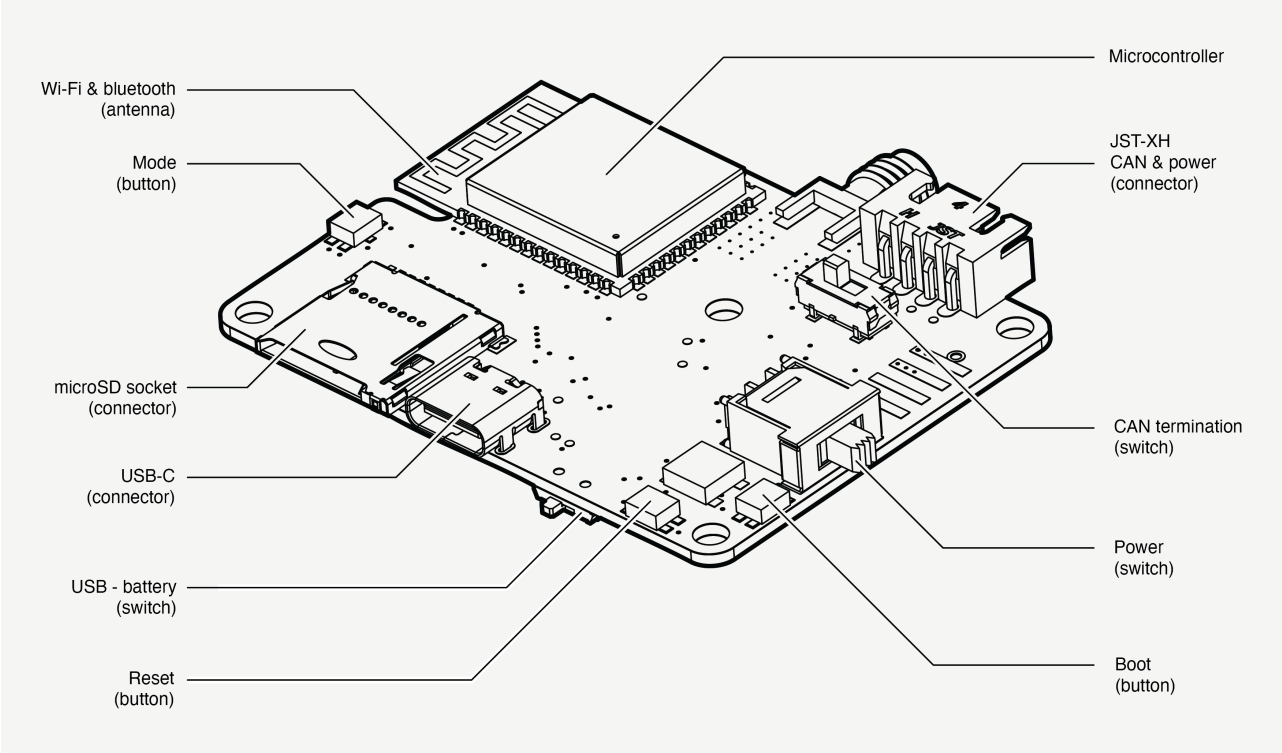
Some tips on soldering the node computer,

- Start with the bottom surface with the slide switch
- Progress to the other small components
- Place large components before placing the smaller ones around it (e.g. CAN transceiver before its capacitors)
- On the top side, place the switch first, then the MCU, then the JST sockets and finally the female pin headers. Note that the pin headers are male on the Fusion 360 project, which is not a problem since male and female have the same PCB footprint.

Receiver

The receiver is a computer board, who's only function is to receive those sensor messages, store them onto a microSD card and handle the radio.

The receiver hardware diagram is presented in the figure below.



Technical diagram of the receiver

Technical specs

{% tabs %} {% tab title="Tech specs" %} The receiver uses a ESP32-S3-WROOM-1-N16R2 module as a microcontroller.

On the top surface it has,

- a microSD socket for datalogging using the ESP's native SD_MMC library
- a USB-C port to connect to a computer
- boot and reset buttons like any Arduino board
- an Adafruit SK6812 RGB LED, programmable with Adafruit's Neopixel Library
- a programmable mode button
- a 4 pin JST-XH connector similar to the node computers for CAN and power
- a CAN termination switch and a power switch to the rest of the system (nodes)

- a Omron XW4K-04A1-V1 board side connector socket to receive 5V from a power source (battery, etc.) connected with a Omron XW4H-04A1 wire side connector plug

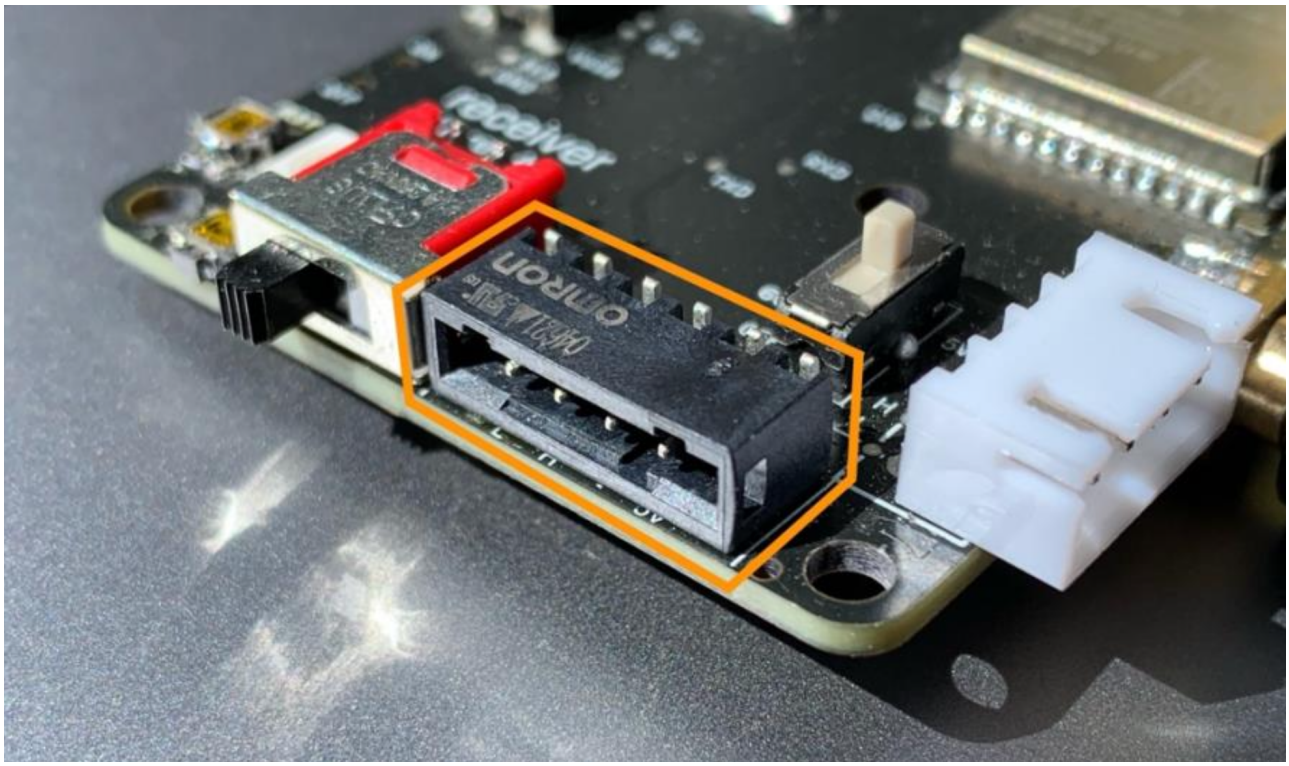


Image of the soldered Omron XW4K-04A1-V1 board side connector socket

On the underside it has,

- a USB - Battery power switch, to disconnect the battery from the receiver MCU (hard reset)
- a 433MHz RFM69HCW radio module
- a male SMA radio antenna connector

