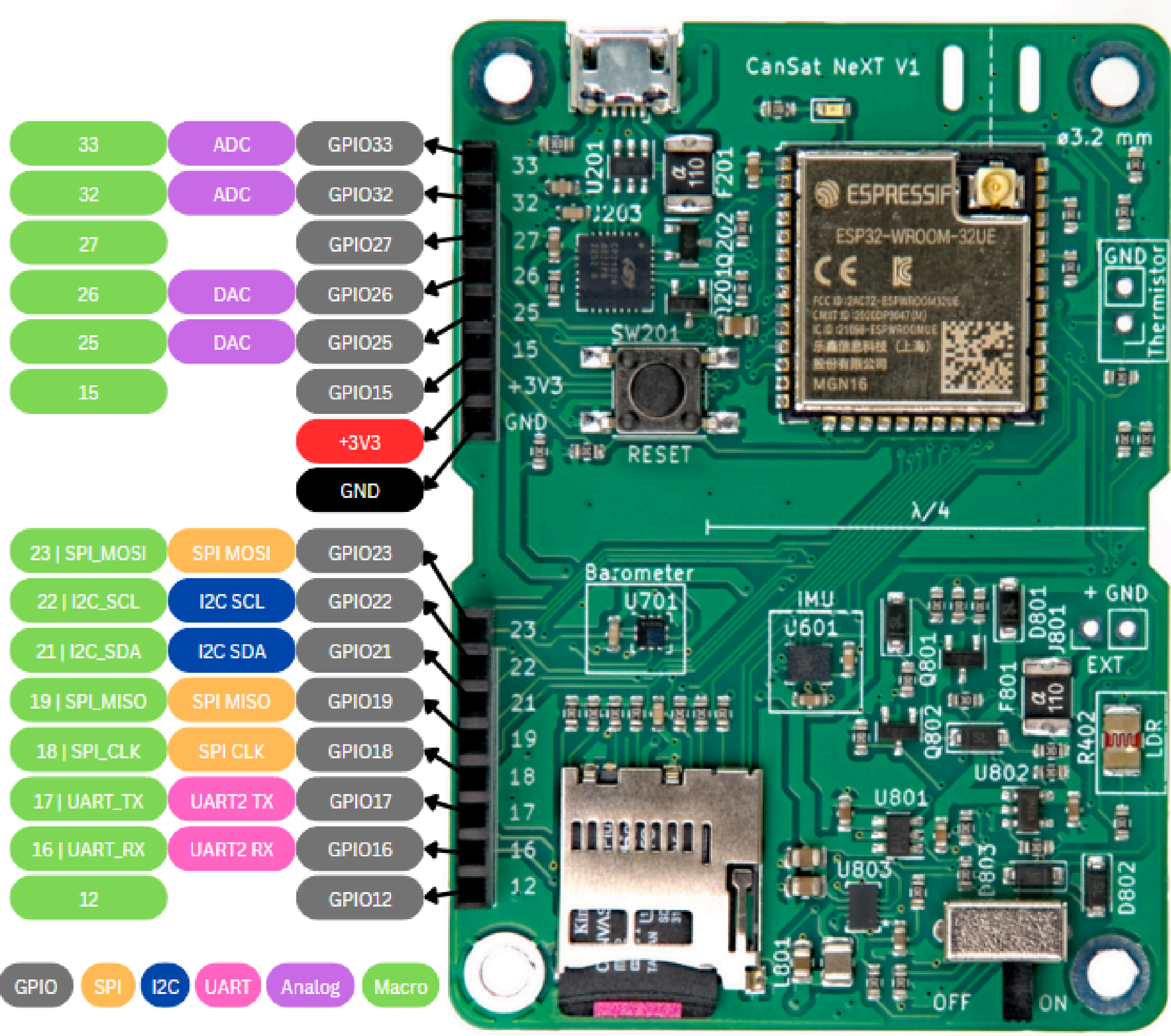
Welcome to Documentation

Welcome to CanSat NeXT documentation page! This site includes detailed technical documentation of the CanSat NeXT hardware and software, as well as easy to approach tutorials on setting up your CanSat NeXT and using the various hardware features on the board.

If you are new here, head to Getting Started Page for information on how to start using CanSat NeXT. You may also be interested in information of making the basic antenna from the materials included with the kit. For that, head to article Communication and Antennas.

While you are here, don't forget to also check our blog, where we show projects using CanSat NeXT for purposes beyond CanSat. These are not CanSat projects, but still showcase the possibilities of what can be done with CanSat NeXT.

Finally, if you are anything like me, you probably came to this page looking for the pinout. More information about that in Pinout, but here is a quick reference:



License

This library and the CanSat NeXT board are developed by Samuli Nyman, in collaboration with ESERO Finland and Arctic Astronautics Oy. The development is also supported by the Finnish Physical Society. This software library is licensed under the MIT license.

Contribution

If you wish to contribute to the library or if you have feedback, please contact me through samuli@kitsat.fi or start a GitHub issue. You are also welcome to create a pull request.

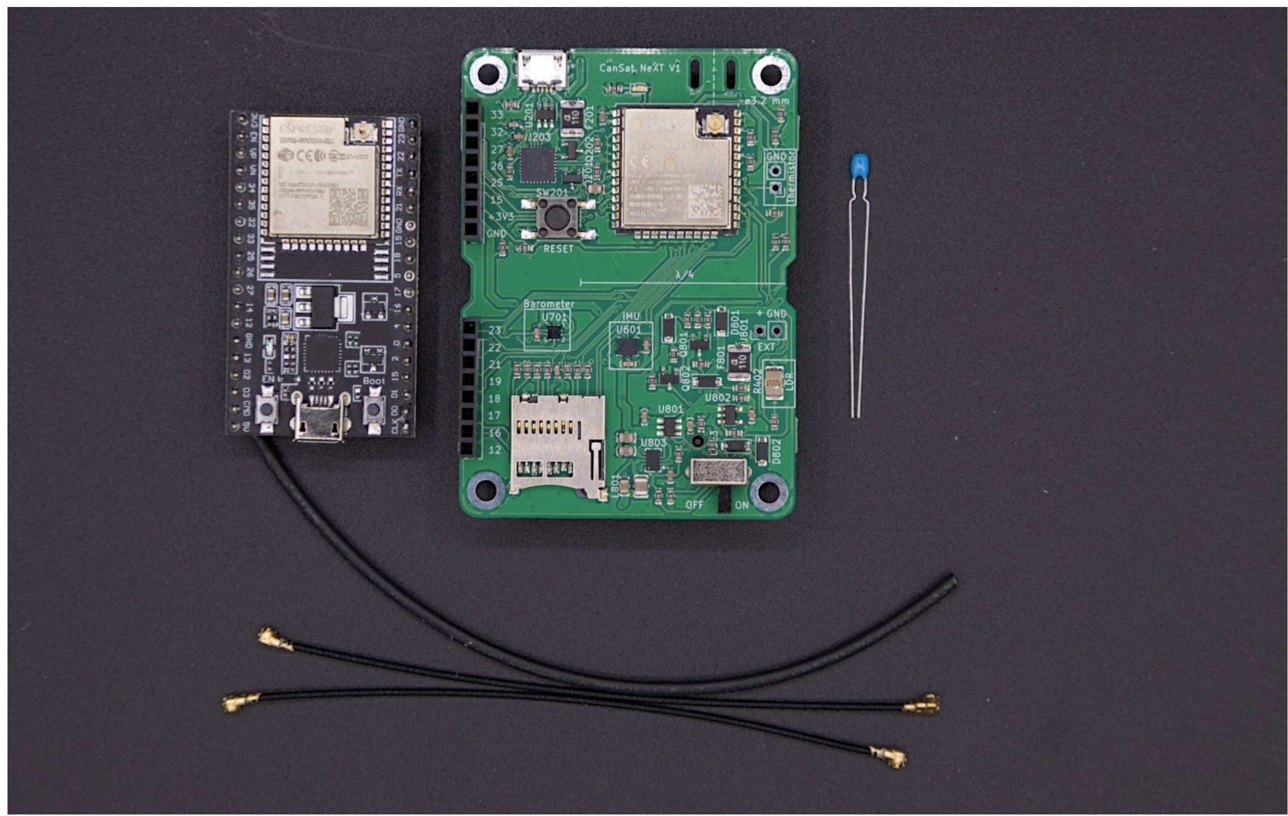
Getting Started with CanSat NeXT

Welcome to CanSat NeXT!

CanSat NeXT is a new variant of the CanSat kit, which integrates the necessary features needed for a successful CanSat launch directly on one board, enabling you to get started immediately with software development and your own missions. However, CanSat NeXT doesn't forget your secondary mission, as you can connect any sensors or external devices to the extension headers. You can think of CanSat NeXT like an Arduino, just with sensors and other features already included straight out of the box.

Your Kit

If you don't yet have a CanSat NeXT kit, you can get one from our webshop: https://holvi.com/shop/kitsat/section/cansat/. Alternatively, schools participating in CanSat competitions and programs are usually eligible to get kits through ESERO network.



The kit includes one CanSat board, which is what you will mostly be working with. Additionally, there is another board, which will be used as the groundstation radio - you will use that to relay messages between a computer and the CanSat.

While CanSat NeXT already has a thermometer on board, the kit also includes a thermistor, which can be soldered to the board to measure temperature outside the board itself.

Finally, the kit includes two radio cables, which can be used to build basic antennas to enable communication up to a kilometer away. Only one cable is needed, but it is nice to have a backup. The heat-shrink tubing is included to add weather protection for the antennas. For instructions on how to build the antenna, refer to article Communication and Antennas.

Lessons

This page includes a growing number of simple lessons to get you smoothly started with your CanSat NeXT kit. The first lesson is about setting up your computer to start CanSat programming, and the following lessons present various hardware features of CanSat NeXT. Additionally, we have a blog for showcasing various projects done with CanSat NeXT, which might be interesting when planning your own CanSat mission.

Click here for the first lesson!.

Lesson 1: Hello World!

This first lesson shows gets you started with CanSat NeXT by showing how to write and run your first program on the board.

After this lesson, you will have the necessary tools to start developing software for your CanSat.

Installing the tooling

CanSat NeXT is recommended to be used with Arduino IDE, so let's begin by installing that and the necesssary libraries and boards.

Install Arduino IDE

If you haven’t already, download and install the Arduino IDE from the official website https://www.arduino.cc/en/software.

Add ESP32 support

CanSat NeXT is based on the ESP32 microcontroller, which is not included in the Arduino IDE default installation. If you haven’t used ESP32 microcontrollers with Arduino before, the support for the board needs to be installed first. It can be done in Arduino IDE from *Tools->board->Board Manager* (or just press (Ctrl+Shift+B) anywhere). In the board manager, search for ESP32, and install the esp32 by Espressif.

Install Cansat NeXT library

The CanSat NeXT library can be downloaded from the Arduino IDE's Library Manager from *Sketch > Include Libraries > Manage Libraries*.

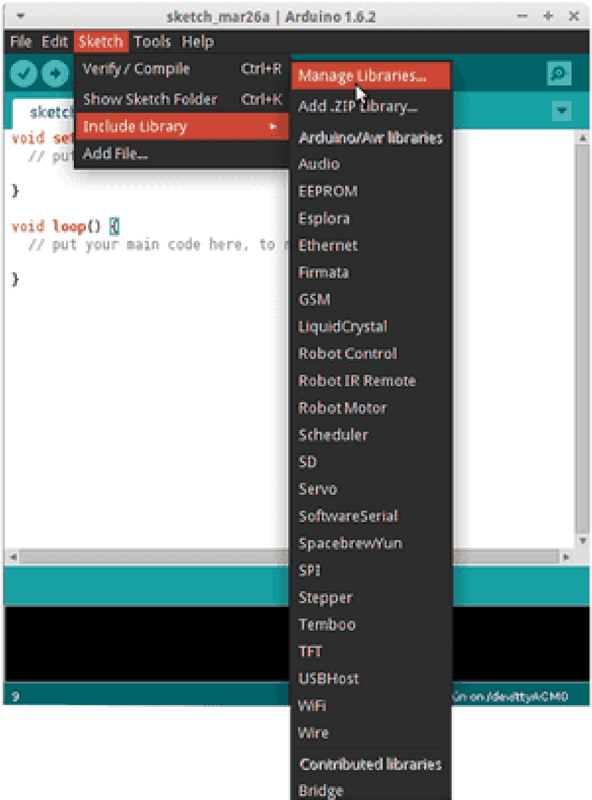


Image source: Arduino Docs, https://docs.arduino.cc/software/ide-v1/tutorials/installing-libraries

In the Library Manager search bar, type "CanSatNeXT" and choose "Install". If the IDE asks if you want to also install the dependencies, click yes.s

Connecting to PC

After installing the CanSat NeXT software library, you can plug in the CanSat NeXT to your computer. In case it is not detected, you may need to install the necessary drivers first. The driver installation is done automatically in most cases, however, on some PCs it needs to be done manually. Drivers can be found on the Silicon Labs website: https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers For additional help

with setting up the ESP32, refer to the following tutorial: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/establish-serial-connection.html

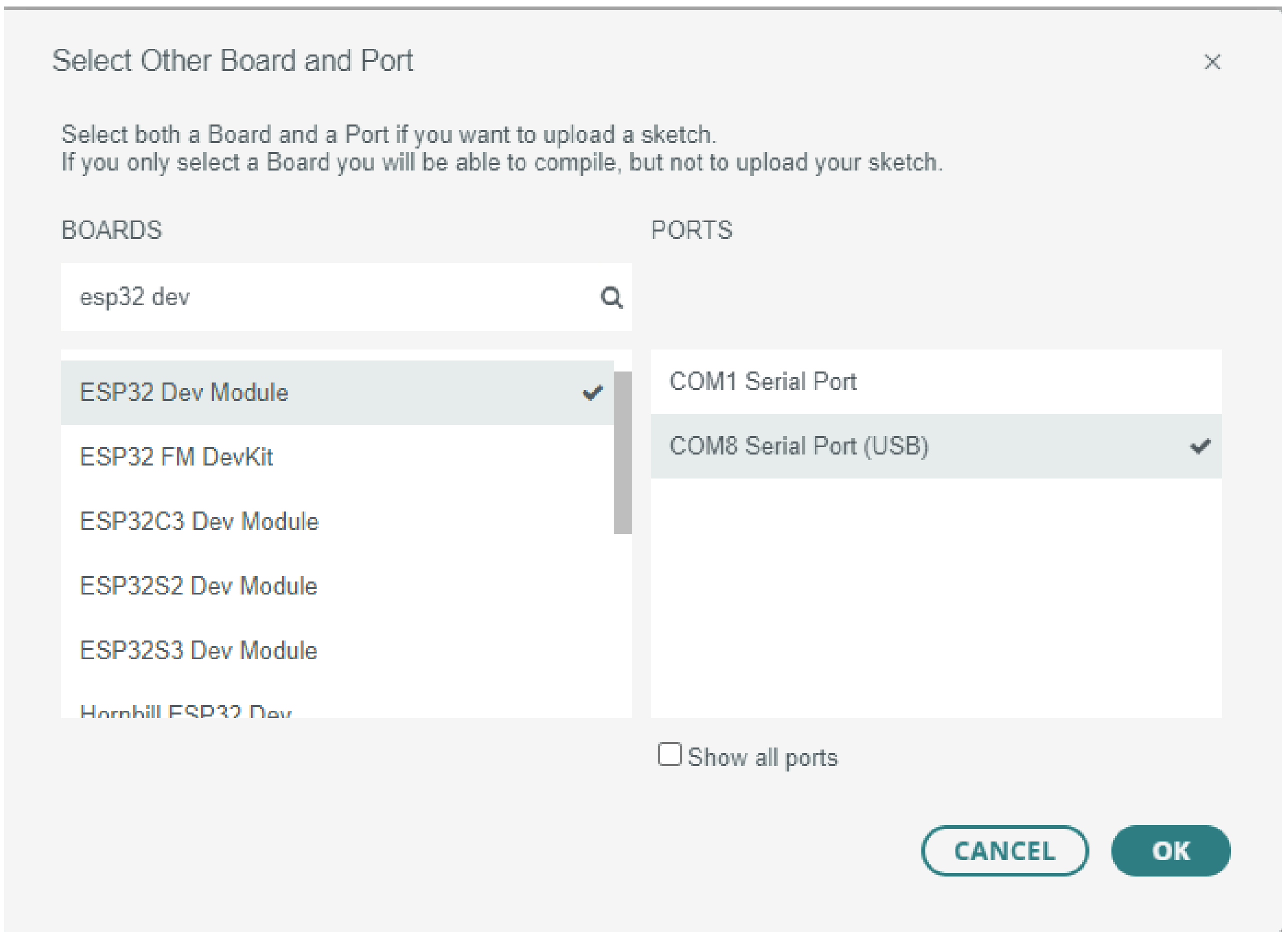
Running your first program

Now, let's use the freshly installed libraries to start running some code on the CanSat NeXT. As is tradition, let's begin by blinking the LED and writing "Hello World!" to the computer.

Selecting the correct port

After plugging the CanSat NeXT into your computer (and turning the power on), you need select the correct port. If you don't know which one is the correct one, simply unplug the device and see which port disappears.

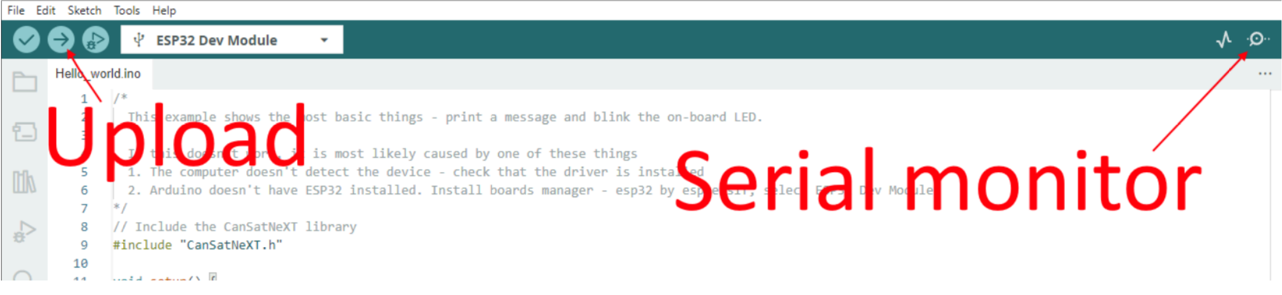
Arduino IDE now prompts your for the device type. Select ESP32 Dev Module.



Choosing an example

The CanSat NeXT library has several example codes showing how to use the various features on the board. You can find these example sketches from File -> Examples -> CanSat NeXT. Pick "Hello_world".

After opening the new sketch, you can upload it to the board by pressing the upload-button.



After a while, the LED on the board should start blinking. Additionally, the device is sending a message to the PC. You can see this by opening the serial monitor, and choosing the baud rate 115200.

Try also to press the button on the board. It should reset the processor, or in other words, restart the code from the beginning.

Hello World explained

Let's see what actually happens in this code by going through it line by line. First, the code begins by **including** the CanSat library. This line should be at the beginning of almost all of the programs written for CanSat NeXT, as it tells the compiler that we want to use the features from the CanSat NeXT library.

Include CanSat NeXT

```
#include "CanSatNeXT.h"
```

After this, the code jumps to the setup function. There we have two calls - first, serial is the interface that we use to send messages to the PC via USB. The number inside the function call, 115200, refers to the baud-rate, i.e. how many ones and zeros are sent each second. The next call, `CanSatInit()`, is from the CanSat NeXT library and it initiates all of the on-board sensors and other features. Similar to the `#include` command, this is usually found in skethes for CanSat NeXT. Anything you'd like to be run just once on startup should included in the setup-function.

Setup

```
void setup() {
  // Start the serial line to print data to the terminal
  Serial.begin(115200);
  // Start all CanSatNeXT on-board systems.
  CanSatInit();
}
```

After the setup, the code starts repeating the loop function endlessly. First, the program writes the output pin LED to be high, i.e. have a voltage of 3.3 volts. This turns on the on-board LED. After 100 milliseconds, the voltage on that output pin is turned back to zero. Now the program waits for 400 ms, and then sends a message to the PC. After the message is sent, the loop function starts again from the beginning.

Loop

```
void loop() {
  // Let's blink the LED
  digitalWrite(LED, HIGH);
  delay(100);
  digitalWrite(LED, LOW);
  delay(400);
  Serial.println("This is a message!");
}
```

You can also try to change the delay values or the message to see what happens. Congratulations for getting this far! Setting up the tooling can be tricky, but it should get more fun from this point onwards. In the next lesson, we will start reading data from the on-board sensors.

Click here for the second lesson!

Lesson 2: Feeling the Pressure

In this second lesson, we will start using the sensors on the CanSat NeXT board. This time, we will focus on measuring the surrounding atmospheric pressure. We will use the on-board barometer LPS22HB to read the pressure, as well as to read the temperature of the barometer itself.

Let's start from the barometer code in the library examples. In Arduino IDE, select File->Examples->CanSat NeXT->Baro.

The beginning of the program looks quite familiar from the last lesson. Again, we start by including the CanSat NeXT library, and setting up the serial connection as well as initializing CanSat NeXT systems.

Setup

```
#include "CanSatNeXT.h"

void setup() {

  // Initialize serial
  Serial.begin(115200);

  // Initialize the CanSatNeXT on-board systems
  CanSatInit();
}
```

The function call `CanSatInit()` initializes all the sensors for us, including the barometer. So, we can start using it in the loop function.

The below two lines are where the temperature and pressure are actually read. When the functions `readTemperature()` and `readPressure()` are called, the processor sends a command to the barometer, which measures the pressure or temperature, and returns the result to the processor.

Reading to variables

```
float t = readTemperature();
float p = readPressure();
```

In the example, the values are printed, and then this is followed by a delay of 1000 ms, so that the loop will repeat roughly once a second.

Printing the variables

```
Serial.print("Pressure: ");
Serial.print(p);
Serial.print("hPa\ttemperature: ");
Serial.print(t);
Serial.println("*C\n");


delay(1000);
```

Using the data

We can also use the data in the code, rather than just to print it or save it. For example, we could make a code that detects if the pressure drops by a certain amount, and for instance turn the LED on. Or anything else you'd like to do. Let's try turning the on-board LED on.

To implement this, we need to slighly modify the code in the example. First, let's start

tracking the previous pressure value. To create **global variables**, i.e. ones that don't only exist while we are executing a specific function, you can simply write them outside any specific function. The variable previousPressure is updated on each cycle of the loop function, right at the end. This way we keep track of the the old value, and can compare it to the newer value.

We can use an if-statement to compare the old and new values. In the code below, the idea is that if the previous pressure is 0.1 hPa lower than the new value, we will turn the LED on, and otherwise the LED is kept off.

Reacting to pressure drops

```
float previousPressure = 1000;

void loop() {

  // read temperature to a float - variable
  float t = readTemperature();

  // read pressure to a float
  float p = readPressure();

  // Print the pressure and temperature
  Serial.print("Pressure: ");
  Serial.print(p);
  Serial.print("hPa\ttemperature: ");
  Serial.print(t);
  Serial.println("*C");

  if(previousPressure - 0.1 > p)
  {
    digitalWrite(LED, HIGH);
  }else{
    digitalWrite(LED, LOW);
  }
```

If you flash this modified loop to the CanSat NeXT, it should both print the variable values like before, but now also look for the pressure drop. The atmospheric pressure drops roughly 0.12 hPa / meter when going up, so if you try to rapidly lifting the CanSat NeXT a meter higher, the LED should turn on for one loop cycle (1 second), and then turn back off. It is probably best to disconnect the USB cable before trying this!

You can also try modifying the code. What happens if the delay is changed? What about if the **hysteresis** of 0.1 hPa is changed, or even totally removed?

In the next lesson, we will get even more physical activity, as we try using the other integrated sensor IC - the inertial measurement unit.

Click here for the next lesson!

Lesson 3: Sensing the Spin

CanSat NeXT has two sensor ICs on the CanSat NeXT board. One of them is the barometer we used in the last lesson, and the other one is *inertial measurement unit* LSM6DS3. The LSM6DS3 is a 6-axis IMU, which means that it is able to perform 6 different measurements. In this case, it is linear acceleration on three axis (x, y, z) and angular velocity on three axis.

In this lesson, we will look at the IMU example in the library, and also use the IMU to do a small experiment.

Library Example

Let's start by looking at how the library example works. Load it from File -> Examples -> CanSat NeXT -> IMU.

The initial setup is again the same - include the library, initialize serial and CanSat. So, let's focus on the loop. However, the loop also looks really familiar! We read the values just like in the last lesson, only this time there are many more of them.

Reading IMU values

```
float ax = readAccelX();
float ay = readAccelY();
float az = readAccelZ();
float gx = readGyroX();
float gy = readGyroY();
float gz = readGyroZ();
```

> **NOTE**
>
> Each axis is actually read some hundreds of microseconds apart. If you need them to be updated simultaneously, check out the functions readAcceleration and readGyro.

After reading the values, we can print them as usually. This could be done using Serial.print and println just like in the last lesson, but this example shows an alternative way to print the data, with much less manual writing.

First, a buffer of 128 chars is created. Then this is first initialized so that each value is 0, using memset. After this, the values are written to the buffer using `snprintf()`, which is a function that can be used to write strings with a specified format. Finally, this is just printed with `Serial.println()`.

Fancy Printing

```
char report[128];
memset(report, 0, sizeof(report));
snprintf(report, sizeof(report), "A: %4.2f %4.2f %4.2f    G: %4.2f
%4.2f %4.2f",
    ax, ay, az, gx, gy, gz);
Serial.println(report);
```

If the above feels confusing, you can just use the more familiar style using print and println. However, this gets a bit annoying when there are many values to print.

Regular Printing

```
Serial.print("Ax:");
Serial.println(ay);
// etc
```

Finally, there is again a short delay before starting the loop again. This is mainly there to ensure that the output is readable - without a delay the numbers would be changing so fast that it is hard to read them.

The acceleration is read in Gs, or multiples of $9.81\ \mathrm{m/s^2}$. The angular velocity is in units of $\mathrm{mrad/s}$.

EXERCISE

Try to identify the axis based on the readings!

Free Fall detection

As an exercise, let's try to detect if the device is in free fall. The idea is that we would throw the board in the air, CanSat NeXT would detect the free fall and turn the LED on for couple of seconds after detecting a free fall event, so that we can tell that our check had triggered even after again catching it.

We can keep the setup just like it was, and just focus on the loop. Let's clear the old loop function, and start fresh. Just for fun, let's read the data using the alternative method.

Read Acceleration

```
float ax, ay, az;
readAcceleration(ax, ay, az);
```

Let's define free fall as an event when the total acceleration is below a treshold value. We can calculate the total acceleration from the individual axis as

$$a = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

Which would look in code something like this.

Calculating total acceleration

```
float totalSquared = ax*ax+ay*ay+az*az;
float acceleration = Math.sqrt(totalSquared);
```

And while this would work, we should note that calculating the square root is really slow computationally, so we should avoid doing it if possible. After all, we could just calculate

$$a^2 = a_x^2 + a_y^2 + a_z^2$$

and compare this to a predefined treshold.

Calculating total acceleration squared

```
float totalSquared = ax*ax+ay*ay+az*az;
```

Now that we have a value, let's start controlling the LED. We could have the LED on always when the total acceleration is below a treshold, however reading it would be easier if the LED stayed on for a while after detection. One way to do this is to make another variable, let's call it LEDOnTill, where we simply write the time to where we want to keep the LED on.

Timer variable

```
unsigned long LEDOnTill = 0;
```

Now we can update the timer if we detect a free fall event. Let's use treshold of 0.1 for now. Arduino provides a function called `millis()`, which returns the time since the program started in milliseconds.

Updating the timer

```
if(totalSquared < 0.1)
{
LEDOnTill = millis() + 2000;
}
```

Finally, we can just check if the current time is more or less than the specified `LEDOnTill`, and control the LED based on that. Here is what the new loop function looks like:

Free fall detecting loop function

```
unsigned long LEDOnTill = 0;

void loop() {
  // Read Acceleration
  float ax, ay, az;
  readAcceleration(ax, ay, az);

  // Calculate total acceleration (squared)
  float totalSquared = ax*ax+ay*ay+az*az;

  // Update the timer if we detect a fall
  if(totalSquared < 0.1)
  {
    LEDOnTill = millis() + 2000;
  }

  // Control the LED based on the timer
  if(LEDOnTill >= millis())
  {
    digitalWrite(LED, HIGH);
  }else{
```

Trying out this program, you can see how fast it now reacts since we don't have a delay in the loop. The LED turns on immediately after leaving the hand when being thrown.

> **EXERCISES**
>
> 1. Try reintroducing the delay in the loop function. What happens?
> 2. Currently we don't have any printing in the loop. If you just add a print statement to the loop, the output will be really difficult to read and the printing will slow down the loop cycle time significantly. Can you come up with way to only print once a second, even if the loop is running continuously? Tip: look at how the LED timer was implemented
> 3. Create your own experiment, using either the acceleration or spinning to detect some type of event.

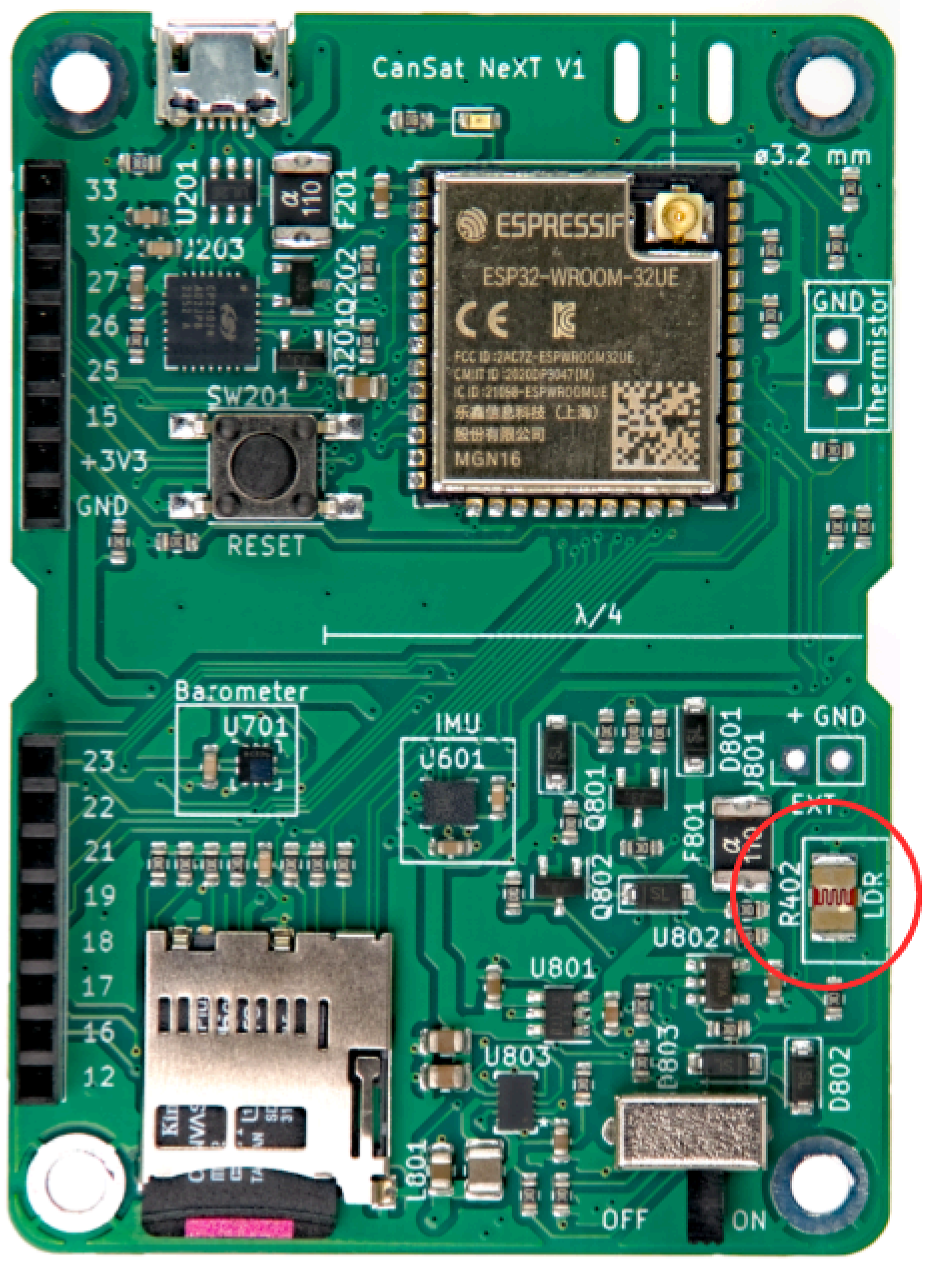
In the next lesson, we will leave the digital domain and try using a different style of sensor - an analogue light meter.

Click here for the next lesson!

Lesson 4: Resistance isn't Futile

So far we have focused on using digital sensor devices to get values directly in SI units. However, electrical devices make the measurement usually in an indirect way, and the conversion to the desired units is then done afterwards. This was done previously by the sensor devices themselves (and by the CanSat NeXT library), but many sensors we use are much more simple. One type of analogue sensors is resistive sensors, where the resistance of a sensor element changes depending on some phenomena. Resistive sensors exist for a multitude of quantities - including force, temperature, light intensity, chemical concentrations, pH, and many others.

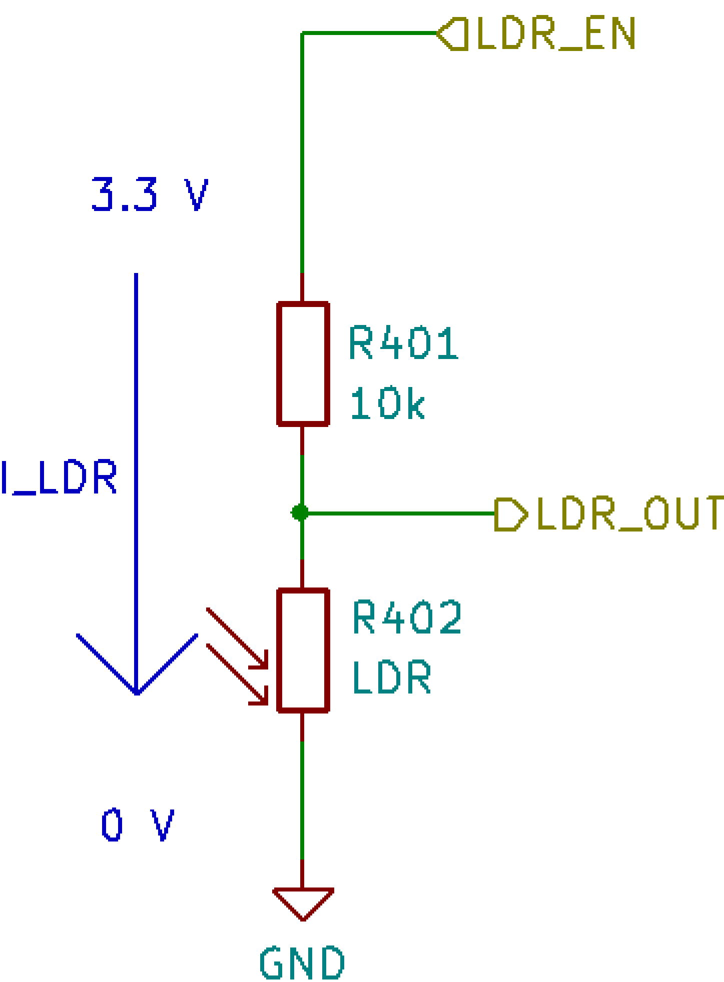
In this lesson, we will be using the light-dependant resistor (LDR) on the CanSat NeXT board to measure surrounding light intensity. While the themistor is used in a very similar way, that will be the focus of a future lesson. The same skills apply directly to using the LDR and thermistor, as well as many other resistive sensors.



Physics of Resistive Sensors

Instead of jumping directly to the software, let's take a step back and discuss how reading a resistive sensor generally works. Consider the schematic below. The voltage at LDR_EN is 3.3 volts (operating voltage of the processor), and we have two resistors

connected in series on its path. One of these is the **LDR** (R402), while the other one is a **reference resistor** (R402). The resistance of the reference resistor is 10 kilo-ohms, while the resistance of the LDR varies between 5-300 kilo-ohms depending on the light conditions.