It can be programmed and display data on the serial port by connecting to a computer through the USB-C port, and opening an IDE (Arduino IDE, PlatformIO or ESP-IDF). {% endtab %}

{% tab title="Pinout" %}

Pin Description	ESP32-S3-WROOM-1-N16R2
RFM69_DIO_0	GPIO 3
RFM69_DIO_1	GPIO 9
RFM69_DIO_2	GPIO 10
RFM69_RESET	GPIO 48
RFM69_NSS (SPI Chip Select)	GPIO 47
SPI_SCK (RMF69)	GPIO 12
SPI_MOSI (RMF69)	GPIO 11
SPI_MISO (RFM69)	GPIO 13
CAN_TX	GPIO 18
CAN_RX	GPIO 8
CAN_STANDBY	GPIO 35
MODE_BUTTON	GPIO 14
NEOPIXEL_LED	GPIO 21
SD_DAT0	GPIO 6
SD_DAT1	GPIO 5
SD_DAT2	GPIO 17
SD_DAT3	GPIO 16
SD_CMD	GPIO 15

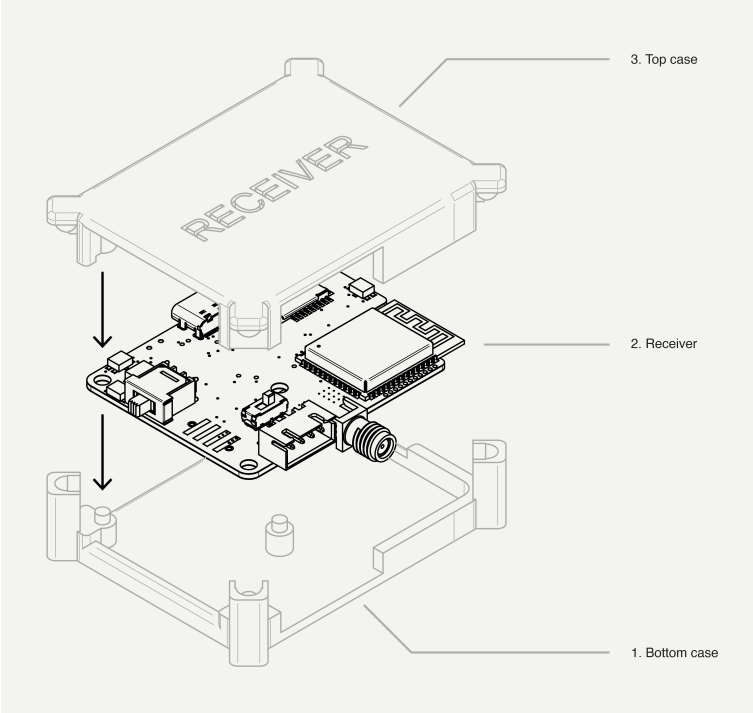
Pin Description	ESP32-S3-WROOM-1-N16R2
SD_CD	GPIO 4

{% endtab %} {% endtabs %}

Setup

Hardware

1. Assembling the receiver case (in ascending order)

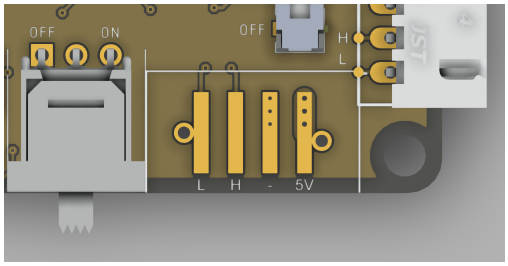


2. Attaching the node to the CAN/Power bus

Just like on the nodes, there is a JST-XH socket to connect the receiver to the CAN and Power bus.

The Omron socket also has CAN lines so that if a power source also has a CAN output (e.g. Battery BMS system outputting CAN messages), they can be attached to the same connector plug.

You can see the connection placement in the centre of the image below, where **L is for CAN_LOW**, **H for CAN_HIGH**, **- for GROUND** and **5V for +5V**.



3. Installing in the vehicle

Just like for a node, you can then place the receiver anywhere in the vehicle.

Software

Similar to in the node software setup.

▶ Details
Similar steps

1. Installing the integrated development environment (IDE)

The system works similar to most Arduino projects, as it has been coded in C++ using the Arduino IDE.

Any IDE will do the job, however we recommend using the following:

Arduino IDE is the easiest to use, has plenty of documentation and forum help and is sufficient almost 99% of the time.

Platform IO (through VSCode) is essentially a pro version, where you have more control but requires a manual setup.

ESP-IDF is the MCU's native IDE, which gives more liberty in using all its functions.

2. ESP board definitions

The ESP boards (MCU) are not defined by default in the Arduino IDE. You will have to add them initially to be able to connect to the nodes and others.

To do this in the Arduino IDE, navigate to **File > Preferences**, and fill **"Additional Boards Manager URLs"** with the url below:
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json

Navigate to **Tools > Board > Boards Manager...**, type the keyword "**esp32**" in the search box, select the latest version of **esp32**, and install it.

3. Installing the receiver library

Download the 15 files from the Github repository in the Receiver folder.

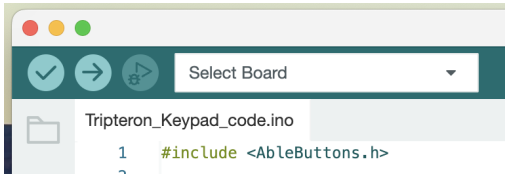
To add the receiver library to your arduino sketch[^2], add all the files in your sketch folder.

▶ Details
Example project organisation

```
example_sketch (folder)
-> Receiver.cpp
-> Receiver.h
-> Config.h
-> SDCardHandler.cpp
-> SDCardHandler.h
-> RadioHandler.cpp
-> RadioHandler.h
-> ModeConfig.cpp
-> ModeConfig.h
-> DataProcessing.cpp
-> DataProcessing.h
-> ConfigFileParser.cpp
-> ConfigFileParser.h
-> CANMessageParser.cpp
-> CANMessageParser.h
-> example_sketch.ino (arduino sketch)
```

4. Connecting to the receiver board

To connect the receiver board to the arduino sketch, select the top-left text box **Select Board** and select **ESP32-S3-USB-OTG**.



1. Assemble the receiver (board and case).
2. Connect a CAN cable to the JST sockets to add the receiver to the CAN bus.
3. Usually the receiver is added at the end of the bus, therefore switch the CAN termination ON.
    1. If the power source connector (Omron) also has a CAN input, then switch the CAN termination OFF. There will need to be a CAN termination at the power source side of the CAN bus.

How to use

Steps

1. Insert the micro-SD card in the socket. See here for more info on the telemetry data configuration.
2. Assemble the node (sensor, computer and case).
3. Connect a CAN cable to one of the JST sockets to add the node on the CAN bus.
4. If the node is added at the end of the bus, switch the CAN termination ON.