Since the resistors are connected in series, the total resistance is

$$R = R_{401} + R_{LDR},$$

and the current through the resistors is

$$I_{LDR} = \frac{V_{OP}}{R},$$

where V_{OP} is the operational voltage of the MCU. Remember that the current has to be the same through both of the of the resistors. Therefore we can calculate the voltage drop over the LDR as

$$V_{LDR} = R_{LDR} * I_{LDR} = V_{OP} \frac{R_{LDR}}{R_{401} + R_{LDR}}.$$

And this voltage drop is the voltage of the LDR that we can measure with an analog-to-digital converter. Usually this voltage can be directly correlated or calibrated to correspond to measured values, like for example from voltage to temperature or brightness. However, sometimes it is desirable to first calculate the measured resistance. If necessary, it can be calculated as:

$$R_{LDR} = \frac{V_{LDR}}{I_{LDR}} = \frac{V_{LDR}}{V_{OP}}(R_{401} + R_{LDR}) = R_{401} \frac{\frac{V_{LDR}}{V_{OP}}}{1 - \frac{V_{LDR}}{V_{OP}}}$$

Reading the LDR in Practice

Reading the LDR or other resistive sensors is very easy, as we just need to query the analog-to-digital converter for the voltage. Let's start this time a new Arduino Sketch from scratch. File -> New Sketch.

First, let's start the sketch like before by including the library. This is done at the beginning of the sketch. In the setup, start the serial and initialize CanSat, just like before.

Basic Setup

```
#include "CanSatNeXT.h"
```

A basic loop to read the LDR isn't much more complicated. The resistors R401 and R402 are already on the board, and we just need to read the voltage from their common node. Let's read the ADC value and print it.

Basic LDR loop

```
void loop() {
    int value = analogRead(LDR);
    Serial.print("LDR value:");
    Serial.println(value);
    delay(200);
}
```

With this program, the values clearly react to lighting conditions. We get lower values when the LDR is exposed to light, and higher values when it is darker. However, the values are in hundreds and thousands, not in an expected voltage range. This is because we are now reading the direct output of the ADC. Each bit represent a voltage comparison ladder being one or zero depending on the voltage. The values are now 0-4095 (2^12-1) depending on the input voltage. Again, this direct measurement is probably what you want to use if you are doing something like detecting pulses with the LDR, but quite often regular volts are nice to work with. While calculating the voltage yourself is a good exercise, the library includes a conversion function that also considers the non-linearity of the ADC, meaning that the output is more accurate than from a simple linear conversion.

Reading the LDR voltage

```
void loop() {
    float LDR_voltage = analogReadVoltage(LDR);
    Serial.print("LDR value:");
    Serial.println(LDR_voltage);
    delay(200);
```

NOTE

This code is compatible with the serial plotter in Arduino Code. Try it out!

EXERCISE

It could be useful to detect the CanSat having been deployed from the rocket, so that for instance the parachute could be deployed at the right time. Can you write a program that detects a simulated deployment? Simulate the launch by first covering the LDR (rocket integration) and then uncovering it (deployment). The program could output the deployment to the terminal, or blink an LED to show that the deployment happened.

The next lesson is about using the SD-card to store measurements, settings, and more!

Click here for the next lesson!

Lesson 5: Saving Bits & Bytes

Sometimes getting the data directly to a PC isn't feasible, like when we are throwing the device around, launching it with a rocket, or taking measurements in hard-to-reach places. In such cases, it is best to save the measured data to an SD card for further processing later. Additionally, the SD card can also be used to store settings - for example we could have some type of treshold setting or address settings stored on the SD card.

SD card in CanSat NeXT library

CanSat NeXT library supports a large range of SD card operations. It can be used to save and read files, but also to create directories and new files, move them around or even delete them. All of these could be useful in various circumstances, but let's keep the focus here on the two basic things - reading a file, and writing data to a file.

> **NOTE**
>
> If you want full control of the filesystem, you can find the commands from the Library Specification or from the library example "SD_advanced".

As an exercise, let's modify the code from the last lesson so that instead of writing the LDR measurements to the serial, we will save them on the SD card.

First, let's define the name of the file we will use. Let's add it before the setup function as a **global variable**.

Modified Setup

```
#include "CanSatNeXT.h"

const String filepath = "/LDR_data.csv";

void setup() {
  Serial.begin(115200);
  CanSatInit();
}
```

Now that we have a filepath, we can write to the SD card. We need just two lines to do it. The best command to use for saving measurement data is `appendFile()`, which just takes the filepath, and writes the new data at the end of the file. If the file doesn't exist, it creates it. This makes using the command very easy (and safe). We can just directly add the data to it, and then follow that with a line change so that the data is easier to read. And that's it! Now we are storing the measurements.

Saving LDR data to the SD card

```
void loop() {
  float LDR_voltage = analogReadVoltage(LDR);
  Serial.print("LDR value:");
  Serial.println(LDR_voltage);
  appendFile(filepath, LDR_voltage);
  appendFile(filepath, "\n");
  delay(200);
}
```

By default, the `appendFile()` command stores floating point numbers with two values after the decimal point. For more specific functionality, you could first create a string in the sketch, and use command `appendFile()` to store that string to the SD card. So for

example:

Saving LDR data to the SD card

```
void loop() {
  float LDR_voltage = analogReadVoltage(LDR);

  String formattedString = String(LDR_voltage, 6) + "\n";
  Serial.print(formattedString);
  appendFile(filepath, formattedString);

  delay(200);
}
```

Here the final string is made first, with the `String(LDR_voltage, 6)` specifying that we want 6 decimals after the point. We can use the same string for printing and storing the data. (As well as transmitting via radio)

Reading Data

It is quite often useful to store something on the SD card for future use in the program as well. These could be for example settings about the current state of the device, so that if the program resets, we can load the current status again from the SD card instead of starting from default values.

To demonstrate this, add on PC a new file to the SD card called "delay_time", and write a number into the file, like 200. Let's try to replace the statically set delay time in our program with a setting read from a file.

Let's try to read the setting file in the setup. First, let's introduce a new global variable. I gave it a default value of 1000, so that if we don't manage to modify the delay time, this is now the default setting.

In the setup, we should first check if the file even exists. This can be done using command `fileExists()`. If it doesn't let's just use the default value. After this, the data can be read using `readFile()`. However, we should note that it is a string - not an integer like we need it to be. So, let's convert it using Arduino command `toInt()`. Finally, we check if the conversion was succesful. If it wasn't, the value will be zero, in which case we will just keep using the default value.

Reading a setting in the setup

```
#include "CanSatNeXT.h"

const String filepath = "/LDR_data.csv";
const String settingFile = "/delay_time";

int delayTime = 1000;

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

  if(fileExists(settingFile))
  {
    String contents = readFile(settingFile);
    int value = contents.toInt();
    if(value != 0){
      delayTime = value;
    }
  }
}
```

Finally, don't forget to change the delay in the loop to use the new variable.

Dynamically set delay value

```
void loop() {
  float LDR_voltage = analogReadVoltage(LDR);

  String formattedString = String(LDR_voltage, 6) + "\n";
  Serial.print(formattedString);
  appendFile(filepath, formattedString);

  delay(delayTime);
}
```

You can now try changing the value on the SD card, or even removing the SD card, in which case it should now use the default value for the delay length.

NOTE

To rewrite the setting in your program, you can use command writeFile. It works just like appendFile, but overwrites any existing data.

EXERCISE

Continue from your solution to the exercise in lesson 4, so that the state is maintained even if the device is reset. I.e. store the current state on the SD card and read it in the setup. This would simulate a scenario where your CanSat suddenly resets in flight or before the flight, and with this program you would still get a succesful flight.

In the next lesson, we will look at using radio to transmit data between processors. You should have some type of antenna in your CanSat NeXT and the groundstation before starting those exercises. If you haven't already, take a look at the tutorial for building a basic antenna: Building an antenna.

Click here for the next lesson!

Lesson 6: Phoning Home

Now we have taken measurements and also saved them on an SD-card. The next logical step is to transmit them wirelessly to the ground, which enables completely new world in terms of measurements and experiments we can perform. For example, trying out the zero-g flight with IMU would have been quite a bit more interesting (and easy to calibrate), if we could have seen the data in real time. Let's take a look at how we can do that!

In this lesson, we will send measurements from CanSat NeXT to the groundstation receiver. Later on, we will also take a look at commanding the CanSat with messaged sent by the groundstation.

Antennas

Before starting this lesson, please make sure you have some type of antenna connected to the CanSat NeXT board and the groundstation.

> **NOTE**
>
> You should never try transmitting anything without an antenna. Not only will it probably not work, there is a possibility that the reflected power will damage the transmitter.

Since we are using 2.4 GHz band, which is shared by systems like Wi-Fi, Bluetooth, ISM, drones etc, there are a lot of commercial antennas available. Most Wi-Fi antennas actually work really well with CanSat NeXT, but you will often need an adapter to connect them to the CanSat NeXT board. We have also tested some adapter models, which are available in the webstore.

More information about antennas can be found in the hardware documentation: Communication and Antennas. This article also has instructions on building your own antenna from the materials in the CanSat NeXT kit.

Sending Data

With the discussion about antennas out of the way, let's start sending some bits. We will start again by looking at the setup, which has actually a key difference this time - we've added a number as an **argument** to the `CanSatInit()` command.

Setup for transmission

```
#include "CanSatNeXT.h"

void setup() {
  Serial.begin(115200);
  CanSatInit(28);
}
```

Passing a number value to `CanSatInit()` tells the CanSat NeXT that we want to now use the radio. The number indicates the value of the last byte of the MAC address. You can think of it as a key to your specific network - you can only communicate to CanSats that share the same key. This number should be shared between your CanSat NeXT and your groundstation. You can pick your favorite number between 0 and 255. I picked 28, since it is perfect.

With the radio initialized, transmitting the data is really simple. It actually operates just like the `appendFile()` that we used in the last lesson - you can add any value and it will transmit it in a default format, or you can use a formatted string and send that instead.

Transmitting the data

```
void loop() {
  float LDR_voltage = analogReadVoltage(LDR);
  sendData(LDR_voltage);
  delay(100);
}
```

With this simple code, we are now transmitting the LDR measurement almost 10 times per second. Next let's take a look at how to receive it.

> **NOTE**
>
> Those familiar to low-level programming might feel more comfortable sending the data in binary form. Don't worry, we've got you covered. The binary commands are listed in the Library Specification.

Receiving Data

This code should now be programmed to another ESP32. Usually it is the second controller board included in the kit, however pretty much any other ESP32 will work as well - including another CanSat NeXT.

The setup code is exactly the same as before. Just remember to change the radio key to your favorite number.

Setup for reception

```
#include "CanSatNeXT.h"
```

However, after that things get a bit different. We make a completely empty loop function! This is because we have actually nothing to do in the loop, but instead the receiving is done through **callbacks**.

Setting up a callback

```
void loop() {
  // We have nothing to do in the loop.
}

// This is a callback function. It is run every time the radio
receives data.
void onDataReceived(String data)
{
  Serial.println(data);
}
```

Where as the function `setup()` runs just once at the start and `loop()` runs continuously, the function `onDataReceived()` runs only when the radio has received new data. This way, we can handle the data in the callback function. In this example, we just print it, but we could have also modified it however we wanted.

Note that the `loop()` function doesn't *have* to be empty, you can actually use it for whatever you want with one caviat - delays should be avoided, as the `onDataReceived()` function will also not run until the delay is over.

If you now have both programs running on different boards at the same time, there should be quite a lot of measurements being sent wirelessly to your PC.

For binary oriented folks - you can use the callback function onBinaryDataReceived.

Real time Zero-G

Just for fun, let's repeat the zero-g experiment but with radios. The receiver code can stay the same, as actually does the setup in the CanSat code.

As a reminder, we made a program in the IMU lesson that detected free-fall and turned an a LED in this scenario. Here is the old code:

Free fall detecting loop function

```
unsigned long LEDOnTill = 0;

void loop() {
  // Read Acceleration
  float ax, ay, az;
  readAcceleration(ax, ay, az);

  // Calculate total acceleration (squared)
  float totalSquared = ax*ax+ay*ay+az*az;

  // Update the timer if we detect a fall
  if(totalSquared < 0.1)
  {
    LEDOnTill = millis() + 2000;
  }

  // Control the LED based on the timer
  if(LEDOnTill >= millis())
  {
    digitalWrite(LED, HIGH);
  }else{
    digitalWrite(LED, LOW);
  }
```

It is tempting to just add the `sendData()` directly to the old example, however we need to consider the timing. We don't usually want to send messages more than ~20 times per second, but on the other hand we want to the loop to be running continuously so that the LED still turns on.

We need to add another timer - this time to send data every 50 milliseconds. The timer is done by comparing the current time to the current time to the last time when data was sent. The last time is then updated each time data is sent. Take also a look at how the string is made here. It could also be transmitted in parts, but this way it is received as a single message, instead of multiple messages.

Free fall detection + data transmission

```
unsigned long LEDOnTill = 0;

unsigned long lastSendTime = 0;
const unsigned long sendDataInterval = 50;


void loop() {

  // Read Acceleration
  float ax, ay, az;
  readAcceleration(ax, ay, az);

  // Calculate total acceleration (squared)
  float totalSquared = ax*ax+ay*ay+az*az;

  // Update the timer if we detect a fall
  if(totalSquared < 0.1)
  {
    LEDOnTill = millis() + 2000;
  }
```

The data format here is actually compatible again with the serial plotter - looking at that data makes it quite clear why we were able to detect the free fall earlier so cleanly - the values really do drop to zero as soon as the device is dropped or thrown.

This concludes the lessons for now. We will add more soon, but in the meanwhile you can find more information about using CanSat NeXT from the other Arduino examples, our blog and the software and hardware documentation. I would love to hear your feedback and ideas regarding CanSat NeXT and these materials, so don't hesitate to contact me at samuli@kitsat.fi.

CanSat NeXT Software

The recommended way to use CanSat NeXT is with the CanSat NeXT Arduino library, available from the Arduino library manager and Github. Prior to installing the CanSat NeXT library, you have to install Arduino IDE and ESP32 board support.

Getting started

Install Arduino IDE

If you haven't already, download and install the Arduino IDE from the official website https://www.arduino.cc/en/software.

Add ESP32 support

CanSat NeXT is based on the ESP32 microcontroller, which is not included in the Arduino IDE default installation. If you haven't used ESP32 microcontrollers with Arduino before, the support for the board needs to be installed first. It can be done in Arduino IDE from *Tools->board->Board Manager* (or just press (Ctrl+Shift+B) anywhere). In the board manager, search for ESP32, and install the esp32 by Espressif.

Install Cansat NeXT library

The CanSat NeXT library can be downloaded from the Arduino IDE's Library Manager from *Sketch > Include Libraries > Manage Libraries*.

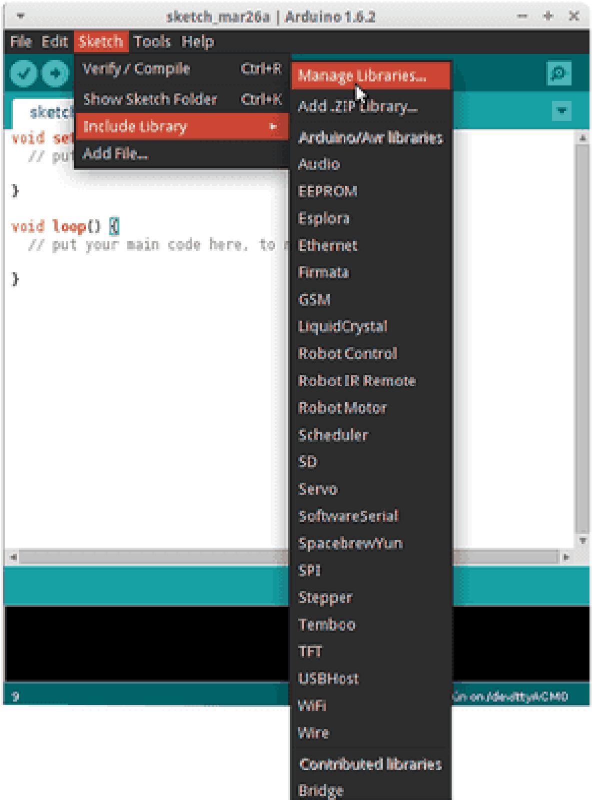


Image source: Arduino Docs, https://docs.arduino.cc/software/ide-v1/tutorials/installing-libraries

In the Library Manager search bar, type "CanSatNeXT" and choose "Install". If the IDE asks if you want to also install the dependencies, click yes.

Manual installation

The library is also hosted on its own GitHub repository and can be cloned or downloaded and installed from source.

In this case, you need to extract the library and move it in to the directory where Arduino IDE can find it. You can find the exact location in *File > Preferences > Sketchbook*.

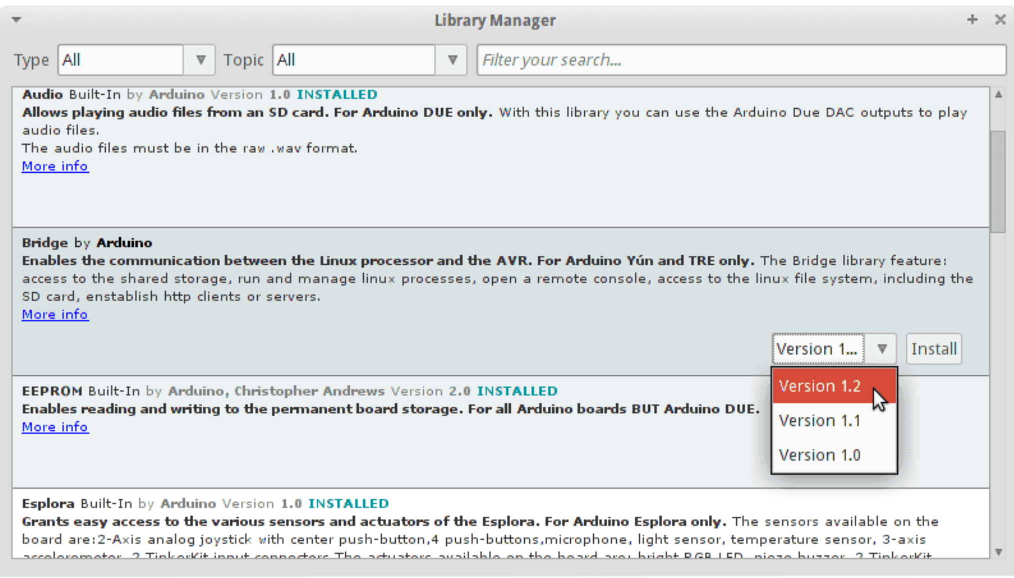


Image source: Arduino Docs, https://docs.arduino.cc/software/ide-v1/tutorials/installing-libraries

Connecting to PC

After installing the CanSat NeXT software library, you can plug in the CanSat NeXT to your computer. In case it is not detected, you may need to install the necessary drivers first. The driver installation is done automatically in most cases, however, on some PCs it needs to be done manually. Drivers can be found on the Silicon Labs website: https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers For additional help with setting up the ESP32, refer to the following tutorial: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/establish-serial-connection.html

You are ready to go!

You can now find CanSatNeXT examples from the Arduino IDE from *File->Examples->CanSatNeXT*.

Library specification

Functions

You can use all regular Arduino functionalities with CanSat NeXT, as well as any Arduino libraries. Arduino functions can be found here: https://www.arduino.cc/reference/en/.

CanSat NeXT library adds several easy to use functions for using the different on-board resources, such as sensors, radio and the SD-card. The library comes with a set of example sketches that show how to use these functionalities. The list below also shows all available functions.

System Initialization Functions

CanSatInit

Function	uint8_t CanSatInit(uint8_t macAddress[6])
Return Type	`uint8_t`
Return Value	Returns 0 if initialization was successful, or non-zero if there was an error.
Parameters	
	`uint8_t macAddress[6]`

Function	uint8_t CanSatInit(uint8_t macAddress[6])
	6-byte MAC address shared by the satellite and the ground station. This is an optional parameter - when it is not provided, the radio is not initialized. Used in example sketch: All
Description	This command is found in the `setup()` of almost all CanSat NeXT scripts. It is used to initialize the CanSatNeXT hardware, including the sensors and the SD-card. Additionally, if the `macAddress` is provided, it starts the radio and starts to listen for incoming messages. The MAC address should be shared by the ground station and the satellite. The MAC address can be chosen freely, but there are some non-valid addresses such as all bytes being `0x00`, `0x01`, and `0xFF`. If the init function is called with a non-valid address, it will report the problem to the Serial.

CanSatInit (simplified MAC-address specification)

Function	uint8_t CanSatInit(uint8_t macAddress)
Return Type	`uint8_t`
Return Value	Returns 0 if initialization was successful, or non-zero if there was an error.
Parameters	

Function	uint8_t CanSatInit(uint8_t macAddress)
	`uint8_t macAddress`
	Last byte of the MAC-address, used to differentiate between different CanSat-GS pairs.
Description	This is a simplified version of the CanSatInit with MAC address, which sets the other bytes automatically to a known safe value. This enables the users to differentiate their Transmitter-Receiver pairs with just one value, which can be 0-255.

GroundStationInit

Function	uint8_t GroundStationInit(uint8_t macAddress[6])
Return Type	`uint8_t`
Return Value	Returns 0 if initialization was successful, or non-zero if there was an error.
Parameters	
	`uint8_t macAddress[6]`
	6-byte MAC address shared by the satellite and the ground station.
Used in	Groundstation receive

Function	uint8_t GroundStationInit(uint8_t macAddress[6])
example sketch	
Description	This is a close relative of the CanSatInit function, but it always requires the MAC address. This function only initializes the radio, not other systems. The ground station can be any ESP32 board, including any devboard or even another CanSat NeXT board.

GroundStationInit (simplified MAC-address specification)

Function	uint8_t GroundStationInit(uint8_t macAddress)
Return Type	`uint8_t`
Return Value	Returns 0 if initialization was successful, or non-zero if there was an error.
Parameters	
	`uint8_t macAddress`
	Last byte of the MAC-address, used to differentiate between different CanSat-GS pairs.
Description	This is a simplified version of the GroundStationInit with MAC

Function	uint8_t GroundStationInit(uint8_t macAddress)
	address, which sets the other bytes automatically to a known safe value. This enables the users to differentiate their Transmitter-Receiver pairs with just one value, which can be 0-255.

IMU Functions

readAcceleration

Function	uint8_t readAcceleration(float &x, float &y, float &z)
Return Type	`uint8_t`
Return Value	Returns 0 if measurement was successful.
Parameters	
	`float &x, float &y, float &z`
	`float &x`: Address of a float variable where the x-axis data will be stored.
Used in example sketch	IMU

Function	uint8_t readAcceleration(float &x, float &y, float &z)
Description	This function can be used to read acceleration from the on-board IMU. The parameters are addresses to float variables for each axis. The example IMU shows how to use this function to read the acceleration. The acceleration is returned in units of G (9.81 m/s).

readAccelX

Function	float readAccelX()
Return Type	`float`
Return Value	Returns linear acceleration on X-axis in units of G.
Used in example sketch	IMU
Description	This function can be used to read acceleration from the on-board IMU on a specific axis. The example IMU shows how to use this function to read the acceleration. The acceleration is returned in units of G (9.81 m/s).

readAccelY

Function	float readAccelY()
Return Type	`float`
Return Value	Returns linear acceleration on Y-axis in units of G.
Used in example sketch	IMU
Description	This function can be used to read acceleration from the on-board IMU on a specific axis. The example IMU shows how to use this function to read the acceleration. The acceleration is returned in units of G (9.81 m/s).

readAccelZ

Function	float readAccelZ()
Return Type	`float`
Return Value	Returns linear acceleration on Z-axis in units of G.

Function	float readAccelZ()
Used in example sketch	IMU
Description	This function can be used to read acceleration from the on-board IMU on a specific axis. The example IMU shows how to use this function to read the acceleration. The acceleration is returned in units of G (9.81 m/s).

readGyro

Function	uint8_t readGyro(float &x, float &y, float &z)
Return Type	`uint8_t`
Return Value	Returns 0 if measurement was successful.
Parameters	
	`float &x, float &y, float &z`
	`float &x`: Address of a float variable where the x-axis data will be stored.
Used in	IMU

Function	uint8_t readGyro(float &x, float &y, float &z)
example sketch	
Description	This function can be used to read angular velocity from the on-board IMU. The parameters are addresses to float variables for each axis. The example IMU shows how to use this function to read the angular velocity. The angular velocity is returned in units mrad/s.

readGyroX

Function	float readGyroX()
Return Type	`float`
Return Value	Returns angular velocity on X-axis in units of mrad/s.
Used in example sketch	IMU
Description	This function can be used to read angular velocity from the on-board IMU on a specific axis. The parameters are addresses to float variables for each axis. The angular velocity is returned in units mrad/s.

readGyroY

Function	float readGyroY()
Return Type	`float`
Return Value	Returns angular velocity on Y-axis in units of mrad/s.
Used in example sketch	IMU
Description	This function can be used to read angular velocity from the on-board IMU on a specific axis. The parameters are addresses to float variables for each axis. The angular velocity is returned in units mrad/s.

readGyroZ

Function	float readGyroZ()
Return Type	`float`
Return Value	Returns angular velocity on Z-axis in units of mrad/s.

Function	float readGyroZ()
Used in example sketch	IMU
Description	This function can be used to read angular velocity from the on-board IMU on a specific axis. The parameters are addresses to float variables for each axis. The angular velocity is returned in units mrad/s.

Barometer Functions

readPressure

Function	float readPressure()
Return Type	`float`
Return Value	Pressure in mbar
Parameters	None
Used in example sketch	Baro
Description	This function returns pressure as reported by the on-board

Function	float readPressure()
	barometer. The pressure is in units of millibar.

readTemperature

Function	float readTemperature()
Return Type	`float`
Return Value	Temperature in Celsius
Parameters	None
Used in example sketch	Baro
Description	This function returns temperature as reported by the on-board barometer. The unit of the reading is Celsius. Note that this is the internal temperature measured by the barometer, so it might not reflect the external temperature.

SD Card / File System Functions

SDCardPresent

Function	bool SDCardPresent()
Return Type	`bool`
Return Value	Returns true if it detects an SD-card, false if not.
Parameters	None
Used in example sketch	SD_advanced
Description	This function can be used to check if the SD-card is mechanically present. The SD-card connector has a mechanical switch, which is read when this function is called. Returns true or false depending on whether the SD-card is detected.

appendFile

Function	uint8_t appendFile(String filename, T data)
Return Type	`uint8_t`
Return Value	Returns 0 if write was successful.
Parameters	
	`String filename`: Address of the file to be appended. If the file doesn't exist, it is created.
	`T data`: Data to be appended at the end of the file.
Used in example sketch	SD_write
Description	This is the basic write function used to store readings to the SD-card.

printFileSystem

Function	void printFileSystem()
Return Type	`void`
Parameters	None

Function	void printFileSystem()
Used in example sketch	SD_advanced
Description	This is a small helper function to print names of files and folders present on the SD-card. Can be used in development.

newDir

Function	void newDir(String path)
Return Type	`void`
Parameters	
	`String path`: Path of the new directory. If it already exists, nothing is done.
Used in example sketch	SD_advanced
Description	Used to create new directories on the SD-card.

deleteDir

Function	void deleteDir(String path)
Return Type	`void`
Parameters	
	`String path`: Path of the directory to be deleted.
Used in example sketch	SD_advanced
Description	Used to delete directories on the SD-card.

fileExists

Function	bool fileExists(String path)
Return Type	`bool`
Return Value	Returns true if the file exists.
Parameters	
	`String path`: Path to the file.
Used in example sketch	SD_advanced

Function	bool fileExists(String path)
Description	This function can be used to check if a file exists on the SD-card.

fileSize

Function	uint32_t fileSize(String path)
Return Type	`uint32_t`
Return Value	Size of the file in bytes.
Parameters	
	`String path`: Path to the file.
Used in example sketch	SD_advanced
Description	This function can be used to read the size of a file on the SD-card.

writeFile

Function	uint8_t writeFile(String filename, T data)
Return Type	`uint8_t`
Return Value	Returns 0 if write was successful.
Parameters	
	`String filename`: Address of the file to be written.
	`T data`: Data to be written to the file.
Used in example sketch	SD_advanced
Description	This function is similar to the `appendFile()`, but it overwrites existing data on the SD-card. For data storage, `appendFile` should be used instead. This function can be useful for storing settings, for example.

readFile

Function	String readFile(String path)
Return Type	`String`
Return Value	All content in the file.
Parameters	
	`String path`: Path to the file.
Used in example sketch	SD_advanced
Description	This function can be used to read all data from a file into a variable. Attempting to read large files can cause problems, but it is fine for small files, such as configuration or setting files.

renameFile

Function	void renameFile(String oldpath, String newpath)
Return Type	`void`

Function	void renameFile(String oldpath, String newpath)
Parameters	
	`String oldpath`: Original path to the file.
	`String newpath`: New path of the file.
Used in example sketch	SD_advanced
Description	This function can be used to rename or move files on the SD-card.

deleteFile

Function	void deleteFile(String path)
Return Type	`void`
Parameters	
	`String path`: Path of the file to be deleted.
Used in example sketch	SD_advanced
Description	This function can be used to delete files from the SD-card.

Radio Functions

onDataReceived

Function	void onDataReceived(String data)
Return Type	`void`
Parameters	
	`String data`: Received data as an Arduino String.
Used in example sketch	Groundstation_receive
Description	This is a callback function that is called when data is received. The user code should define this function, and the CanSat NeXT will call it automatically when data is received.

onBinaryDataReceived

Function	void onBinaryDataReceived(const uint8_t *data, int len)
Return Type	`void`

Function	void onBinaryDataReceived(const uint8_t *data, int len)
Parameters	
	`const uint8_t *data`: Received data as a uint8_t array.
	`int len`: Length of received data in bytes.
Used in example sketch	None
Description	This is similar to the `onDataReceived` function, but the data is provided as binary instead of a String object. This is provided for advanced users who find the String object limiting.

onDataSent

Function	void onDataSent(const bool success)
Return Type	`void`
Parameters	
	`const bool success`: Boolean indicating if data was sent successfully.
Used in	None

Function	void onDataSent(const bool success)
example sketch	
Description	This is another callback function that can be added to the user code if required. It can be used to check if the reception was acknowledged by another radio.

sendData (String variant)

Function	uint8_t sendData(T data)
Return Type	`uint8_t`
Return Value	0 if data was sent (does not indicate acknowledgment).
Parameters	
	`T data`: Data to be sent. Any type of data can be used, but is converted to a string internally.
Used in example sketch	Send_data
Description	This is the main function for sending data between the ground station and the satellite. Note that the return value does not indicate if data was actually received, just that it was sent. The callback

Function	uint8_t sendData(T data)
	`onDataSent` can be used to check if the data was received by the other end.

sendData (Binary variant)

Function	uint8_t sendData(char *data, uint16_t len)
Return Type	`uint8_t`
Return Value	0 if data was sent (does not indicate acknowledgment).
Parameters	
	`char *data`: Data to be sent as a char array.
	`uint16_t len`: Length of the data in bytes.
Used in example sketch	None
Description	A binary variant of the `sendData` function, provided for advanced users who feel limited by the String object.

ADC Functions

adcToVoltage

Function	float adcToVoltage(int value)
Return Type	`float`
Return Value	Converted voltage as volts.
Parameters	
	`int value`: ADC reading to be converted to voltage.
Used in example sketch	AccurateAnalogRead
Description	This function converts an ADC reading to voltage using a calibrated third-order polynomial for more linear conversion. Note that this function calculates the voltage at the input pin, so to calculate the battery voltage, you need to also consider the resistor network.

analogReadVoltage

Function	float analogReadVoltage(int pin)
Return	`float`

Function	float analogReadVoltage(int pin)
Type	
Return Value	ADC voltage as volts.
Parameters	
	`int pin`: Pin to be read.
Used in example sketch	AccurateAnalogRead
Description	This function reads voltage directly instead of using `analogRead` and converts the reading to voltage internally using `adcToVoltage`.

Extension interface

Custom devices can be built and used together with CanSat. These can be used to make interesting projects, which you can find ideas for from our Blog.

CanSat's extension interface features a free UART line, two ADC pins, and 5 free digital I/O pins. Additionally, SPI and I2C lines are available for the extension interface, although they are shared with SD card and the sensor suite, respectively.

The user can also choose to use the UART2 and ADC pins as digital I/O, in case serial communication or analog to digital conversion is not needed in their solution.

Pin number	Pin name	Use as	Notes
12	GPIO12	Digital I/O	Free
15	GPIO15	Digital I/O	Free
16	GPIO16	UART2 RX	Free
17	GPIO17	UART2 TX	Free
18	SPI_CLK	SPI CLK	Co-use with SD card
19	SPI_MISO	SPI MISO	Co-use with SD card
21	I2C_SDA	I2C SDA	Co-use with sensor suite
22	I2C_SCL	I2C SCL	Co-use with sensor suite

Pin number	Pin name	Use as	Notes
23	SPI_MOSI	SPI MOSI	Co-use with SD card
25	GPIO25	Digital I/O	Free
26	GPIO26	Digital I/O	Free
27	GPIO27	Digital I/O	Free
32	GPIO32	ADC	Free
33	GPIO33	ADC	Free

Table: Extension interface pin lookup table. Pin name refers to library pin name.

Communication options

The CanSat library does not include communication wrappers for the custom devices. For UART, I2C and SPI communication between CanSat NeXT and your custom payload device, refer to Arduino's default UART, Wire, and SPI libraries, respectively.

UART

The UART2 line is a good alternative as it serves as an unallocated communication interface for extended payloads.

For sending data through the UART line, please refer to the Arduino

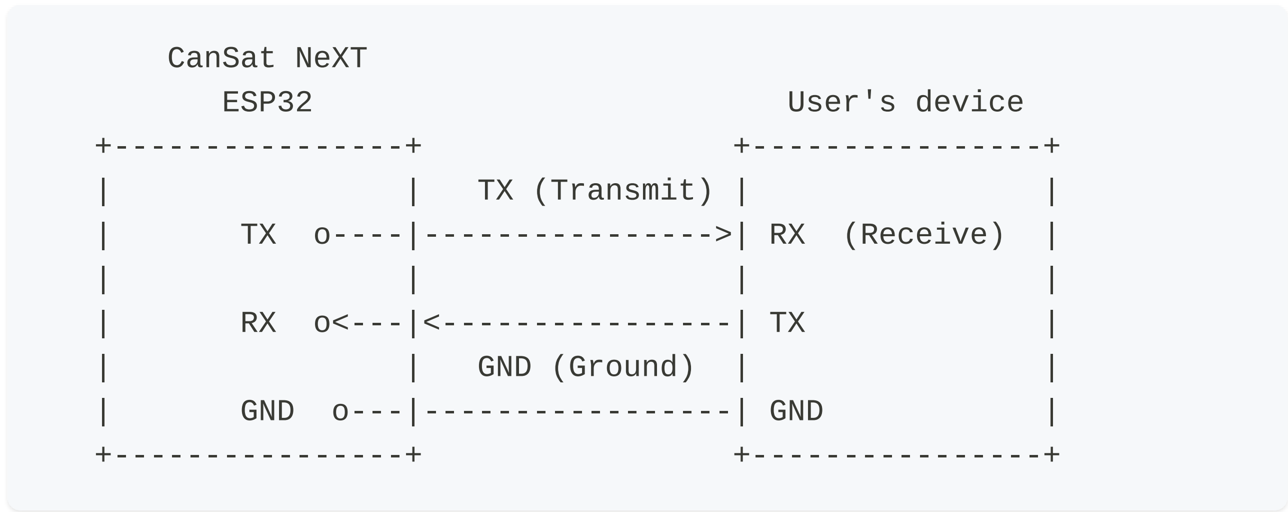


Image: UART protocol in ASCII

I2C

Use of I2C is supported, but the user must bear in mind that another subsystem exists on the line.

With multiple I2C slaves, the user code needs to specify which I2C slave the CanSat is using at a given time. This is distinguished with a slave address, which is unique hexadecimal code to each device and can be found from the subsystem device's data sheet.

SPI

Use of SPI is supported as well, but the user must bear in mind that another subsystem exists on the line.

With SPI, the slave distinction is instead made by specifying a chip select pin. The user must dedicate one of the free GPIO pins to be a chip select for their custom extended payload device. The SD Card's chip select pin is defined in the `CanSatPins.h` library file

as `SD_CS`.