Troubleshooting

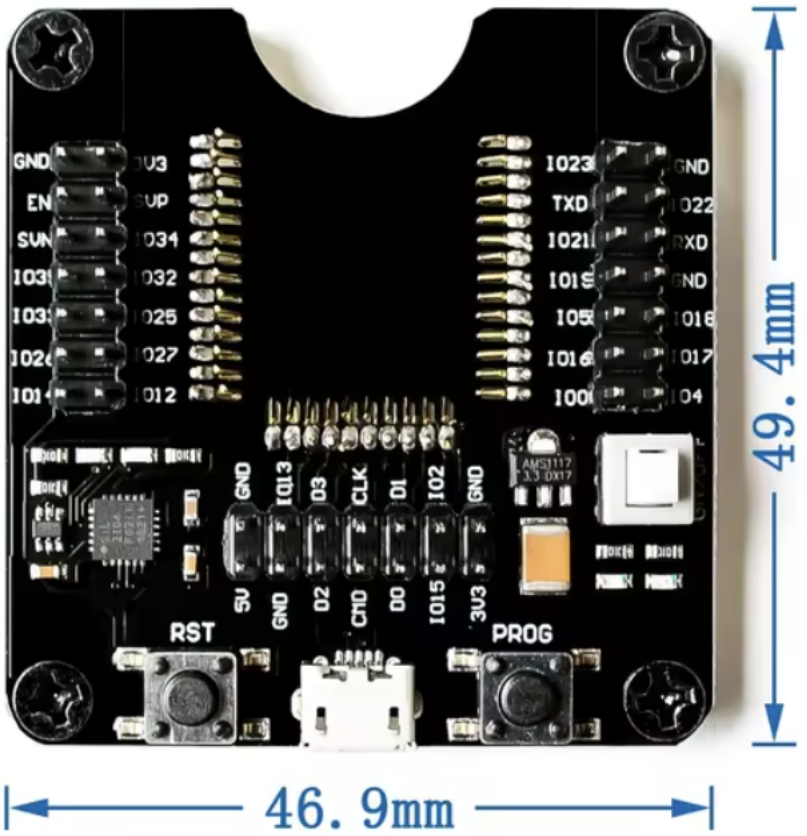
If it seems as though the receiver is not working as intended:

1. Disassemble the receiver.
2. Connect the receiver board to your laptop through USB.
3. Open Arduino IDE, and check that you have the correct board definition in the top-left text box **Select Board**.

▶ Details
No board appearing

If you don't see a board appearing, then the MCU is not powered, or the serial connection has been severed. This can be because,

- The USB cable is not properly connected either to the node or laptop
- The MCU is not receiving power, which can be checked with a multimeter on the MCU 3.3V pins to see if it detects a voltage. This can be due to a severed power connection on the circuit. It is recommended to check the Espressif's chip datasheet in detail.
- The serial connection is severed, which can be because
    - the USB connector on the MCU has been damaged,
    - or because something (e.g. static electricity) has caused your laptop to disconnect the USB port (safety mode). It happens that connection is re-established after some time.
        - You can first try to reset the board by clicking the **RESET BUTTON** once while the board is connected to your PC. If that does not work, hold the **BOOT BUTTON**, connect the board to your PC while holding the **BOOT** button, and then release it to enter **bootloader mode**.
    - a very unlucky reason can be that the MCU chip has failed. This has happened a few times in the past, and can be fixed by carefully desoldering the MCU chip with a hot air gun and heat-resistant Capton tape around other components, to then either replacing with a new chip or de-bricking the "dead" chip by connecting it with a test board module (possibly left in the MechSpace Workshop C drawers).

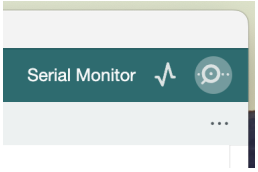


▶ Details
The wrong board is appearing

If the wrong board is appearing, that is a good sign because it recognises something.

You can open the text box, click on select **other board and port...** and search for the appropriate board (ESP32-S3-USB-OTG).

4. Open the serial monitor, in the top-right corner.



This will present what is output on the serial port by the board, which you can use to deduce where the issues are coming from.

{% hint style="info" %} The serial output will only output things that it has been coded to output, so it is your responsibility to leave serial print commands in your code for debugging.

This also means that you can check the receiver sketch and library codebase to find out where from the serial print output is coming from in the code. {% endhint %}

Development

Receiver Library

{% hint style="info" %} The Receiver codebase is quite a convoluted assortment of libraries to separate and classify all the different functions.

There is a handler for everything SD communication related, Radio related, mode button, {% endhint %}

{% tabs %} {% tab title="API Reference" %} The `Receiver` library provides a framework for interacting with a CAN bus network using the ESP-IDF framework on ESP32 devices. It includes functionalities for initialising an SD card, a CAN bus, receiving, processing and storing CAN messages, and transmitting them over radio.

Classes and Structures

`Receiver`

Constructor

```
Receiver::Receiver(const String& configFilePath)
```

Creates a new `Receiver` instance initialized with the specified configuration file path.

Public Methods

1. `void Receiver::begin(bool liveSendingMode)`
   - Initialises the Receiver object by setting up the SD card, radio, configuration parser, and CAN interface. liveSendingMode Flag indicating whether to enable live sending mode.
2. `void Receiver::receiveCAN(SemaphoreHandle_t& dataMutex, bool displayParameters)`
   - Receives a CAN message and handles it if the message is valid.
     - dataMutex is a semaphore handle used to protect access to shared data.
     - the displayParameters Flag indicating whether to display updated parameters after handling the message.
3. `void Receiver::printParameters()`
   - Prints the CAN message parameters stored in the configuration parser.
4. `void Receiver::displayUpdatedParameters()`
   - Displays the updated parameters by iterating over the CAN message configurations and checking for updates.
5. `void transmitAllMessages(bool lowPowerMode = false, uint32_t sleepDurationMs = 1000)`
   - Transmits all messages in the Node's message map to the CAN bus.
6. `void Receiver::initLapFence(float latA, float lngA, float latB, float lngB, float latC, float lngC, float latD, float lngD)`
   - Initialises the lap fence to count laps with the given coordinates.
7. `void Receiver::initRadioFence(float latA, float lngA, float latB, float lngB, float latC, float lngC, float latD, float lngD)`
   - Initialises the radio fence to send telemetry data with the given coordinates.
8. `void Receiver::togglePrintStatements()`
   - Toggles the state of print statements.