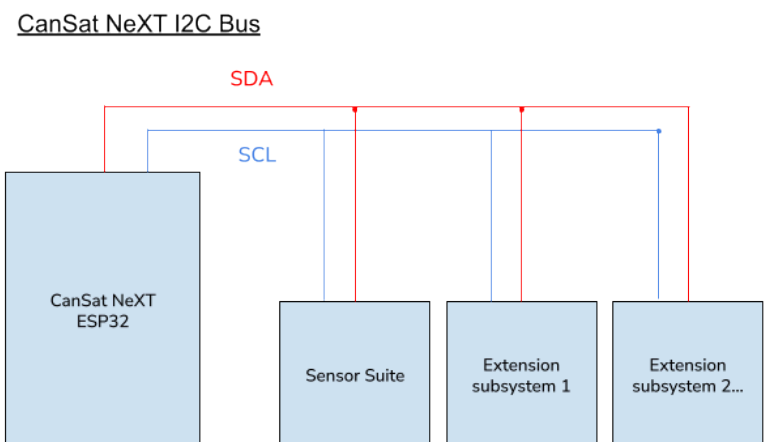


Image: the CanSat NeXT I2C bus featuring several secondary, or "slave" subsystems. In this context, the Sensor suite is one of the slave subsystems.

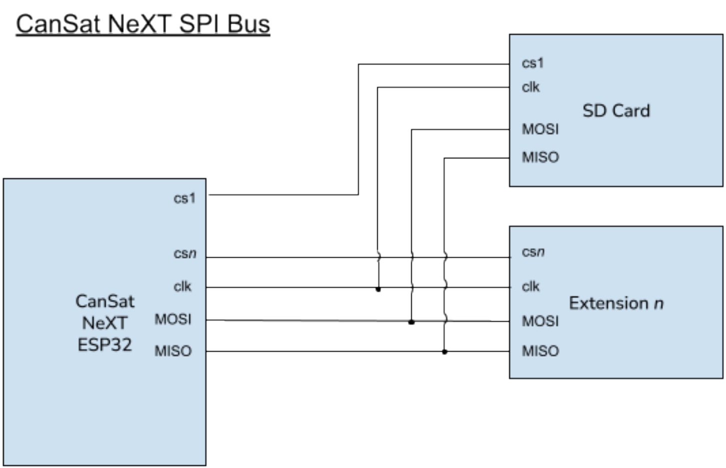


Image: the CanSat NeXT SPI bus configuration when two secondary, or "slave" subsystems

are present. In this context, the SD card is one of the slave subsystems.

CanSat NeXT Hardware

These pages have articles about various hardware aspects of CanSat NeXT, such as articles about the extension header pinout or the power system. The articles currently have quite basic information, but they actively being expanded. If there is some more information you need, don't hesitate to contact samuli@kitsat.fi with your questions and wishes about what to include here next.

If you are new to CanSat NeXT, check out our lessons about using CanSat NeXT. Or, if you are looking for information about the library, take a look at the software specification.

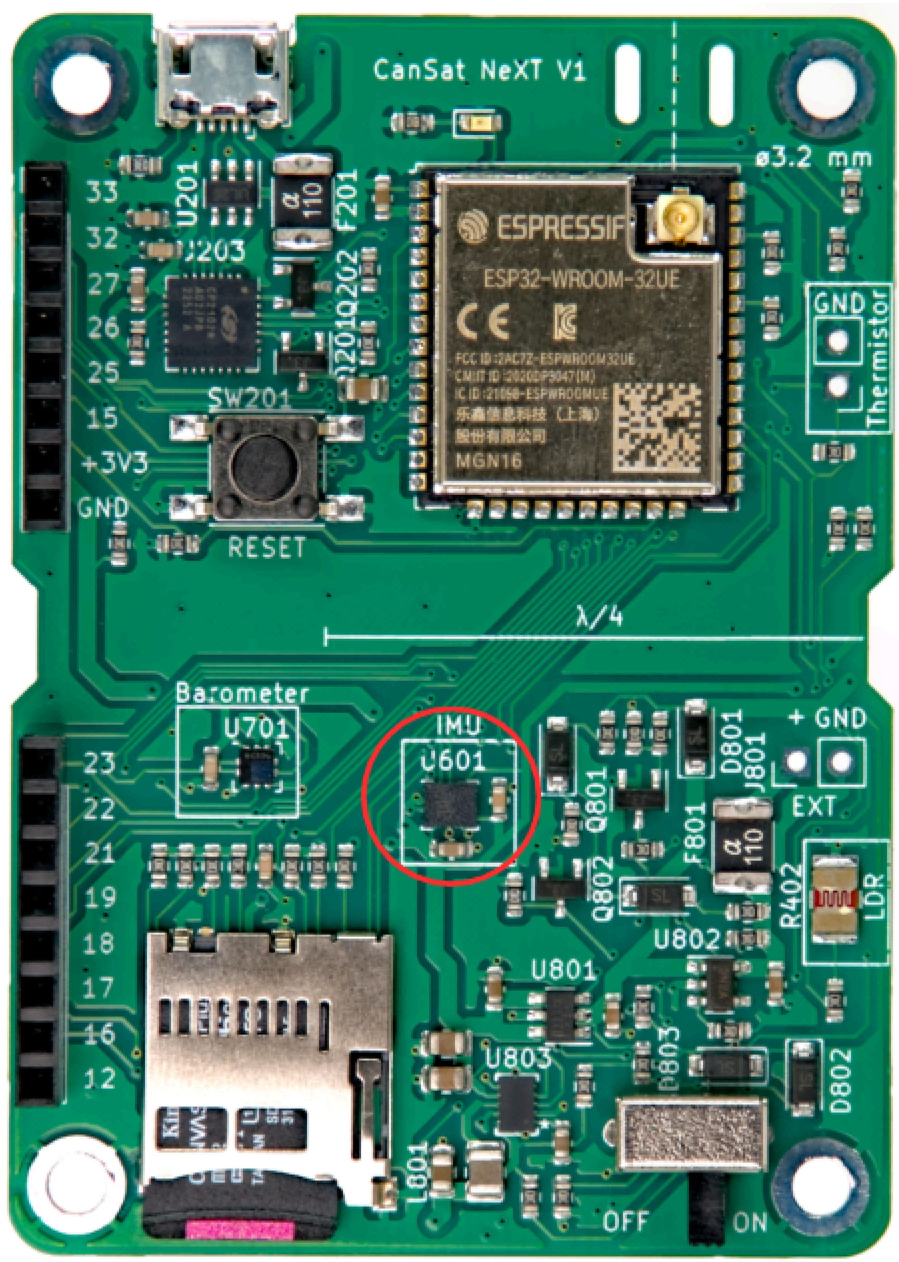
On-Board Sensors

This article introduces the sensors integrated to the CanSat NeXT main board. The use of the sensors is covered in the software documentation, whereas this article provides more information about the sensors themselves.

There are three on-board sensors on the CanSat NeXT main board. These are the IMU LSM6DS3, pressure sensor LPS22HB and the LDR. Additionally, the board has a through-hole slot for adding an external thermistor. As the LPS22HB already has both pressure and temperature measuring capabilities, it theoretically suffices to fulfill the primary mission criteria of the CanSat competitions on its own. However, as it is measuring the internal junction temperature, or basically the temperature of the PCB on that spot, it is not a good atmospheric temperature measurement in most configurations. Additionally, the absolute measurement of the pressure sensor can be supported by the additional data from the IMU accelerometer. The LDR has been added first and foremost to help students learn the concepts regarding analog sensors as the response to stimuli is almost instant, whereas a thermistor takes time to heat up and cool down. That being said, it can also support the creative missions the student will come up with, just like the IMUs accelerometer and gyroscope. Furthermore, in addition to the on-board sensor, the CanSat NeXT encourages the use of additional sensors through the extension interface.

Inertial Measurement Unit

The IMU, LSM6DS3 by STMicroelectronics is an SiP (system-in-package) style MEMS sensor device, integrating an accelerometer, gyroscope and the readout electronics into a small package. The sensor supports SPI and I2C serial interfaces, and also includes an internal temperature sensor.



The LSM6DS3 has switchable acceleration measurement ranges of ±2/±4/±8/±16 G and angular rate measurement ranges of ±125/±250/±500/±1000/±2000 deg/s. The use of a higher range also decreases the resolution of the device.

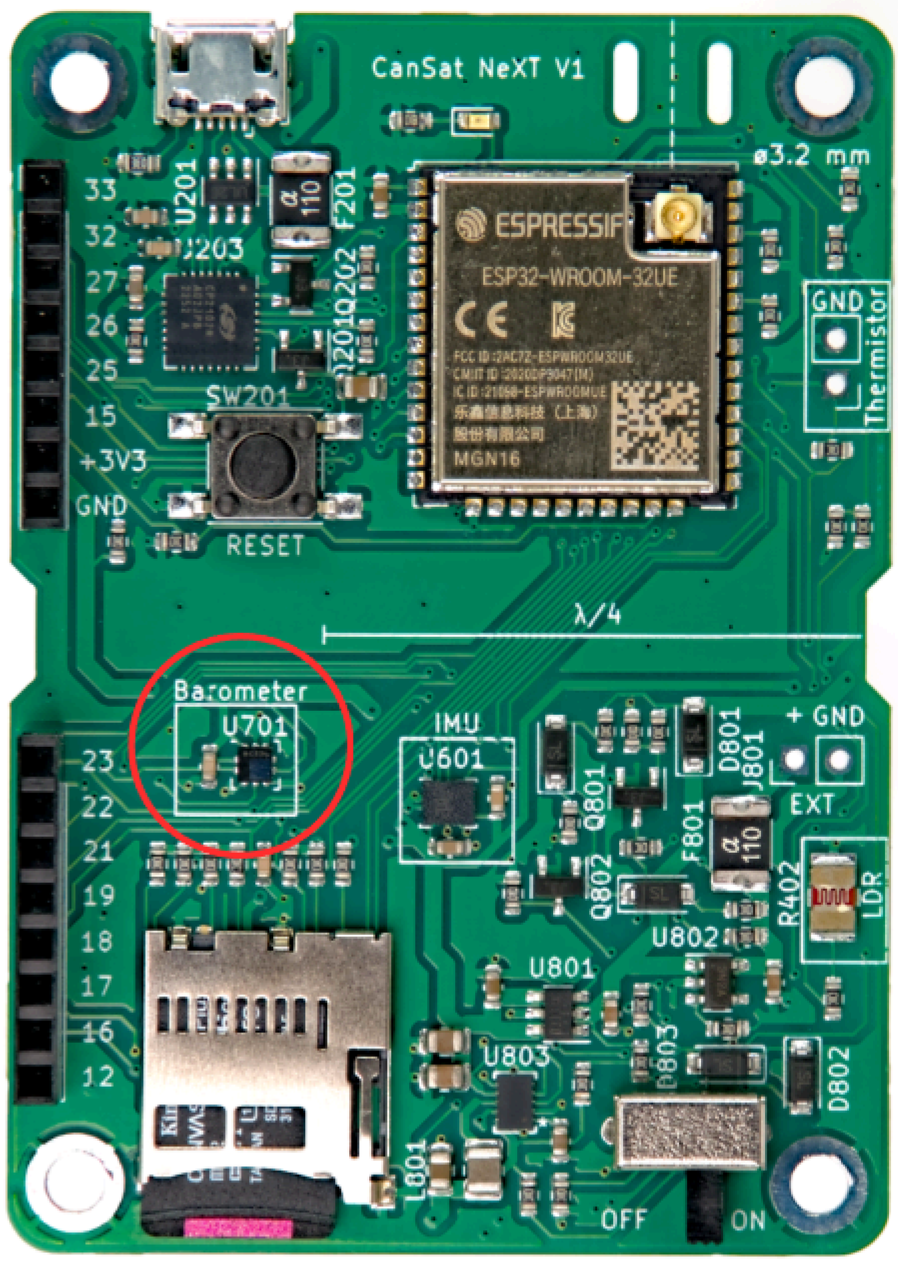
In CanSat NeXT, the LSM6DS3 is used in I2C mode. The I2C address is 1101010b (0x6A), but the next version will add support for modifying the hardware to change the address to 1101011b (0x6B) if an advanced user has a need for using the original address for

something else.

The measurement ranges will be set to maximum by default in the library in order to capture most data from the violent rocket launch. The data ranges are also modifiable by the user.

Barometer

The pressure sensor LPS22HB by STMicroelectronics is another SiP MEMS device, designed for measurement of pressure from 260-1260 hPa. The range it reports data in is significantly larger, but the accuracy of measurements outside that range is questionable. The MEMS pressure sensors work by measuring piezoresistive changes in the sensor diaphragm. As temperature affects the resistance of the piezo element as well, it needs to be compensated. To enable this, the chip also has a relatively accurate junction-temperature sensor as well right next to the piezoresistive element. This temperature measurement can also be read from the sensor, but it has to be kept in mind that it is a measurement of the internal chip temperature, not of the surrounding air.



Similar to the IMU, the LPS22HB can also be communicated with using either SPI or I2C interface. In CanSat NeXT, it is connected to the same I2C interface as the IMU. The I2C address of the LPS22HB is 1011100b (0x5C), but we will add support to change it to 0x5D if desired.

Analog to Digital Converter

This refers to the voltage measurement using the analogRead() command.

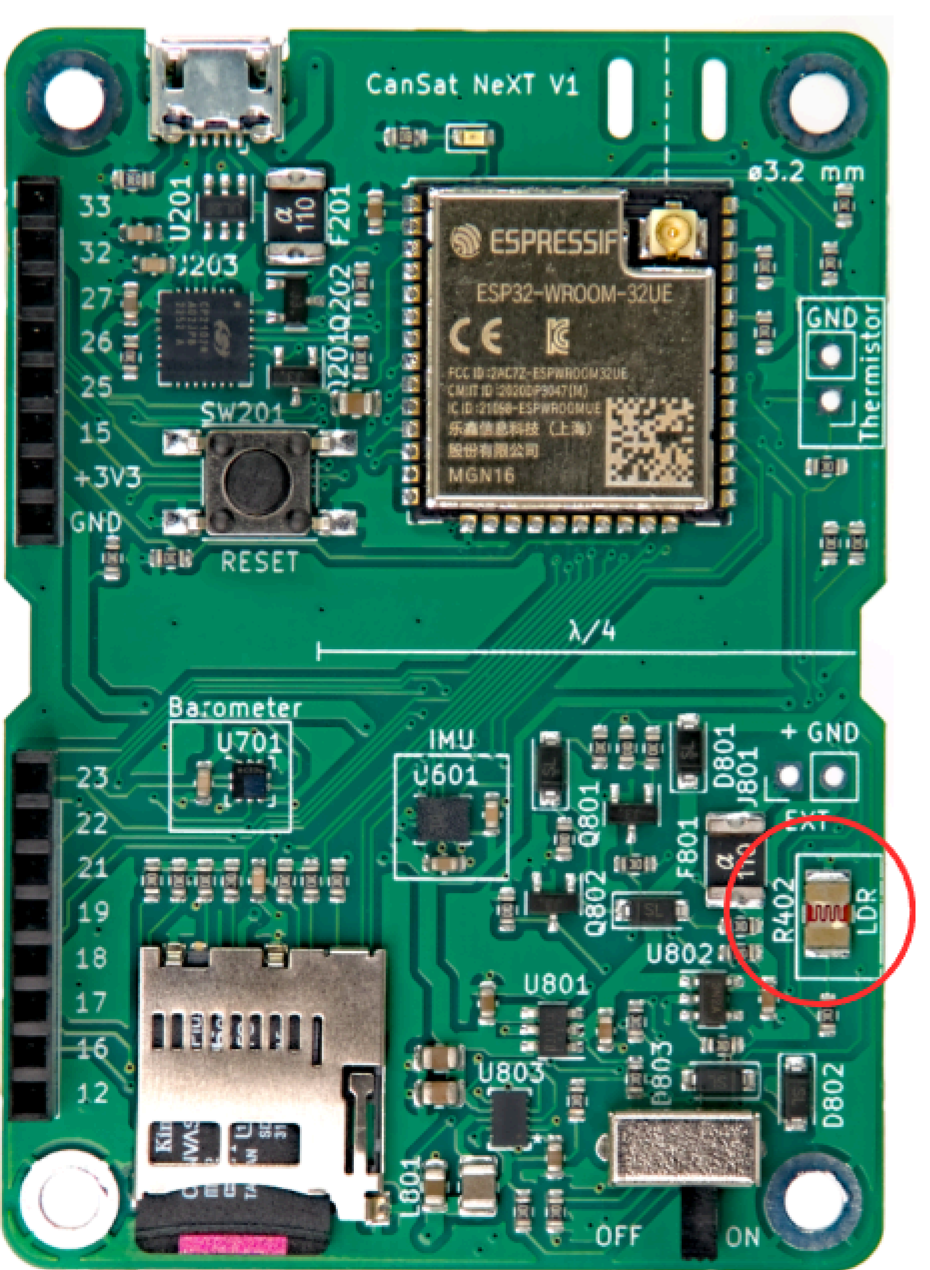
The 12 bit analog-to-digital converter (ADC) in ESP32 is notoriously nonlinear. This doesn't matter for most applications, such as using it to detect temperature changes or changes in LDR resistance, however making absolute measurements of battery voltage or NTC resistance can be a bit tricky. One way around this is careful calibration, which would make for sufficiently accurate data for the temperature for example. However, the CanSat library also provides a calibrated correction function. The function implements a third order polynomial correction for the ADC, correlating the ADC reading with the actual voltage present on the ADC pin. The correction function is

$$V = -1.907217e \times 10^{-11} \times a^3 + 8.368612 \times 10^{-8} \times a^2 + 7.081732e \times 10^{-4} \times a + 0.1572375$$

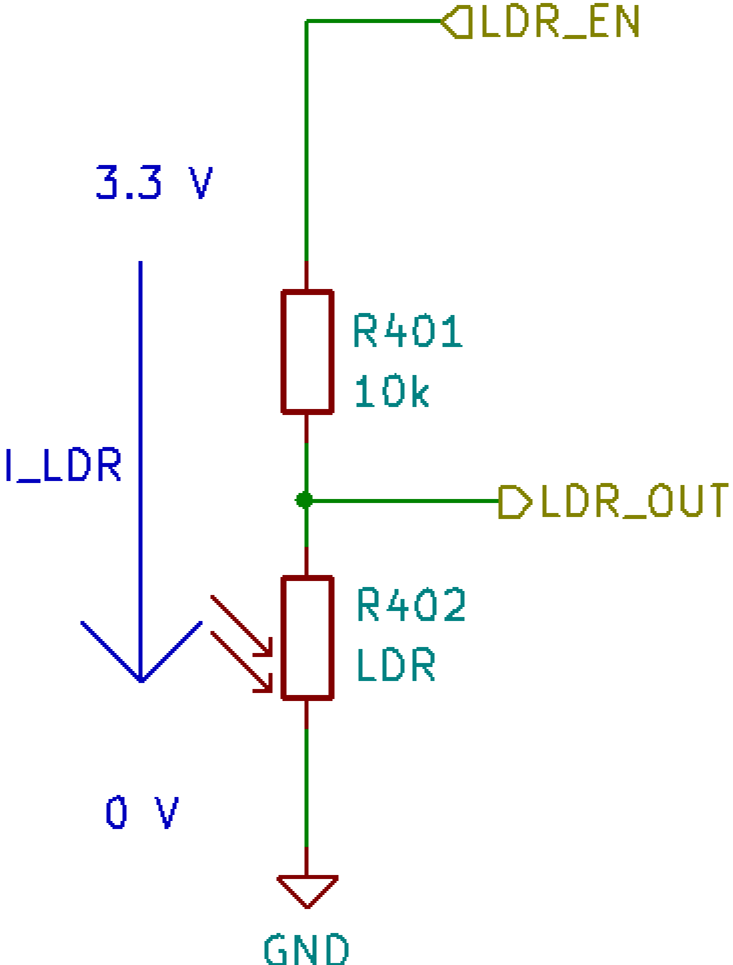
Where V is the measured voltage and a is the 12-bit ADC reading from analogRead(). The function is included in the library, and is called adcToVoltage. Using this formula makes the ADC reading error less than 1% inside a voltage range 0.1 V - 3.2 V.