Usage Notes

- The `Receiver` class provides a high-level interface for handling CAN messages, radio communication, and SD card operations in a receiver system. It abstracts away the complexities of direct hardware manipulation, making it easier to integrate these functionalities into a larger system.
- Key features include:
  - Initialisation of the receiver system with configurable options
  - Reception and handling of CAN messages
  - Displaying and updating of CAN message parameters
  - Support for both lap fences and radio fences
  - Toggleable print statements for debugging and monitoring
  -

{% endtab %}

{% tab title="Example sketch" %} This example shows the use of the receiver library in the receiver sketch.

```
#include <RHReliableDatagram.h>
#include <Adafruit_NeoPixel.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/twai.h"
#include "esp_system.h"
#include <RH_RF69.h>
#include "esp_err.h"
#include "esp_log.h"
#include <stdlib.h>
#include "SD_MMC.h"
#include <stdio.h>
#include <SPI.h>
#include "FS.h"
#include <map>
#include <EEPROM.h>
```

```
#include "Receiver.h"

#define ESP_EXCEPTION_DECODER

Receiver receiver("/config/README.txt");

SemaphoreHandle_t dataMutex;  // Mutex to protect shared data

void CANHandlerTask(void *pvParameters) {
  for (;;) {
    // Handle button press events here
    receiver.receiveCAN(dataMutex, true);
    //receiver.displayUpdatedParameters();
    vTaskDelay(pdMS_TO_TICKS(0));  // Adjust delay according to your application requirements
  }
}

void setup() {
  Serial.begin(115200);

  delay(100);  // delay to let serial begin have a breather huff huff

  receiver.begin(true);
  receiver.printParameters();

  //receiver.initLapFence(51.5280372, -0.1426583, 51.5280462, -0.1425165, 51.5279792, -0.1425112, 51.5279757, -0.1426627);    //
NW1 3QS Compton Close Park
  //receiver.initRadioFence(51.5280363, -0.1430492, 51.5280545, -0.1422529, 51.5279662, -0.1422228, 51.5279683, -0.1430449);  //
NW1 3QS Compton Close Park

  //receiver.initLapFence(51.4199761, 0.3482651, 51.4200143, 0.3483264, 51.4198693, 0.3488128, 51.4198191, 0.3487607);    //
CycloPark
  //receiver.initRadioFence(51.4205795, 0.3484007, 51.4206552, 0.3484280, 51.4202241, 0.3496966, 51.4201799, 0.3496453);  //
CycloPark

  receiver.initLapFence(43.7708310, -0.0408539, 43.7709265, -0.0406479, 43.7706997, -0.0404429, 43.7706075, -0.0406456);    //
Nogaro track
  receiver.initRadioFence(43.7719449, -0.0423221, 43.7723014, -0.0417674, 43.7712505, -0.0403314, 43.7707955, -0.0409531);  //
Nogaro track

  // Create mutex
  dataMutex = xSemaphoreCreateMutex();
  if (dataMutex == NULL) {
    printf("Failed to create mutex\n");
    // Handle the error appropriately, e.g., stop the program
    while (1)
      ;
  }

  xTaskCreatePinnedToCore(
    CANHandlerTask,    // Task function
    "CANHandlerTask",  // Task name
    4096,              // Stack size (bytes)
    NULL,              // Task parameters
    3,                 // Task priority
    NULL,              // Task handle
    0                  // Core to run the task (0 or 1)
  );
}

void loop() {
}
```

{% endtab %} {% endtabs %}

Telemetry Data Configuration

On the SD card connected to the receiver there is a **config** folder with a **README.TXT** file.

The README file is used to configure the telemetry messages received from nodes and which ones to be sent over radio to the race engineer.

▶ Details
README file format

The format of the configuration is,

```
<Message_name>, <CAN_ID>, <Param1_Name>, <Param1_Unit>, <Param1_type>, <Param1_bits>, <Param2_Name>, <Param2_Unit>,
<Param2_type>,<Param2_bits>
```

For example, for the GPS node which we will name GPS1,

- a CAN bus message ID of 10
- sending latitude in degrees in the first 4 bytes with a variable type of 32-bit integer
- sending longitude in degrees in the last 4 bytes with a variable type of 32-bit integer

Altogether, to add a message from the GPS node into one line,

```
GPS1, 22, Lat, deg, I, 32, Lon, deg, I, 32
```

To specify wether the message is also to be forwarded to the race engineer over radio, you can add an @ sign in front of the line.

```
@GPS1, 22, Lat, deg, I, 32, Lon, deg, I, 32
```

This process is repeated for all the expected messages from the nodes,

```
<Message_name>, <CAN_ID>, <Param1_Name>, <Param1_Unit>, <Param1_type>, <Param1_bits>, <Param2_Name>, <Param2_Unit>,
<Param2_type>,<Param2_bits>

@SUPCAP1, 10, SOC,%, F, 32, CAP_Volt, V,F, 32

THROTTLE, 11, Volt1_2, V,F,32

@CURR1, 14, Curr1, A,F, 32, Curr1_raw, bit, I, 32

@CURR2, 15, Curr2, A, F, 32, Curr2_raw, bit, I, 32

@CURR3, 16, Curr3, A,F, 32, Curr3_raw, bit, I, 32

GPS1, 22, Lat, deg, I, 32, Lon, deg, I, 32

etc...
```

When the receiver sees a message and cross-identifies it from the README file, it will record it into a folder with its Message_name. Messages that are not written down correctly onto the README file will not be recorded onto the SD card.

▶ Details
How radio modes work

There are two radio modes:

- **Live-send telemetry**, where all the data to be sent is sent the instant it is received by the in-vehicle receiver and is never sent again after that. This is useful to have real-time information, but is only possible if the vehicle is in view and at a short distance (e.g. 200m)
- **Batch-send telemetry**, where all the data to be sent is stored in a file on the SD card until the vehicle enters a pre-defined geofenced zone where it starts sending all the stored data for a lap together. The idea is that you would be sending a whole lap's data just in one section of the track. This is useful if the vehicle is not in view at all times, or if it goes further than the max transmission distance (400-500m).

To activate either mode, you need to define it in the begin statement of the Receiver class/object.

```
Receiver receiver("/config/README.txt");
bool liveSendModeOn = false; // This means the batch-send mode will be used
...
void setup() {
  receiver.begin(liveSendModeOn);
...
}
```

Build a receiver

If you need to order and assemble a new receiver, you can use the gerber files provided in the Github repository, and upload them to JLCPCB.

{% tabs %} {% tab title="Schematics" %} **Electrical Diagram**

{% file src=".gitbook/assets/receiver v2ra schematic.pdf" %}

PCB footprint

{% file src=".gitbook/assets/Receiver PCB Footprint.pdf" %} {% endtab %}

{% tab title="3D Viewer" %} {% embed url="https://a360.co/3wpQ9fS" %} {% endtab %}

{% tab title="Parts list (BOM)" %}

Qty	Value	Device	Package	Parts	Description	MANUFACTURER_NAME	MANUFACTURER_PA
1		SMA CONNECTOR EDGE	SMA_EDGELAUNCH	X1	SMA Female Connector	Amazon	N/A
1	0	R_CHIP-0603(1608-METRIC)	RESC1608X60	R22	Resistor Fixed - SMD		
2	0.1 uF	C_CHIP-0603(1608-METRIC)	CAPC1608X85	C1	Capacitor - Ceramic SMD		
7	0.1uF	C_CHIP-0603(1608-METRIC)	CAPC1608X85	C2	Capacitor - Ceramic SMD		
2	0.1uF (NC)	C_CHIP-0603(1608-METRIC)	CAPC1608X85	C12	Capacitor - Ceramic SMD		
2	100k	R_CHIP-0603(1608-METRIC)	RESC1608X60	R2	Resistor Fixed - SMD		
12	10k	R_CHIP-0603(1608-METRIC)	RESC1608X60	R4	Resistor Fixed - SMD		
4	10uF	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C5	Capacitor - Ceramic SMD		
1	1k	R_CHIP-0603(1608-METRIC)	RESC1608X60	R25	Resistor Fixed - Generic		
1	1uF	C_CHIP-0603(1608-METRIC)	CAPC1608X85	C20	Capacitor - Ceramic SMD		
1	300	R_CHIP-0603(1608-METRIC)	RESC1608X60	R12	Resistor Fixed - Generic		
1	400k	R_CHIP-0603(1608-METRIC)	RESC1608X60	R21	Resistor Fixed - Generic		
2	5.1k	R_CHIP-0603(1608-METRIC)	RESC1608X60	R3	Resistor Fixed - SMD		
1	500	R_CHIP-0603(1608-METRIC)	RESC1608X60	R1	Resistor Fixed - SMD		
2	60	R_CHIP-0805(2012-METRIC)	RESC2012X65	R8_C	Resistor Fixed - SMD		
4	NC	C_CHIP-0603(1608-METRIC)	CAPC1608X85	C4	Capacitor - Ceramic SMD		
2	NC	R_CHIP-0603(1608-METRIC)	RESC1608X60	R14	Resistor Fixed - SMD		
1	500ASSP1M6QE	500ASSP1M6QE	500ASSP1M6QE	S1	Power Slide Switch	E-SWITCH	500ASSP1M6QE
1	CDSOT23-T24CAN	CDSOT23-T24CAN	SOT95P230X117-3N	D1	CAN TVS Diode	Bourns	CDSOT23-T24CAN
1	COM-13909	COM-13909	MOD_COM-13909	RFM69	Rfm69hcw General ISM < 1GHz Transceiver Module 915/433MHz Surface Mount	HOPERF	RFM69HCW
1	ESP32-S3-WROOM-1-N16R2	ESP32-S3-WROOM-1-N16R2	XCVR_ESP32-S3-WROOM-1-N16R2	U1	Espressif ESP32-S3 Microcontroller	Espressif	ESP32-S3-WROOM-1

Qty	Value	Device	Package	Parts	Description	MANUFACTURER_NAME	MANUFACTURER_PA
1	JS102011JCQN	JS102011JCQN	JS102011JCQN	S3	Slide Switch .3A 6VDC SPDT VERT MNT SMT J BEND (CAN Termination)	C & K COMPONENTS	JS102011JCQN
1	JST-XH-04-PIN-LONG-PAD	JST-XH-04-PIN-LONG-PAD	JST-XH-04-PACKAGE-LONG-PAD	X4	JST XH Connector 4 Pin	Amazon	N/A
1	LM1117MPX-3.3_NOPB	LM1117MPX-3.3_NOPB	SOT230P700X180-4N	IC3	Space saving 800-mA low-dropout linear regulator with internal current limit	Texas Instruments	LM1117MPX-3.3/NOP
1	MEM2075-00-140-01-A	MEM2075-00-140-01-A	MEM20750014001A	J1	MEM2075-00-140-01-A - MICRO SD CARD CONN	GCT	MEM2075-00-140-01
1	NEOPIXEL	1655	1655	LED1	NeoPixel 5050 RGB LED with Integrated Driver Chip	Adafruit	NeoPixel 5050 - 1655
1	PCM12SMTR	PCM12SMTR	PCM12SMTR	S2	Slide Switches 0.3A SPDT ON-ON (USB Switch)	C & K COMPONENTS	PCM12SMTR
1	PMEG1020EJ	PMEG1020EJ	SOD-323	D4	Schottky Rectifier, 10 V, 2 A, Single, SOD-323, 2 Pins, 460 mV	NEXPERIA	
3	PTS841ESDGKPSMTRLFS	PTS841ESDGKPSMTRLFS	PTS841ESDGKPSMTRLFS	BOOT	Side-actuated tactile switch	C&K	PTS841 ESD GKP SMT
1	SRV05-4.TCT	SRV05-4.TCT	SOT95P280X145-6N	D2	USB TVS Diode ESD Suppressor	Semtech	SRV05-4.TCT
1	TCAN1462VDRQ1	TCAN1462VDRQ1	SOIC127P600X175-8N	CAN_T	CAN FD transceiver	Texas Instruments	TCAN1462VDRQ1
1	TPS2110APWR	TPS2110APWR	SOP65P640X120-8N	IC2	Autoswitching Power Mux	Texas Instruments	TPS2110APWR
1	TYPE-C-31-M-12	TYPE-C-31-M-12	HRO_TYPE-C-31-M-12	J2	USB Type C Connector (JLCPCB Assembled)	HRO Electronics Co.	TYPE-C-31-M-12
1	XW4K-04A1-V1CONN_XW4K-04A1_OMR	XW4K-04A1-V1CONN_XW4K-04A1_OMR	CONN_XW4K-04A1_OMR	J3	Pluggable Terminal Block 4 pin Vertical Port	Omron	XW4K-04A1-V1
1	XW4H-04A1	XW4H-04A1	XW4H-04A1	N/A	Pluggable Terminal Blocks PCB Terminal Block Wire Side, 4 con	Omron	XW4H-04A1
1	PMEG3010EJ	PMEG3010EJ	SOD-323	D3	Schottky Rectifier, 10 V, 2 A, Single, SOD-323, 2 Pins	NEXPERIA	

{% endtab %} {% endtabs %}

Some tips on soldering the receiver,

- Start with the bottom surface with the slide switch
- Progress to the other small components (capacitors and resistors)
- Place large components before placing the smaller ones around it (e.g. CAN transceiver before its capacitors)
- On the top side, place the LED first, then the pushbutton switches, then the SD socket, then the CAN termination switch, then the Power switch, then the SMA connector, then the MCU chip, then the Omron connector and finally the JST-XH connector.

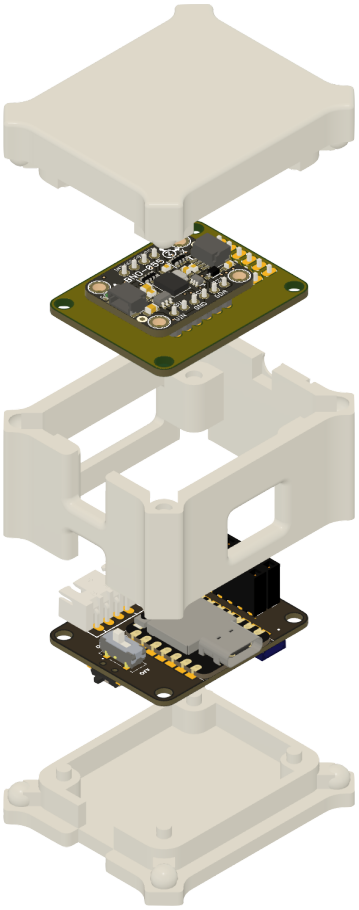
Sensors

IMU

The IMU shield uses the Bosch BNO055 breakout board from Adafruit, soldered onto a shield board.

It is the simplest of the sensor shields, showcasing its use case with a typical sensor breakout board.

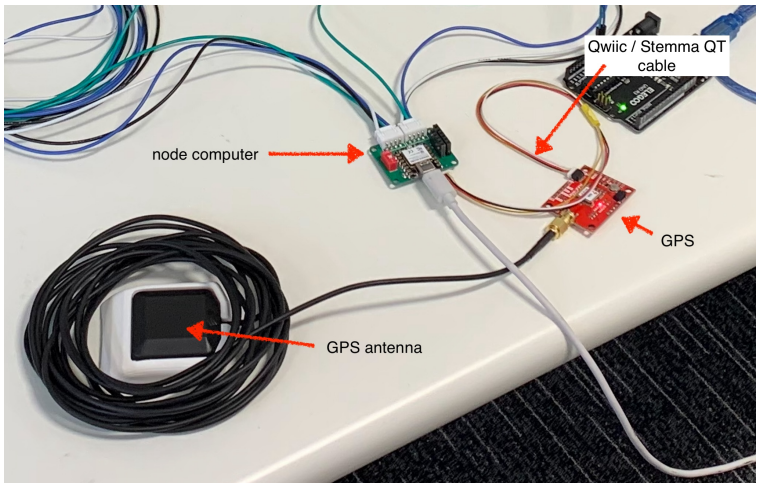
The shield only connects the power lines (5V and GND) as well as the I2C communication lines (SDA and SCL) of the sensor to the node computer.