Light Dependant Resistor

The CanSat NeXT main board incorporates an LDR to the sensor set as well. The LDR is a special kind of resistor, in that the resistance varies with illumination. The exact characteristics may vary, but with the LDR we are currently using, the resistance is 5-10 kΩ at 10 lux, and 300 kΩ in the dark.



CanSat NeXT V1
ø3.2 mm
ESPRESSIF
ESP32-WROOM-32UE
MGN16
U201
F201
SW201
RESET
33
32
27
26
25
15
+3V3
GND
Thermistor
λ/4
Barometer
U701
IMU
U601
+ GND
EXT
D801
J801
Q801
F801
Q802
R402
LDR
U802
U801
U803
D803
D802
L801
23
22
21
19
18
17
16
12
OFF
ON



The way this is used in CanSat NeXT, is that a voltage of 3.3 V is applied to a comparison resistor from the MCU. This causes the voltage at LDR_OUT to be

$$V_{LDR} = V_{EN}\frac{R402}{R401+R402}.$$

And as the R402 resistance changes, the voltage at the LDR_OUT will change as well. This voltage can be read with the ESP32 ADC, and then correlated to the resistance of the LDR. In practice however, usually with LDRs we are interested in the change rather than the absolute value. For example, it usually suffices to detect a large change in the voltage when the device is exposed to light after being deployed from the rocket, for

example. The threshold values are usually set experimentally, rather than calculated analytically. Note that in CanSat NeXT, you need to enable the analog on-board sensors by writing MEAS_EN pin HIGH. This is shown in the example codes.

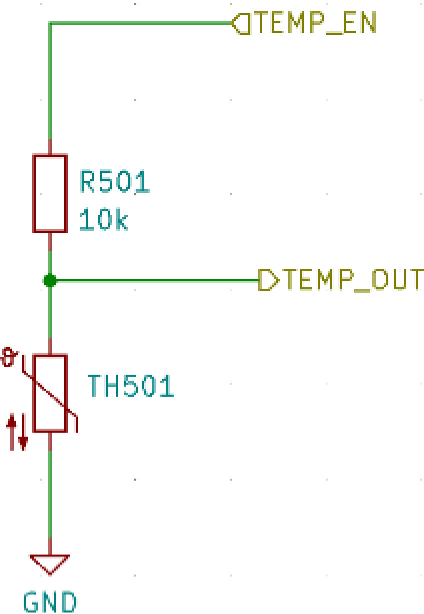
Thermistor

The circuit used to read the external thermistor is very similar to the LDR readout circuit. The exact same logic applies, that when a voltage is applied to the comparison resistor, the voltage at TEMP_OUT changes according to

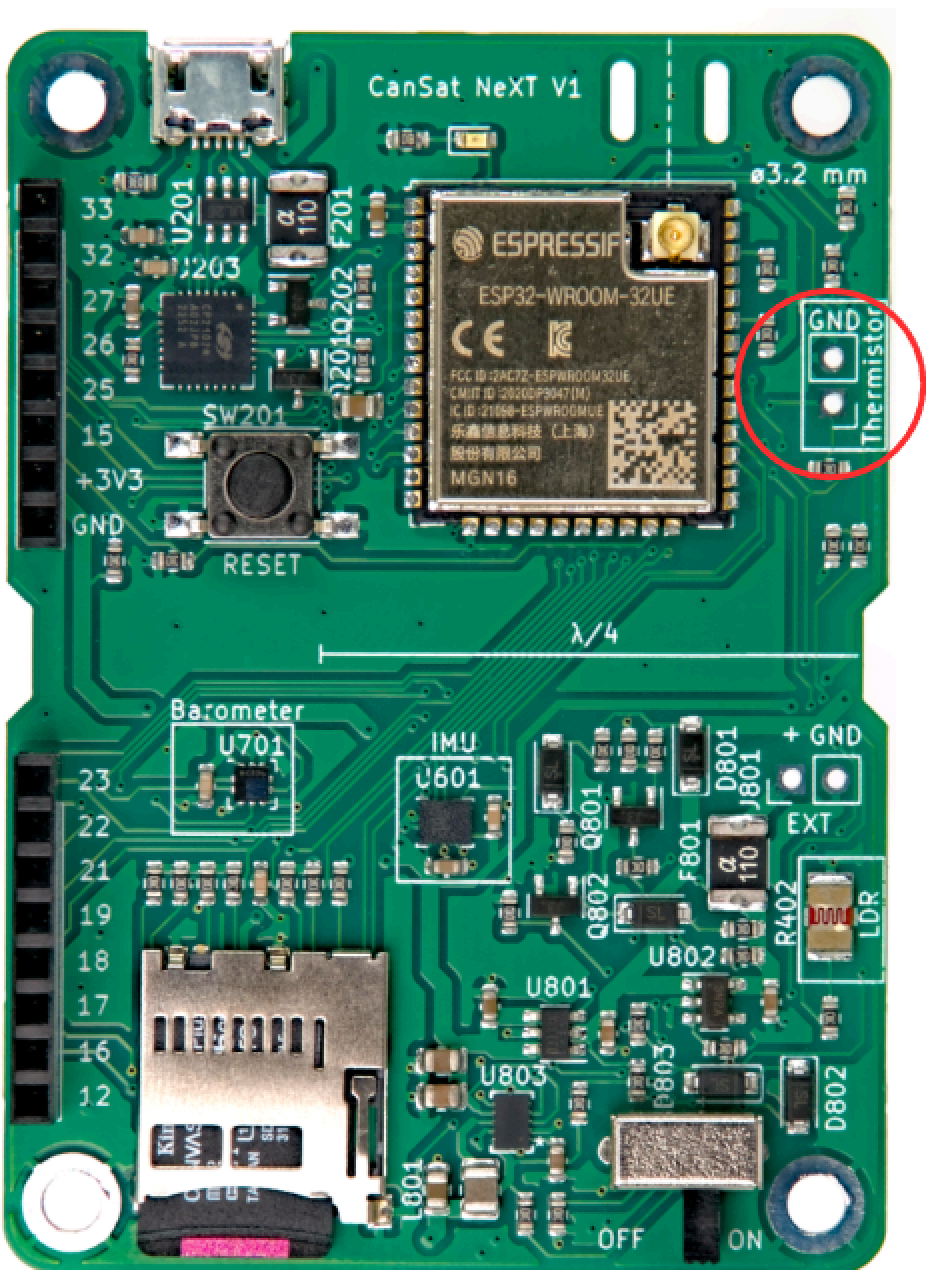
$$V_{TEMP} = V_{EN} \frac{TH501}{TH501 + R501}.$$

In this case however, we are usually interested in the absolute value of the thermistor resistance. Therefore the VoltageConversion is useful, as it linearizes the ADC readings and also calculates the V_temp directly. This way, the user can calculate the resistance of the thermistor in the code. The value should still be correlated with temperature using measurements, although the thermistor datasheet might also include some clues as to how to calculate the temperature from the resistance. Note that if doing everything analytically, you should also take into account the resistance variance of R501. This is done most easily by measuring the resistance with a multimeter, instead of assuming it is 10 000 ohms.

The comparison resistor on the PCB is relatively stable over a temperature range, however it also changes slightly. If very accurate temperature readings are desired, this should be compensated for. The junction temperature measurement from the pressure sensor can be used for this. That being said, it is definitely not required for CanSat competitions. For those interested, the thermal coefficient of the R501 is reported by the manufacturer to be 100 PPM/°C.



While the barometer temperature mostly reflects the temperature of the board itself, the thermistor can be mounted such that it reacts to temperature changes outside the board, even outside the can. You can also add wires to get it even further away. If it will be used, the thermistor can be soldered to the appropriate location on the CanSat NeXT board. The polarization doesn't matter, i.e. it can be mounted either way.



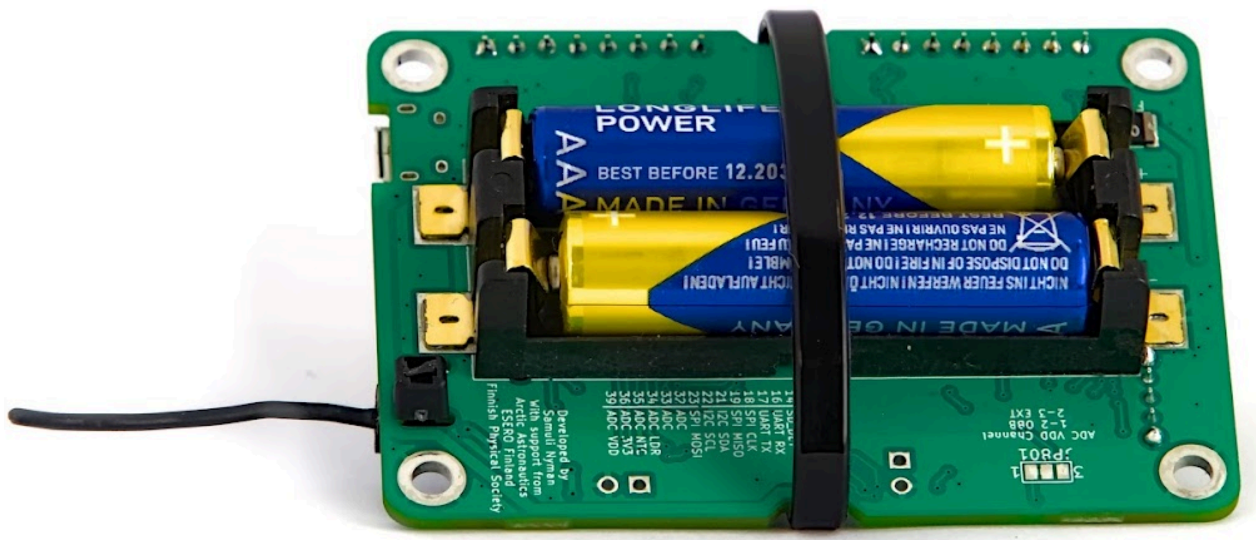
CanSat NeXT V1
ø3.2 mm
ESPRESSIF
ESP32-WROOM-32UE
GND
Thermistor
33
32
27
26
25
15
+3V3
GND
U201
F201
SW201
RESET
λ/4
Barometer
U701
IMU
U601
+ GND
D801
J801
EXT
Q801
F801
Q802
R402
LDR
U802
U801
U803
D803
D802
L801
23
22
21
19
18
17
16
12
OFF
ON

Electrical Power Management

This article explains how to power on the CanSat NeXT board, how to safely connect external devices to the board, and finally how the power system works.

Getting Started

For most users, it is often enough to simply add two AAA-batteries to the on-board battery holder and secure them in place. When the USB is connected, CanSat NeXT automatically switches to use the USB power instead of the batteries, so that the battery life is extended. Remember to switch to fresh batteries before a flight.



CanSat NeXT Power System

There are three ways to power the CanSat NeXT. The default way is to power it with USB, so that when the user is developing the software, the PC powers the device and no external power is required. Second way is to use the on-board batteries (OBB). This is done by inserting two standard 1.5 V AAA batteries into the battery connector on the bottom side of the main board. The USB is still the default way even if batteries are inserted, i.e. the battery capacity is not used when USB is plugged in.

These are the usual options, and should cover most use cases. Additionally, however, there are two “advanced” options for powering CanSat NeXT if needed for a special purpose. First, the board has empty through-hole headers labeled EXT, that can be used for connecting an external battery. The battery voltage can be 3.2-6V. The EXT line is automatically disconnected when USB is not present to extend battery life and to protect the battery. There is a safety feature that the OBB is disabled if a battery is connected, but the OBB should still not be present when external batteries are used.

There is also one last option that gives all responsibility to the user, and that is inputting 3V3 to the device through the extension interface. This is not a safe way to power the device, but advanced users who know what they are doing might find this the easiest way to achieve the desired functionalities.

In summary, there are three safe ways to power CanSat NeXT:

1. Using USB - main method used for development
2. Using on-board batteries - recommended method for flight
3. Using an external battery - For advanced users

Using regular AAA batteries, a battery life of 4 hours was reached in room temperature, and 50 minutes in -40 degrees celsius. During the test, the device read all the sensors and transmitted their data 10 times per second. It should be noted that regular alkaline

batteries are not designed to work in such low temperatures, and they usually start leaking potassium after this kind of torture tests. This is not dangerous, but the alkaline batteries should be always disposed of safely afterwards, especially if they were used in an uncommon environment such as extreme cold, or having been dropped from a rocket. Or both.

When using USB, the current draw from the extension pins should not exceed 300 mA. The OBB are slightly more forgiving, giving at most 800 mA from the extension pins. If more power is required, an external battery should be considered. This is most likely not the case unless you are running motors (small servos are fine) or heaters, for example. Small cameras etc. are still fine.

Extra - how the adaptive multi-source power scheme works

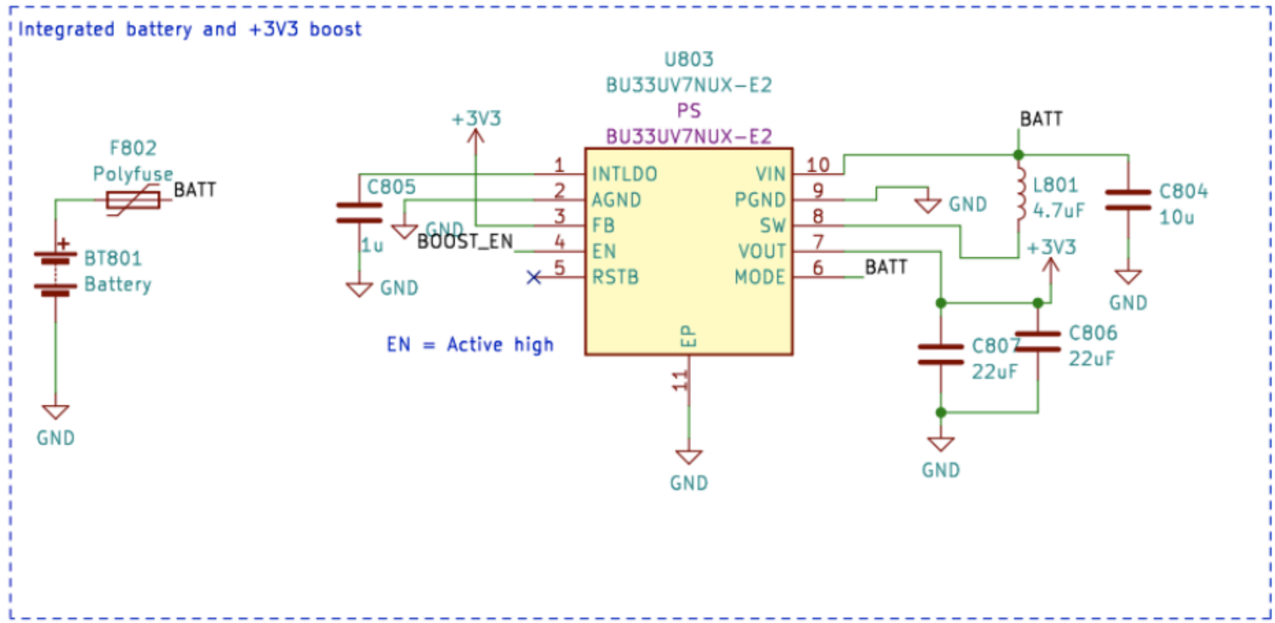
To achieve the desired functionalities safely, we need to consider quite many things in the power system design. First, to safely be able to connect USB, EXT and OBB at the same time, the power system needs to switch on and off the various power sources. This is further complicated by the fact that it can't be done in software, as the user needs to be able to have any software they desire without endangering safe operations. Furthermore, the OBB has a quite different voltage range to the USB and external battery. This necessitates the OBB to use a boost regulator, while the USB and EXT need either a buck regulator or an LDO. For simplicity and reliability, an LDO is used in that line. Finally, one power switch should be able to disconnect all of the power sources.

Below is the schematic for the boost converter. The IC is BU33UV7NUX, a boost converter specifically designed to give +3.3V from two alkaline batteries. It is enabled when the BOOST_EN line is high, or above 0.6 V.