Exploded view of the IMU node, with the sensor shield on top.

GPS

The GPS is not a shield but a breakout board directly connected via the Qwiic connector on both the sensor and node computer. This cable connects the power and I2C lines.



Test bench showcasing the GPS connected to the node computer through the Qwiic cable.

Power meter (Voltage and Current)

{% hint style="danger" %} It is recommended to re-design this shield as it presents a few flaws, most notably not enough protection and confusing layout making it hard to use.

The components were prone to failure (fried) when used with the hydrogen setup during testing and competition.

More on this in the future work section. {% endhint %}

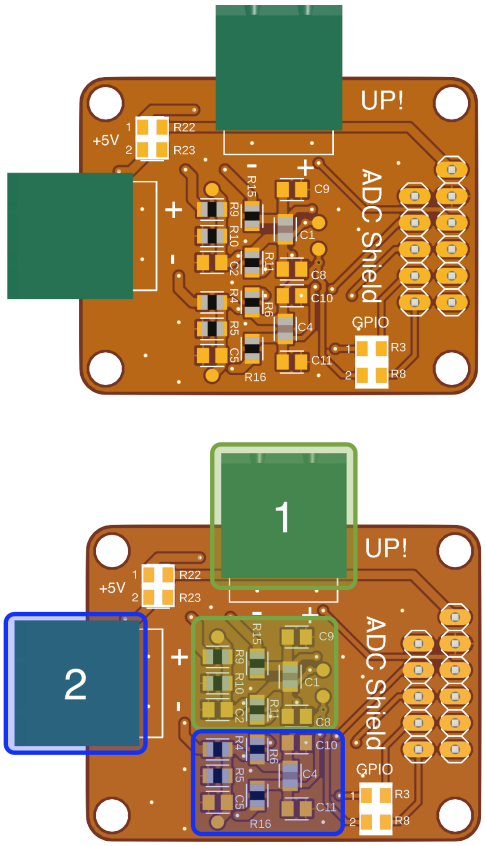
The power meter, sometimes referred to as the voltmeter or current meter, can measure either two voltages at the same time or be slightly modified to connect to an analog current transducer to measure one current. It is not possible to measure a voltage and current at the same time.

It uses a TI ADS1115 chip, which is a 16-bit analog to digital converter (ADC). The reason for using a separate ADC chip to a op-amp to MCU ADC configuration was because the ESP32 ADC is not very good (noisy, low impedance and small range).

The current transducer used is a LEM HO 50-S/SP30-0100.

{% tabs %} {% tab title="Overview" %} The shield has two voltage inputs (+ and -, green terminal plug connectors), which can be modified into one power output and one voltage measurement when used with a current transducer.

All the resistors and capacitors on the topside are interchangeable to change the purpose of the sensor, and change the output voltage of the simple voltage dividers.



{% endtab %}

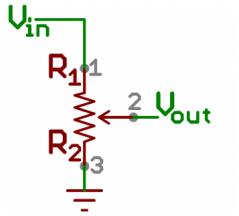
{% tab title="Voltage Measurement" %} {% hint style="warning" %} This is a guess from my previous work as I do not have access to the shields.

Look at the shield available to figure out if that is indeed how it is configured (i.e. resistor placement and values). {% endhint %}

To set it up for voltage measurements, the ADS1115 chip is used in differential mode[^3].

- Footprints R9/R4 and C1/C4 are populated by the voltage dividing resistors (yes a resistor on a capacitor footprint), R9 being the first resistors and C1 connecting the measured positive pole to the measured negative pole without grounding it to the node. Since the measured voltage and the node's power are not connected, I think it is better not to connect the grounds. But I'm not 100% sure.
- Footprints R15/R11/R6/R16 are populated with a 0 Ohm resistor, or can just be connected with solder.
    - The resistor value that were used:
        - R9/R4 -> a low value e.g. 30kOhm
        - C1/C4 -> a very high value[^4] e.g. 1 MOhm

      The goal was to lower the maximum voltage experienced by the supercapacitor (around 32.4V) down to the maximum voltage of the ADC chip (configured as 2V in software) to give us the highest possible measurement resolution.



{% endtab %}

{% tab title="Current Measurement" %} To set it up for current measurements, the ADS1115 chip is used in differential mode[^5].

The LEM transducer has 4 pins we are connecting to:

- **Power pins**, namely voltage in Uc and GND, routed to connector 1.
    - For this, populating R22 and R2[^6] connect the + and - terminals of connector 1 to +5V and GND respectively.
- **Voltage output pins**, Vout and Vref, routed to connector 2.
    - The ADC compares the Vout[^7] which is a voltage between 0V and Uc (+5V) to Vref which is the reference voltage at Uc/2.
    - For this, populating R4 and C4 with voltage dividing resistors in order to bring the 5V maximum output down to the maximum of ADC chip. This is setup just like a normal voltage measurement in differential mode.
        - Footprints R6/R16 are populated with a 0 Ohm resistor, or can just be connected with solder.

{% hint style="warning" %} The current transducer senses the electro-magnetic field around a wire.

It was observed that the measurements could sometimes be offset by a fixed value when large currents passed through the wire within a short period of time.

Therefore, we implemented a re-calibration feature in software.

1. The node calibrates the sensor to 0.0A on startup so make sure no current is flowing on whatever you are measuring.
2. In the serial monitor command line, write the letter 'c' without the ' ' to start the calibration process to a maximum current value currently running in the wire and that is know (use an adequate power supply). You will need to enter the current value in Amperes that you are calibrating to.
3. The node automatically recalibrates the sensor to 0.0A when a shift from the initial calibration is detected within a small range around 0 (e.g. +/- 300 bits, which can represent a few mA). This is to counteract the shift/offset effect of the sensor, but it does not solve the problem on drastic shifts. {% endhint %} {% endtab %} {% endtabs %}

{% tabs %} {% tab title="Schematics" %} **Electrical diagram**

{% file src=".gitbook/assets/Power Shield Schematic.pdf" %}

PCB footprint

{% file src=".gitbook/assets/Power Shield PCB Footprint.pdf" %} {% endtab %}

{% tab title="3D Viewer" %} {% embed url="https://a360.co/3AmoNJs" %} {% endtab %}

{% tab title="Parts list (BOM)" %}

Qty	Value	Device	Package	Parts	Description
6		C_CHIP-0805(2012-METRIC)	CAPC2012X110	C1	
1		PINHD-1X5	1X05	JP3	PIN HEADER
1		PINHD-1X6	1X06	JP1	PIN HEADER
4		R_CHIP-0805(2012-METRIC)	RESC2012X65	R6	

Qty	Value	Device	Package	Parts	Description
2	(NC)	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C2	
1	(NC) 10k	R_CHIP-0805(2012-METRIC)	RESC2012X65	R21	Resistor Fixed - Generic
8	0 (NC)	R_CHIP-0805(2012-METRIC)	RESC2012X65	R2	
3	0.1uF	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C3	
4	10k	R_CHIP-0805(2012-METRIC)	RESC2012X65	R17	
2	2P 5.08mm pitch	2828XX-2282837-2	TERMBLK_508-2N	J1	
2	3.3k	R_CHIP-0805(2012-METRIC)	RESC2012X65	R5	
2	43k	R_CHIP-0805(2012-METRIC)	RESC2012X65	R4	
1	ADS1115IDGSR	ADS1115IDGSR	SOP50P490X110-10N	IC2	16-Bit 860SPS 4-Ch Delta-Sigma ADC With PGA
2	Fuse	R_CHIP-0603(1608-METRIC)	RESC1608X60	R1	
2	MCP6401T-E/OT	LMH6629_SOT23	SOT95P280X145-5N	IC1	
1	SOT23 TVS DIODE	CDSOT23-T24CAN	SOT95P230X117-3N	D1	Bourns CDSOT23-T24CAN
4	TEST-POINT3X5	TEST-POINT3X5	PAD.03X.05	TP-PWM-1	

{% endtab %} {% endtabs %}

Driver display

The driver display (4D Systems gen4-uLCD-70D) acts as a sensor node, as the screen is connected via a 30-pin FPC cable to a shield which communicates with the node computer and thus the CAN bus.