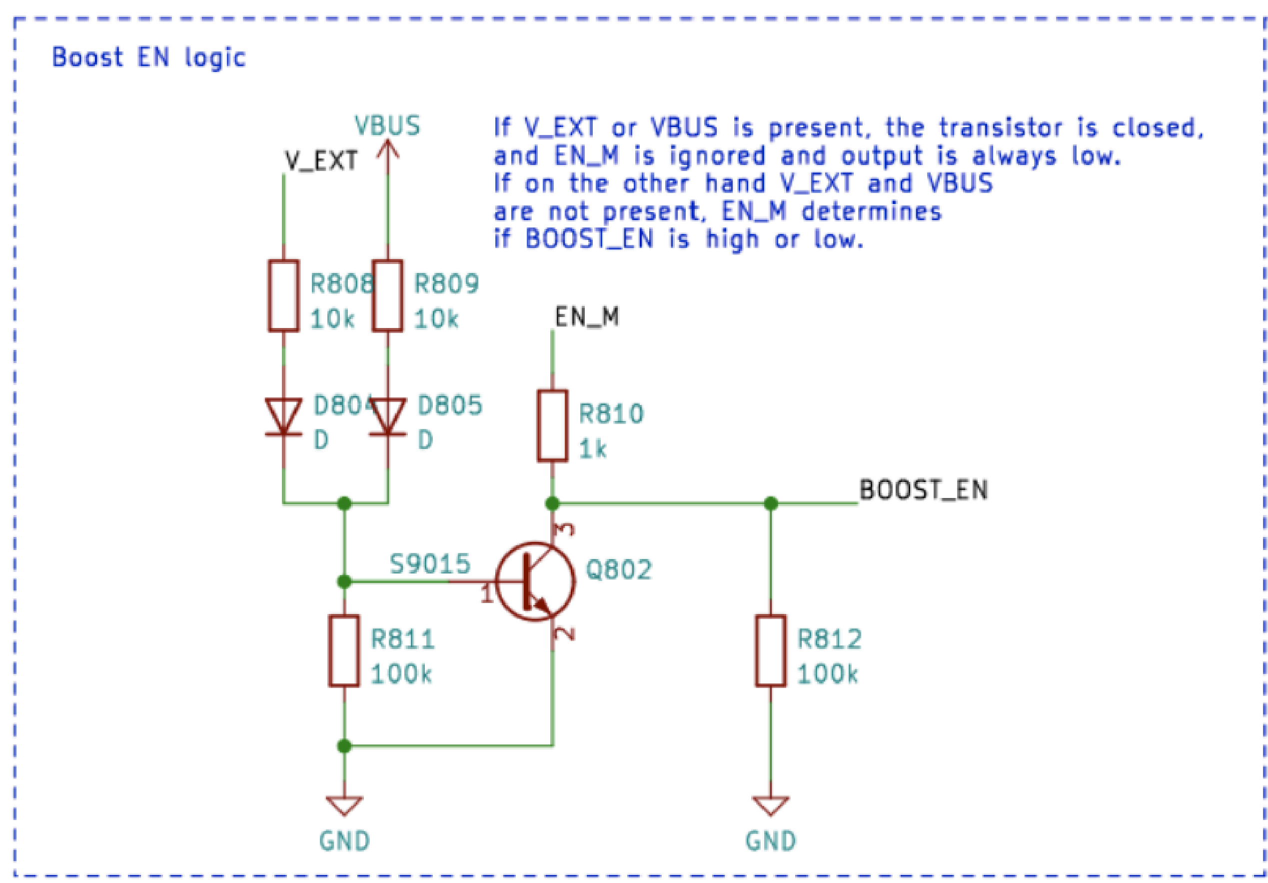
All OBB, USB and EXT lines are protected with a fuse, over-current protection, reverse-voltage and current protection and over temperature protection. Furthermore, the OBB

is protected with under voltage lock out and short circuit protection, as those situations should be avoided with alkaline batteries.

Note in the following section, that external battery voltage is V_EXT, USB voltage is VBUS and OBB voltage is BATT.



The BOOST_EN line is controlled by a switch circuit, which either takes the input from EN_MASTER (EN_M) line, or ignores that if V_EXT or VBUS is present. This is made to ensure that the boost is always off when VBUS and V_EXT is present, and it is only enabled if both VBUS and V_EXT are at 0V and the EN_M is high.

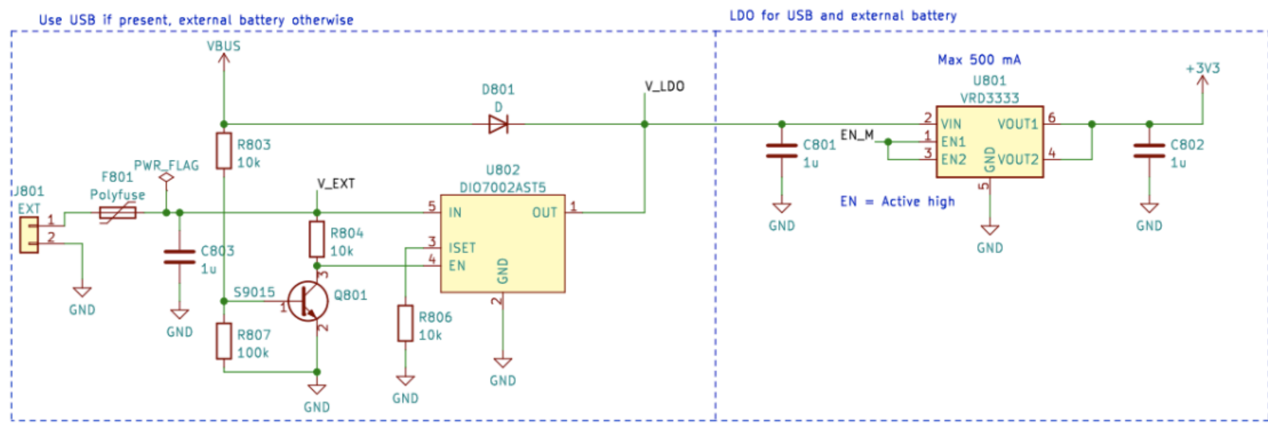


Or as a truth table:

V_EXT	VBUS	EN_M	BOOST_EN
1	1	1	0
1	1	0	0
0	0	0	0
0	0	1	1

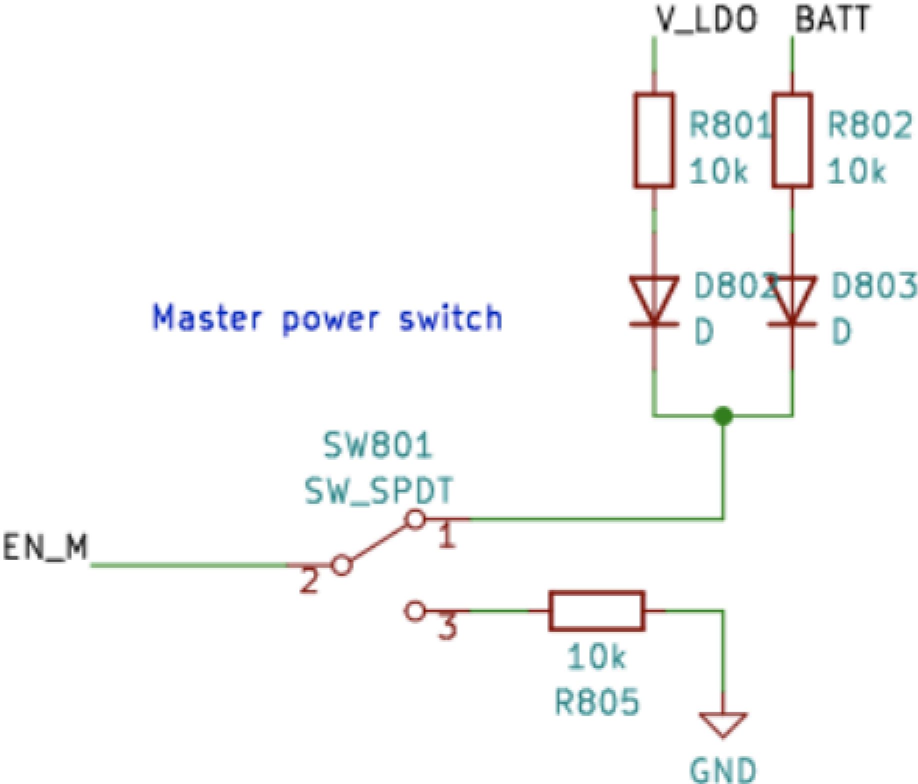
So BOOST_EN = EN_M ∧ !(V_EXT ∨ V_BUS).

Next, we need to disconnect V_EXT if VBUS is present to prevent undesired discharge or accidental charging. This is done using a power switch IC with help of a transistor circuit which takes the enable-line of the power switch down if VBUS is present. This disconnects the battery. The USB line is always used when present, so it is routed to the LDO with a simple schottky diode.



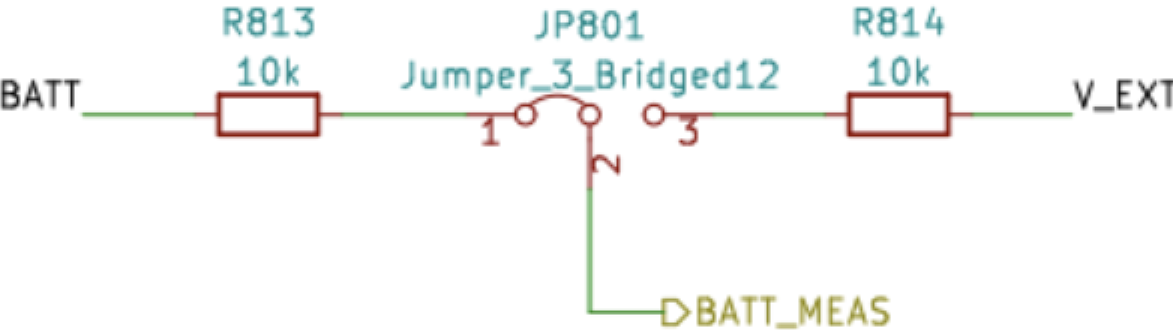
Overall, this circuit leads to a functionality where USB power is used if present, and V_EXT used when USB is not present. Finally, the EN_M is used to enable or disable the LDO.

The EN_M is controlled by the user through a power switch. The switch connects EN_M to either USB or EXT, or the battery voltage when only OBB is used. When the switch is turned off, it connects EN_M to ground, turning off both the LDO and the boost regulator.



So in practice, the power switch turns the device on/off, USB is used if present, and V_EXT is preferred over OBB. Finally, there is one more detail to consider. What voltage should ESP32 measure as the battery voltage?

This was solved in a simple way. The voltage connected to the ESP32 ADC is always the OBB, but the user can select V_EXT instead by cutting the jumper with a scalpel and soldering the jumper JP801 to short 2-3 instead. This selects V_EXT to the BATT_MEAS instead.



The jumper can be found from the bottom side of the CanSat NeXT main board. The jumper is quite easy to solder, so don't be afraid to cut the 1-2 line if you are using an external battery. It can always be resoldered to again use 1-2 instead.

Communication and Antennas

This article introduces the key concepts needed for wireless data transmission with CanSat NeXT. First, the communication system is discussed on a general level, next some different options are presented for antenna selection when using CanSat NeXT. Finally, the last part of article presents a simple tutorial for building a quarter-wave monopole antenna from the parts included in the kit.

Getting Started

CanSat NeXT is almost ready to start wireless communication straight out of the box. Only thing that is needed is the proper software, and an antenna for both the transmitter and the receiver. For the first one, refer to the software materials on this page. For the latter one, this page includes instructions on how to select an external antenna, and to how construct a simple monopole antenna from the materials included with the CanSat NeXT.

While the board is quite resilient to such things thanks to software checks, you should never attempt to transmit anything from a radio without an antenna. Although unlikely due to the low powers involved with this system, the reflected radiowave can cause real harm to the electronics.

CanSat NeXT Communication System

CanSat NeXT handles the wireless data transfer a bit differently to the older CanSat kits. Instead of a separate radio module, CanSat NeXT uses the MCU’s integrated WiFi-radio for the communication. The WiFi-radio is normally used to transfer data between an

ESP32 and the internet, enable the use of ESP32 as a simple server, or even connect ESP32 to a bluetooth device, but with certain clever TCP-IP configuration tricks, we can enable direct peer-to-peer communication between ESP32 devices. The system is called ESP-NOW, and it is developed and maintained by EspressIf, who are the developers of ESP32 hardware. Furthermore, there are special low-rate communication schemes, which by increasing the energy-per-bit of the transmission, significantly increase the possible range of the wifi-radio over the usual few tens of meters.

The data rate of ESP-NOW is significantly faster than what would be possible with the old radio. Even with simply decreasing the time between packets in the example code, CanSat NeXT is able to transmit ~20 full packets to the GS in a second. Theoretically the data rate can be up to 250 kbit/s in the long range mode, but this can be hard to achieve in the software. That being said, transmission of for example full pictures from a camera during the flight should be entirely feasible with correct software.

Even with simple quarter-wavelength monopole antennas (a 31 mm piece of wire) at both ends, CanSat NeXT was able to send data to the ground station from 1.3 km away, at which point the line of sight was lost. When testing with a drone, the range was limited to roughly 1 km. It is possible that the drone interfered with the radio enough to somewhat limit the range. However, with a better antenna, the range could be increased even more. A small yagi antenna would have theoretically increased the operational range 10-fold.

There are a couple practical details that differ from the older radio communication system. First, the “pairing” of satellites to ground station receivers happens with Media Access Control (MAC) addresses, which are set in the code. The WiFi system is clever enough to handle the timing, collision and frequency issues behind the scenes. The user simply needs to ensure that the GS is listening to the MAC address the satellite is transmitting with. Secondly, the frequency of the radio is different. The WiFi radio operates at 2.4 GHz band (center frequency is 2.445 GHz), which means that both the propagation characteristics and requirements for antenna design are different than before. The signal is somewhat more sensitive to rain, and line-of-sight issues, and might

not be able to transmit in some cases where the old system would have worked.

The wavelength of the radio signal is also different. Since

$$\lambda = \frac{c}{f} \approx \frac{3*10^8 \text{ m/s}}{2.445*10^9 \text{Hz}} = 0.12261 \text{ m},$$

a quarter wavelength monopole antenna should have a length of 0.03065 m or 30.65 mm. This length is also marked on the CanSat NeXT PCB to make cutting of the cable a bit easier. The antenna should be cut precisely, but within ~0.5 mm is still fine.

A quarter wavelength antenna has sufficient RF performance for the CanSat competitions. That being said, it might be of interest to some users to get even better range. One possible place of improvement is in the length of the monopole antenna. In practice the quarter-wavelength resonance might not be exactly at the right frequency, since other parameters such as environment, surrounding metal elements or the portion of the wire still covered with grounded metal might affect the resonance a bit. The antenna could be tuned with the use of a vector network analyzer (VNA). I think I should do this at some point, and correct the materials accordingly.

A more robust solution would be to use a different style of antenna. At 2.4 GHz, there are loads of fun antenna ideas on the internet. These include a helix antenna, yagi antenna, pringles antenna, and many others. Many of these, if well constructed, will outperform the simple monopole easily. Even just a dipole would be an improvement over a simple wire.

The connector used on most ESP32 modules is a Hirose U.FL connector. This is a good quality miniature RF connector, which provides good RF performance for weak signals. One problem with this connector however is that the cable is quite thin making it a bit impractical in some cases. It also leads to larger-than-desired RF losses if the cable is long, as it might be when using an external antenna. In these cases, a U.FL to SMA adapter cable could be used. I’ll look to see if we could provide these in our webshop. This would enable teams to use a more familiar SMA connector. That being said, it is completely possible to build good antennas with just using U.FL.

Unlike SMA however, U.FL relies mechanically on snap-on retaining features to hold the connector in place. This is usually sufficient, however for extra safety it is a good idea to add a zip tie for extra security. The CanSat NeXT PCB has slots next to the antenna connector to accommodate a small zip tie. Ideally, a 3d-printed or otherwise constructed support sleeve would be added for the cable before the zip tie. A file for the 3d-printed support is available from the GitHub page.

Antenna Options

An antenna is essentially a device that transforms unguided electromagnetic waves into guided ones, and vice versa. Due to the simple nature of the device, there are a multitude of options from which to select the antenna for your device. From a practical point of view, the antenna selection has a lot of freedom, and quite many things to consider. You need to consider at least

1. Operating frequency of the antenna (should include 2.45 GHz)
2. Bandwidth of the antenna (At very least 35 MHz)
3. Impedance of the antenna (50 ohms)
4. Connector (U.FL or you can use adapters)
5. Physical size (Does it fit to the can)
6. Cost
7. Manufacturing methods, if you are making the antenna yourself.
8. Polarization of the antenna.

Antenna selection can seem overwhelming, and it often is, however in this case it is made much easier by the fact that we are in fact using a Wi-Fi-radio - we can actually use almost any 2.4 GHz Wi-Fi antenna with the system. Most of them however are too large, and also they tend to use connectors called RP-SMA, rather than U.FL. However, with a suitable adapter they can be good choices to use with the groundstation. There are even directive antennas available, meaning that you can get extra gain to improve the radio

link.

Wi-Fi antennas are a solid choice, however they have one significant drawback - polarization. They are almost always linearly polarized, which means that the signal strength varies significantly depending on the orientation of the transmitter and the receiver. In worst cases, the antennas being perpendicular to each other might even see the signal fade out completely. Therefore, an alternative option is to use drone antennas, which tend to be circularly polarized. In practice this means that we have some constant polarization losses, but they are less dramatic. An alternative clever solution to get around the polarization problem is to use two receivers, with antennas mounted perpendicular to each other. This way at least one of them will always have a suitable orientation for receiving the signal.

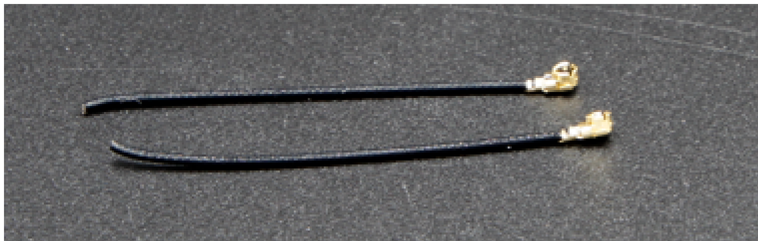
Of course, a true maker will always want to make their own antenna. Some interesting constructions that are suitable for DIY-manufacturing include a helix-antenna, "pringles" antenna, yagi, dipole, or a monopole antenna. There are a lot of instructions online for building most of these. The last part of this article shows how to make your own monopole antenna, suitable for CanSat competitions, from the materials shipped with CanSat NeXT.

Building a quarter-wave monopole antenna

This section of article describes how to build a resonably effective quarter-wave monopole antenna from the materials included in the kit. The antenna is called that since it only has one pole (compare to a dipole), and its length is quarter of the wavelength that we are transmitting.