This shield is a copy of the screen manufacturer's own module, and was designed after looking at their schematics.

{% tabs %} {% tab title="Schematics" %} Electrical Diagram

{% file src=".gitbook/assets/gen4-IB shield schematic.pdf" %}

PCB footprint

{% file src=".gitbook/assets/gen4-IB shield pcb layout.pdf" %} {% endtab %}

{% tab title="3D Viewer" %} {% embed url="https://a360.co/3UKuSaM" %} {% endtab %}

{% tab title="Parts list (BOM)" %}

Qty	Value	Device	Package	Parts	Description	Manufacturer	Link
1		PINHD-1X5	1X05	JP3	PIN HEADER	n/a	n/a
1		PINHD-1X6	1X06	JP1	PIN HEADER	n/a	n/a
3	0	R_CHIP-0805(2012-METRIC)	RESC2012X65	R4	Resistor Fixed - Generic	n/a	n/a
1	0.1 uF	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C1	Capacitor - Generic	n/a	n/a
1	10uF	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C2	Capacitor - Generic	n/a	n/a

Qty	Value	Device	Package	Parts	Description	Manufacturer	Link
1	2P 2.54mm Screw Terminal	2828XX-2282834-2	TERMBLK_254-2N	J2	2 Position Wire to Board Terminal Block Horizontal with Board	n/a	https://www.amazon.co.uk/Screw-Terminal-Connector-2-54mm-UniveP7Qjboh0Ml7oAVJ5ja5ESzpyAxvcU1fATTIdL92SoxXLcmprO9iZYqXb1QtgjbEtG7ArG6NllpZu6R71XRHyFFYaBFDTsqyIZnzfKrzSZcSM.bqkXbXQG2
1	4.7k	R_CHIP-0805(2012-METRIC)	RESC2012X65	R7	Resistor Fixed - Generic	n/a	n/a
3	68	R_CHIP-0805(2012-METRIC)	RESC2012X65	R1	Resistor Fixed - Generic	n/a	n/a
1	SFV30R-4STBE1HLF	SFV30R-4STBE1HLF	SFV30R4STBE1HLF	J1	0.50mm Flex Connector	Amphenol	https://www.mouser.co.uk/ProductDetail/Amphenol-FCI/SFV30R-4STB

{% endtab %} {% endtabs %}

It also features two holes that can be used for a screw terminal to attach a toggle switch. This toggle switch can be programmed with the MCU to perform logic operations such as dimming the lights of the screen.

micro-SD

The micro-SD shield allows the user to transform a node into a CAN datalogger device.

Connect it to the node computer, attach it to the CAN bus and program it to record any CAN message sent.

{% hint style="warning" %} Make sure to configure the datalogger node in LISTEN ONLY mode of CAN if the receiver is also attached to the bus, otherwise it will not be able to register messages. {% endhint %}

{% tabs %} {% tab title="Schematics" %} Electrical diagram

{% file src=".gitbook/assets/microSD shield v1ra schematic.pdf" %}

PCB layout

{% file src=".gitbook/assets/microSD shield v1ra pcb.pdf" %} {% endtab %}

{% tab title="3D Viewer" %} {% embed url="https://a360.co/48m8Cr2" %} {% endtab %}

{% tab title="Parts list (BOM)" %}

Qty	Value	Device	Package	Parts	Description	Manufacturer	Link	Unit (£)
1	PINHD-1X5	1X05	JP3	PIN	HEADER	n/a	n/a	n/a
1	PINHD-1X6	1X06	JP1	PIN	HEADER	n/a	n/a	n/a
1	0	R_CHIP-0805(2012-METRIC)	RESC2012X65	R5	Resistor	n/a	n/a	n/a
1	0.1uF	C_CHIP-0805(2012-METRIC)	CAPC2012X110	C1	Capacitor	n/a	n/a	n/a
5	10k	R_CHIP-0805(2012-METRIC)	RESC2012X65	R4,	Resistor	n/a	n/a	n/a
1	MEM2075-00-140-01-A	MEM2075-00-140-01-A	MEM20750014001A	J2	uSD card connector	GCT	https://www.mouser.co.uk/ProductDetail/GCT/MEM2075-00-140-01-A?qs=KUoIvG/9IlZvfWpeERlq3Q%3D%3D	1.58

{% endtab %} {% endtabs %}

RFM69(HCW)

Similar to the datalogger node, there can also be a radio node.

{% tabs %} {% tab title="Schematics" %} **Electrical diagram**

{% file src=".gitbook/assets/RFM69HCW Shield Schematic.pdf" %}

PCB Footprint

{% file src=".gitbook/assets/RFM69HCW PCB Footprint.pdf" %} {% endtab %}

{% tab title="3D Viewer" %} {% embed url="https://a360.co/3OLsfBB" %} {% endtab %}

{% tab title="Parts list (BOM)" %}

Qty	Value	Device	Package	Parts	Description	Manufacturer	Link	U (
1		PINHD-1X5	1X05	JP3	PIN HEADER	n/a	n/a	n
1		PINHD-1X6	1X06	JP1	PIN HEADER	n/a	n/a	n
2	100k	R_CHIP-0603(1608-METRIC)	RESC1608X60	R1	Resistor - Generic	n/a	n/a	n
1	10uF	C_CHIP-0603(1608-METRIC)	CAPC1608X85	C1	Capacitor - Generic	n/a	n/a	n
2	NC	R_CHIP-0603(1608-METRIC)	RESC1608X60	R2	Resistor - Generic	n/a	n/a	n
1	14	SMACONNECTOR_EDGE	SMA_EDGELAUNCH	X1	SMA Connector	n/a	n/a	n
1	COM-13909	COM-13909	MOD_COM-13909	U2	Rfm69hcw Wireless Transceiver	HOPERF	https://www.mouser.co.uk/ProductDetail/Adafruit/5693?qs=mELouGlnn3fnO8nGmYJgWQ%3D%3D	4

{% endtab %} {% endtabs %}

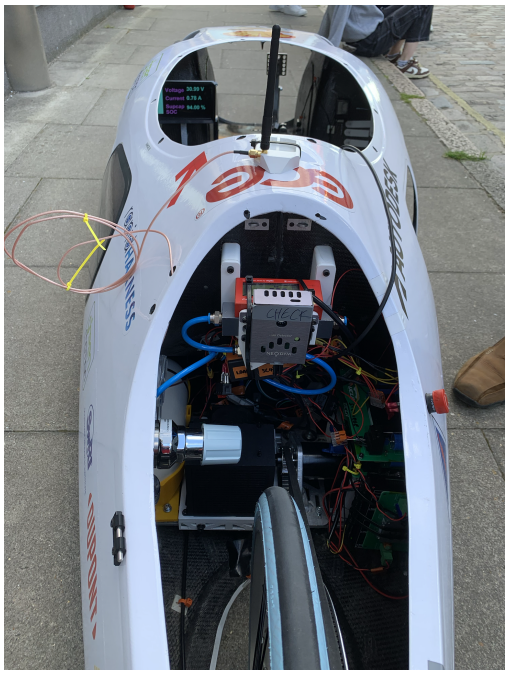
2024 configuration

The 2024 UCL vehicle had the following nodes,

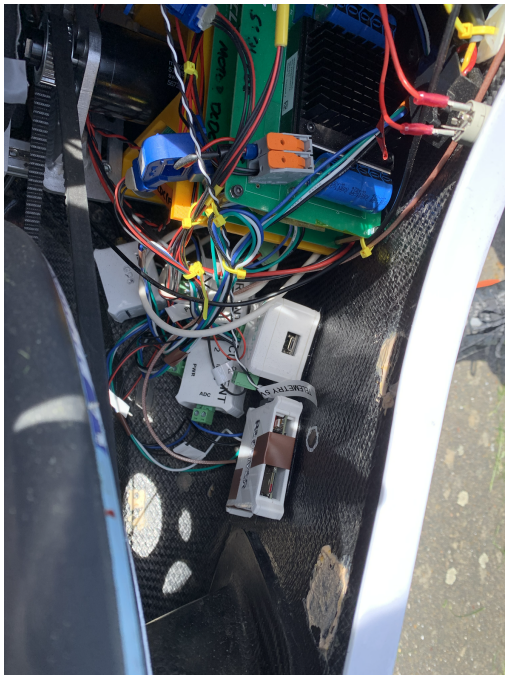
- (x1) voltage sensor
  - on the super-capacitor poles to measure the State-Of-Charge of the super-capacitor. The SOC was calculated by multiplying the measured voltage (0 to 32.4V) by 3.08642.
  - on the DC-DC output.
- (x3) current sensors
  - on the super-capacitor output.
  - on the motor controller output to the motor.
  - on the DC-DC output.

{% hint style="info" %} Note that because of how the voltmeter shield was designed, it can be used to measure two voltages at the same time, or re-arranged to measure one current sensor. {% endhint %}

- An Inertial Measurement Unit (IMU) - placed close to the centre of mass
- A GPS - with antenna



Outside view of the vehicle loom



The placement during competition became very difficult due to the space constraints

- An LCD screen - for the driver

Driver display showing a test setup

Future Work

New power shield (Voltage + Current)

The voltage shields failed multiple times at the competition.

Sometimes it was only the sensor chip that blew, other times the whole node died (sensor and node computer). We were able to tell the sensor died because when connecting looking at the serial monitor it was showing that it couldn't find any I2C devices on the bus.

The exact reason for the failures is unknown, but there were problems with un-even groundings and the whole chassis of the car being grounded (i.e. you could measure a voltage when touching the carbon fibre). This would mean that, although the theoretical maximum voltage differentials of the sensors were not reached, they may have been destroyed from some electro-static discharge of some kind.

Thus, for future years it is necessary to redesign this sensor shield to incorporate as much protection as possible (ask William Backhouse in MechSpace about isolation circuitry using an isolation amplifier for a voltmeter/current-meter). The redesign should also - if possible - try to make it easier for the user to measure a voltage either by better suited connector plug and sockets (currently pluggable Phoenix terminal blocks) and make it easier to use in general (e.g. making changing the voltage divider layout more obvious and easy to understand).

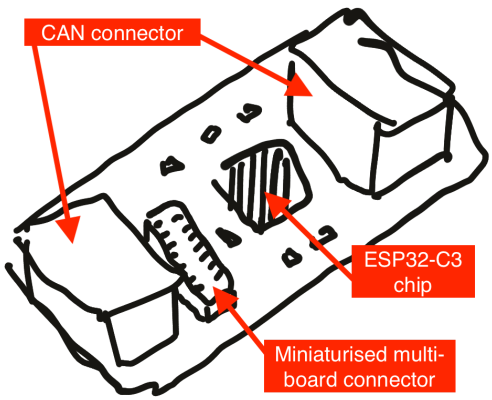
{% hint style="info" %} Note: if you're making methods to add 0 Ohm resistors as connectors, consider using special footprints that allow you to easily connect traces with solder. In our implementation we used normal footprints with solder which was often difficult to bridge. Adafruit/Sparkfun probably have some footprints. {% endhint %}

Smaller nodes

The obvious improvement to the system is to miniaturise it, as it is quite bulky and inconvenient as-is.

This would mean copying the PCB design of the XIAO microcontrollers, but removing all the breakout board aspects to only be left with the critical components. You could then place them all to use the less amount of space, and overall get rid of all the pointless switches (power source switching) and accessibility features (termination resistor switch) of the current node computer design.

It is important to keep and add all safety features.



Quick drawing of a miniaturised design with a bare ESP32-C3 chip.

New Driver Display

The driver display is a major part of the current consumption of the system, taking it directly from whatever power source that the vehicle has (e.g. auxiliary 12V battery).

From memory, it was required in the 2024 competition rules that everything be powered from the same source, thus the telemetry system would take from the aux battery charged by the fuel cell, impacting the energy performance.

I think the Politecnico di Torino team used a smartphone as a screen, telemetry communication with the pits and saving device instead of having them all separate.

It is true in hindsight that that is a more viable option if you can get serial communication to work between a simple node computer and an Android phone.

1. Is inherently powered by the phone battery, thus the whole system impacting the performance much much less. This means using at least 3.5Wh less than currently.
    - The competition may not be able to enforce that the phone be counted in the power usage of the vehicle, because it could be the same phone used by the driver to communicate with the race engineer.
2. Better display overall: better brightness/resolution, more processing power, potentially easier to program.
3. So much easier to install in the car, would just need a phone holder and a USB-C cable.
4. Actually cheaper than the 4D Systems display if you get a low budget android smartphone.

So the plan would be that the phone is connected to a node computer via the USB-C port, and communicates over serial.

The node computer would send the necessary data it receives similarly to how the radio station is communicating with the race engineer's software, but here software on the phone receives, processes and displays it on a purpose-built app to the driver.

The phone would simultaneously transmit the data of 4G/5G to a server which the race engineer connects to. This could be a webpage.

The phone would also save all the data on-board in a text or csv file, similar to how it is doing it on the SD card. This may even be way faster than currently.

And finally the driver would also be calling the race engineer with that same phone and some wireless earbuds, over 4G.

This is quite a big software project of its own, which may require someone to work on it for a few months.

[^1]: This is the name for an arduino project, essentially the code you upload to a microcontroller.

[^2]: This is the name for an arduino project, essentially the code you upload to a microcontroller.

[^3]: Compares voltage on one line to the other, necessary as measured ground may be at different level to node ground.

[^4]: to have a very high impedance to minimise current draw -> minimal disturbance to the measured system

[^5]: Compares voltage on one line to the other, necessary as measured ground may be at different level to node ground.

[^6]: Board's underside

[^7]: The sensor detects current in both directions by setting the 0.0A value at Uc/2. Currents above this reference indicate positive flow, while currents below it represent negative flow.