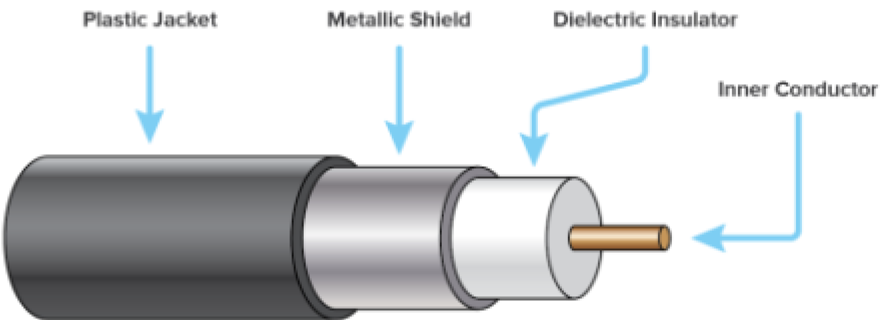
In addition to the coaxial cable and piece of heat shrink tubing, you'll need some type of wire strippers and wire cutters. Almost any type will work. Additionally you will need a heat source for the heat shrink, such as a hot air gun, soldering iron or even a lighter.

First, begin by cutting the cable roughly in half.



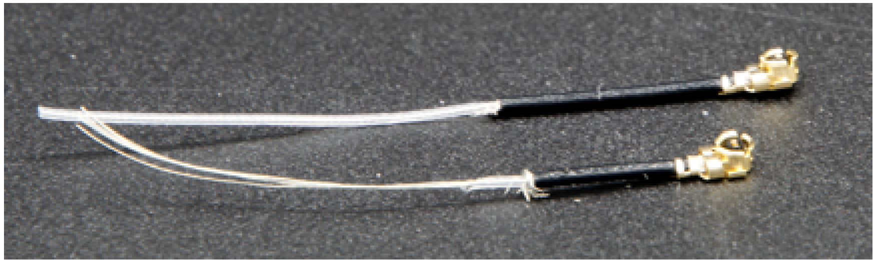
Next, we will build the actual antenna. This part should be done as precisely as you can. Within 0.2 mm or so will work fine, but try to get it as close to the correct length as possible, as that will help with the performance.

A coaxial cable consists of four parts - a center conductor, dielectric, shield, and an outer jacket. Usually, these cables are used to transmit radio frequency signals between devices, so that the currents on the center conductor are balanced by those in the shield. However, by removing the shield conductor, the currents on the inner conductor will create an antenna. The length of this exposed area will determine the wavelenght or operating frequency of the antenna, and we now want it to match our operating frequency of 2.445 GHz, so we need remove the shield from length of 30.65 mm.

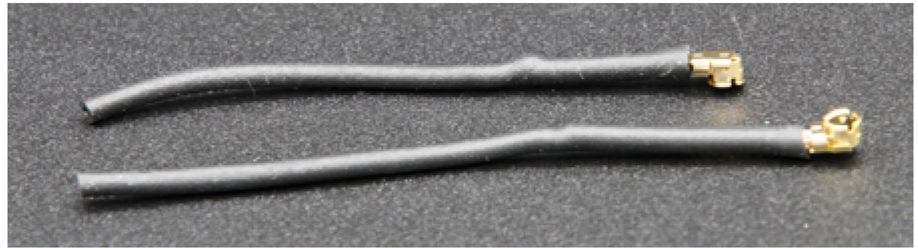


Carefully strip the outer jacket from the cable. Ideally, try to remove only the jacket and the shield from the desired length. However, cutting the insulator is not a catastrophe. It is usually easier to remove the outer jacket in parts, rather than all at once. Furthermore, it might be easier to first remove too much, and then cut the inner conductor to the right length, rather than try to get it exactly right on the first try.

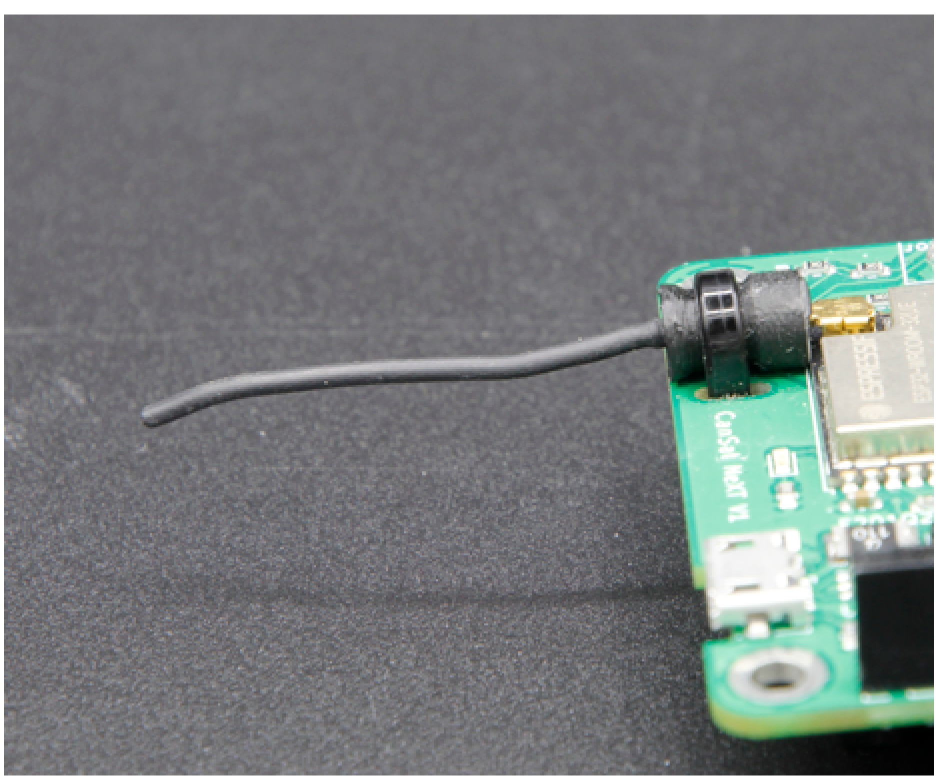
The image below shows the stripped cables. Try to make it like the upper one, but the lower one will work as well - it just might be more sensitive to moisture. If there are dangling pieces of the shield left, carefully cut them off. Make sure that there are no possibility that the inner conductor and the shield are touching each other - even a single strand would render the antenna unusable.



The antenna is now totally functional at this point, however it may be sensitive to moisture. Therefore, we want to now add a new jacket to this, which is what the heat shrink tubing is for. Cut two pieces, slightly longer than the antenna you have made, and place it over the antenna and use a heat source to shrink it in place. Be careful not to burn the heat shrink tubing, especially if using something else than a hot air gun.



After this, the antennas are ready. On the groundstation side, the antenna is probably fine like this. On the other hand, while the connector is fairly secure, it is a good idea to support the connector somehow on the CanSat side. A very robust way is to use a 3d-printed support and some ziptie, however many other methods will work as well. Remember to also consider how the antenna will be placed inside the can. Ideally, it should be in a location where the transmission is not blocked by any metal parts.



Finally, here is a step-file of the support shown in the image. You can import this into most CAD software, and modify it, or print it with a 3d-printer.

Download step-file

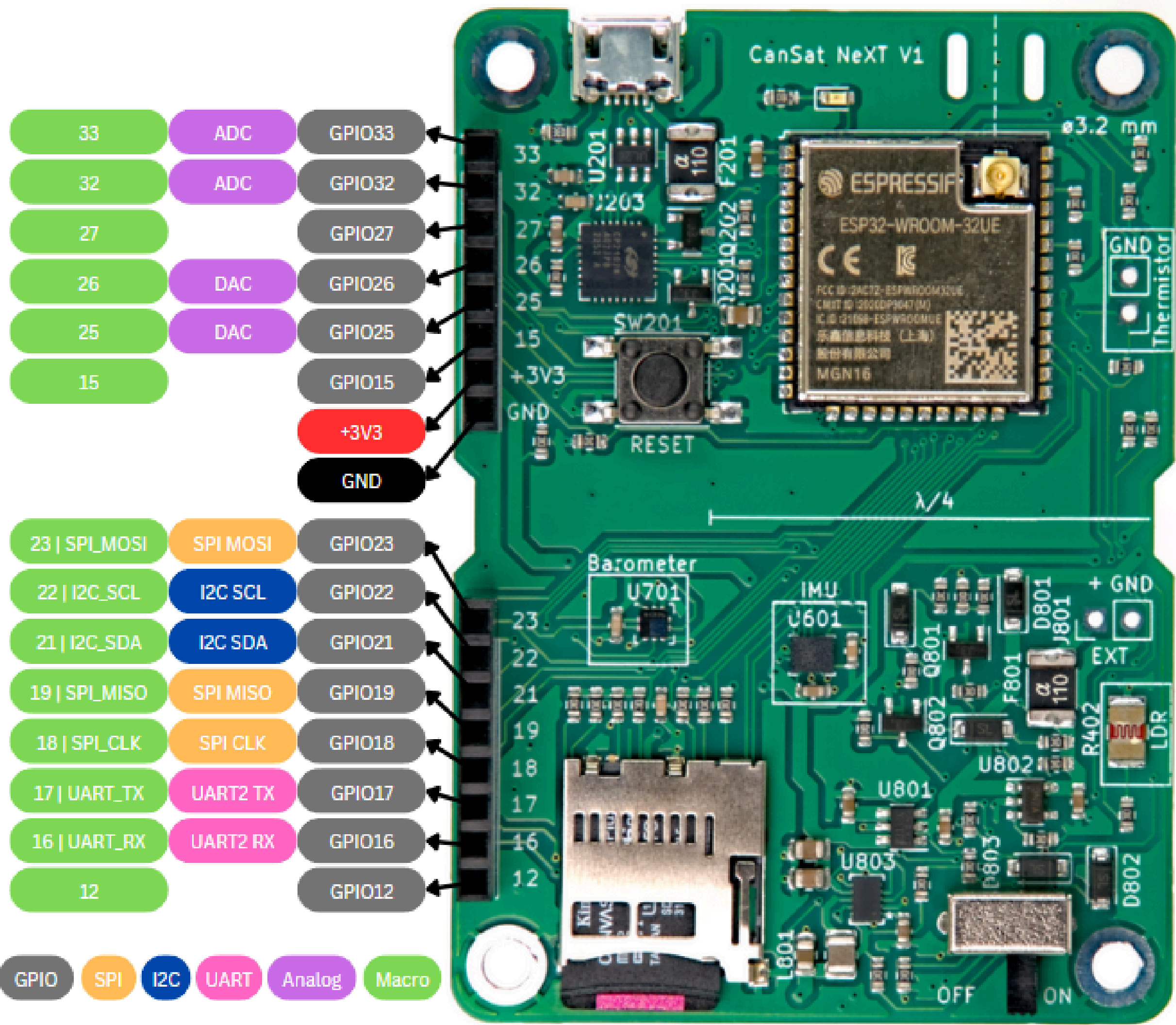
Pinouts

This article shows the pin names used by the processor in CanSat NeXT, as well as shows what pins you can use to extend your project.

Pinout

The picture below shows the pins for using the extension header for adding external electronics to the board.

CanSat NeXT V1 Pinout



Here is the full list of pins used by CanSat NeXT board. The internal use refers to the pin being used for the on-board resources, and extension refers to the pins having been routed to the extension interface. Some pins, those for I2C and SPI, are used both internally and externally. The library name refers to a macro name, which can be used instead of the pin number when CanSatNeXT library has been included.

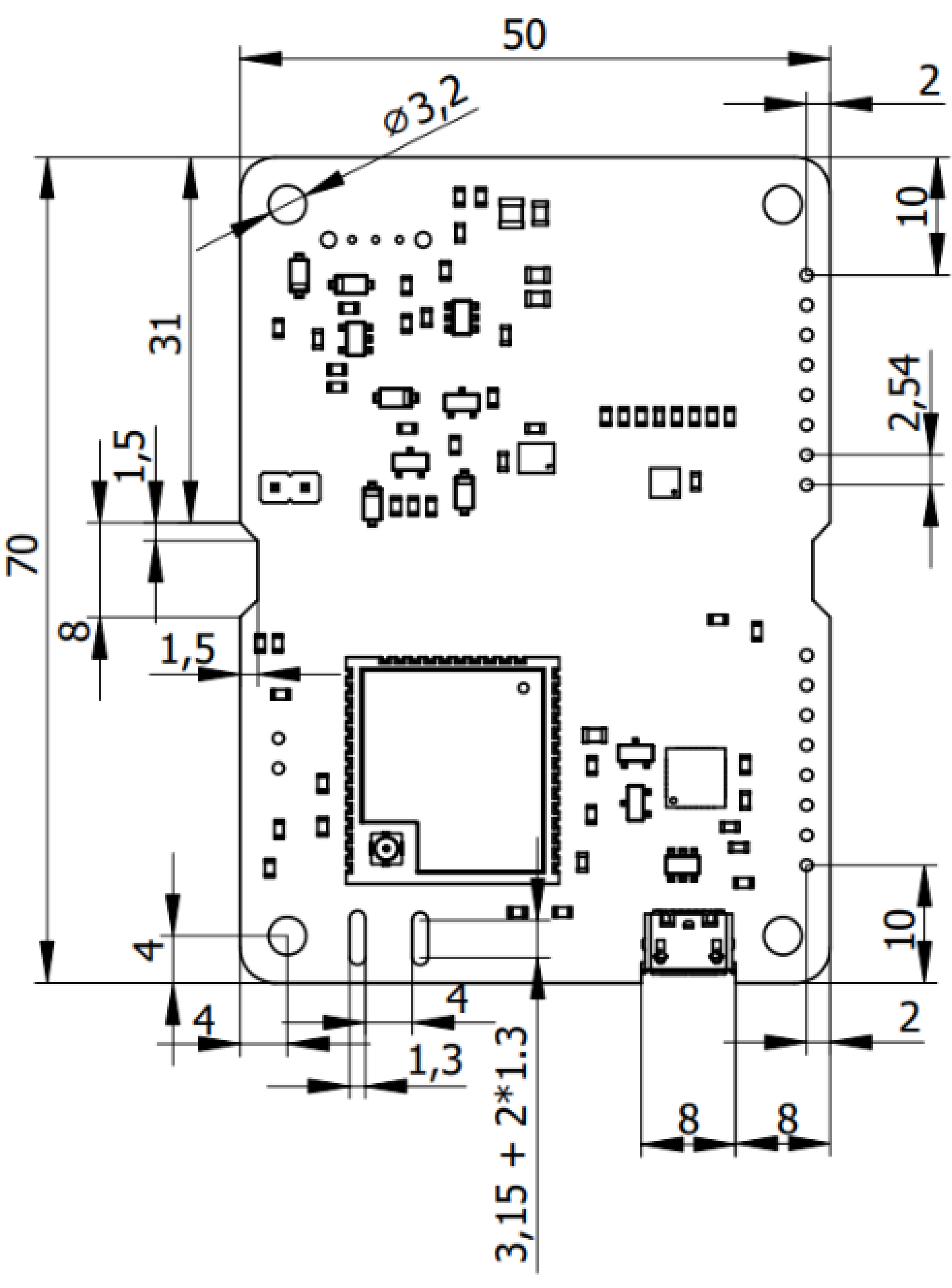
Pin Number	Library name	Note	Internal/ External
0	BOOT		Used internally
1	USB_UART_TX	Used for USB	Used internally
3	USB_UART_RX	Used for USB	Used internally
4	SD_CS	SD card chip select	Used internally
5	LED	Can be used to blink on-board LED	Used internally
12	GPIO12		Extension interface
13	MEAS_EN	Drive high to enable LDR and thermistor	Used internally
14	GPIO14	Can be used to read if SD-card is in place	Used internally
15	GPIO15		Extension interface
16	GPIO16	UART2 RX	Extension interface

Pin Number	Library name	Note	Internal/ External
17	GPIO17	UART2 TX	Extension interface
18	SPI_CLK	Used by the SD-card, also available externally	Both
19	SPI_MISO	Used by the SD-card, also available externally	Both
21	I2C_SDA	Used by the on-board sensors, also available externally	Both
22	I2C_SCL	Used by the on-board sensors, also available externally	Both
23	SPI_MOSI	Used by the SD-card, also available externally	Both
25	GPIO25		Extension interface
26	GPIO26		Extension interface
27	GPIO27		Extension interface
32	GPIO32	ADC	Extension interface

Pin Number	Library name	Note	Internal/ External
33	GPIO33	ADC	Extension interface
34	LDR	ADC for the on-board LDR	Used internally
35	NTC	ADC for the thermistor	Used internally
36	VDD	ADC used to monitor supply voltage	Used internally
39	BATT	ADC used to monitor battery voltage	Used internally

Mechanical Design

PCB Dimensions



The CanSat NeXT main board is built on a 70 x 50 x 1.6 mm PCB, with electronics on the top side and battery on the bottom side. The PCB has mounting points on each corner, 4 mm from the sides. The mounting points have a diameter of 3.2 mm with a grounded pad area of 6.4 mm, and they are intended for M3 screws or standoffs. The pad area is also large enough to fit a M3 nut. Additionally, the board has two trapezoidal 8 x 1.5 mm cutouts on the sides and a component-free area on the top side in the center, so that a zip tie or other extra support can be added for the batteries for flight operations. Similarly, two 8 x 1.3 mm slots can be found next to the MCU antenna connector so that the antenna can be secured to the board with a small zip tie or piece of string. The USB connector is slightly intruded to the board to prevent any extrusions. A small cutout is added to accommodate certain USB cables despite the intrusion. The extension headers are standard 0.1 inch (2.54 mm) female headers, and they are placed so that the center of the mounting hole is 2 mm from the long edge of the board. The header closest to the short edge is 10 mm away from it. The thickness of the PCB is 1.6 mm, and the height of the batteries from the board is roughly 13.5 mm. The headers are roughly 7.2 mm tall. This makes the height of the enclosing volume roughly 22.3 mm. Furthermore, if standoffs are used to stack compatible boards together, the standoffs, spacers or other mechanical mounting system should separate the boards at least 7.5 mm. When using standard pin headers, the recommended board separation is 12 mm.

Below, you can download a .step file of the perf-board, which can be used to add the PCB into a CAD-design for reference, or even as a starting point for a modified board.

Download step-file

Designing a Custom PCB

If you want to take your electronics design to the next level, you should consider making a custom PCB for the electronics. KiCAD is a great, free software that can be used to design PCBs, and getting them manufactured is surprisingly affordable.

Here are resources on getting started with KiCAD:
https://docs.kicad.org/#_getting_started

Here is a KiCAD template for starting your own CanSat compatible circuit board:
Download KiCAD template