ARCTIC ASTRO NAUTICS

CanSat NeXT Technical Description

Arctic Astronautics Ltd.

Version history

Date	Notes	Authors
October 2023	Initial version	Samuli Nyman, Tessa Nikander

1 Introduction

This document contains a detailed technical description of CanSat NeXT V1 system hardware and firmware. The document is oriented as a reference for advanced users, project contributors and others interested in the details of the CanSat NeXT system.

This document as well as CanSat NeXT hardware and the software library are under active development. It is possible that this document contains information that does not reflect the most current status of the technical systems. If you find errors or just something you want to ask about, please leave a comment on the document or send an email to samuli@kitsat.fi.

2 Overview

CanSat NeXT is designed to provide a better starting point for design of a CanSat satellite than the previously available DIY kits. The design goals were iterated together with ESERO Finland, and were as follows:

1. No soldering - The solution should be ready to use with minimal assembly, in order to get the students right to science and technology, instead of spending valuable time assembling and reassembling a kit.
2. Comprehensive - The solution should include everything required for the main mission of CanSat competitions - i.e. measurement of temperature and pressure on board, saving and transmitting the measurements. Additionally, the radio range and power system should be designed such that the solution can be used in competitions in all environmental conditions.
3. Low cost and good availability - The solution should have lower cost than the previous kit, and it should be designed such that parts can be swapped between batches if necessary.
4. Extendability - The solution should still provide ways to extend and adjust the electronics to fit all student missions, so as not to limit the creativity and ambitiousness of the projects.

Some additional, more technical requirements were found as well as the project matured. The additional requirements are:

1. Compatibility with Arduino system - The student teams, CanSat trainers and the teachers are already familiar with Arduino. So rather than learn a new system, they should be able to use the environment they are already familiar with.
2. Sensor features- The device should have at least as comprehensive a set of sensors, so as not to be a downgrade from the older devices.
3. Mechanical robustness - The device should be built strong enough to withstand the rocket launches, as well as provide good mounting points to the satellite frame.

To check all the boxes, we decided to make the new kit essentially a custom made development board, designed specifically to have the necessary features for CanSat competitions, while maintaining extendability and compatibility with the Arduino IDE.

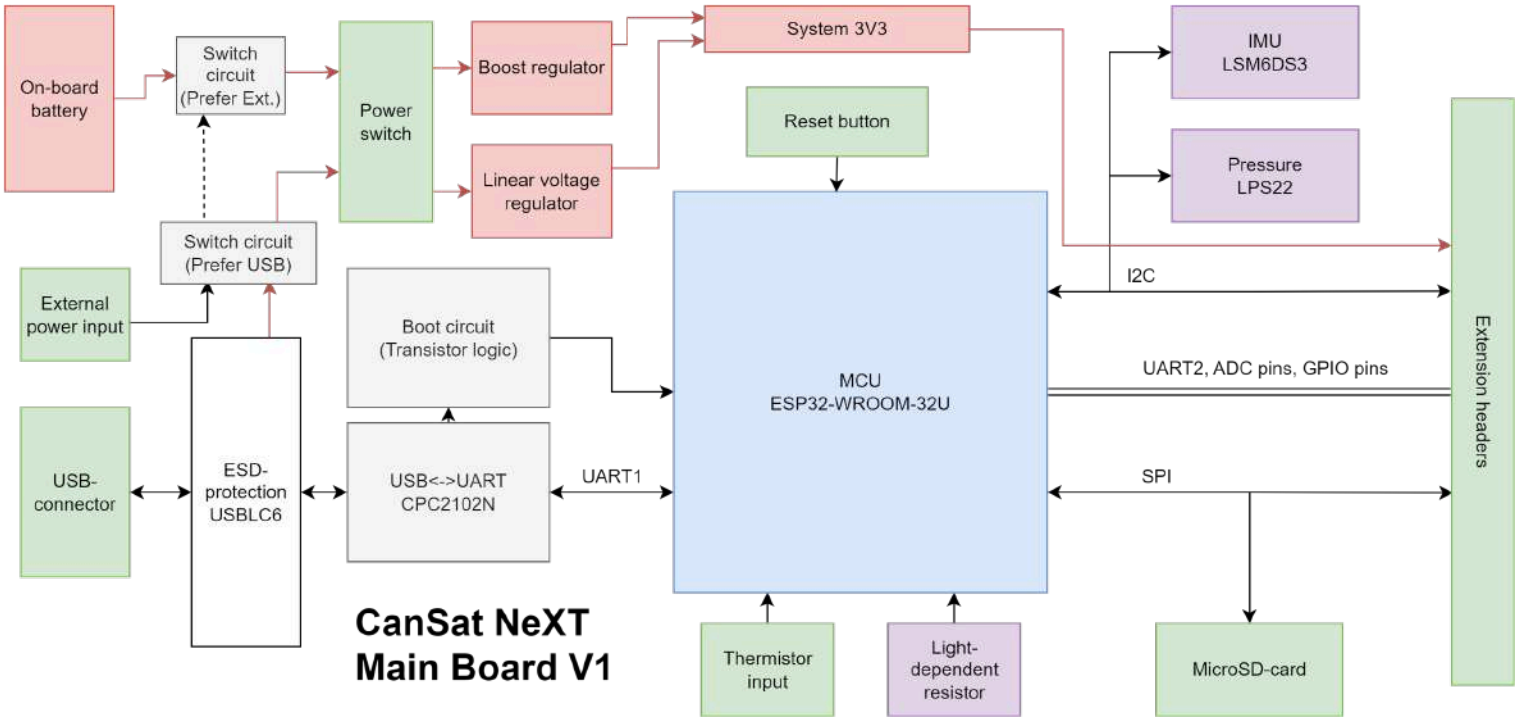
The processor on the CanSat NeXT board is an ESP32-WROOM-32U. The processor is very fast and capable, with large memory compared to the ATmega MCU used before. Furthermore, it is well integrated with the Arduino system, and a mature board setup can be found from the Arduino IDE board manager. The main benefit of using the ESP32 is the WiFi radio integrated with the MCU. With some tricks, we can use this radio as the communication channel for the CanSat. The receiver will be another ESP32. The easiest way to get another ESP32 to act as the ground receiver is to include a standard ESP32 development board with the system.

The board has three sensors on-board. These are inertial measurement unit (IMU) LSM6DS3 and pressure sensor LPS22HB, both manufactured by STMicroelectronics, and a light-dependent resistor (LDR), which can be used as a rough luminosity measurement to for example detect the deployment from the rocket.

For on-board memory, the device has a connector for standard micro SD-card. A <32 GB card is recommended for best performance with the SPI interface.

The power system on the board consists of two main parts - USB and on-board batteries (OBB). The OBB are standard 1.5V AAA-batteries available in any grocery store, which give the device roughly 4 hours of runtime. The default power system however is the USB, so that when USB is connected, the OBB is not used. This means that development can be done with batteries in the device, and they are only used when power is on and USB is not connected. Additionally, the board has through-hole pads that can be used to connect an external battery, if more power is required. Please refer to the chapter on the power system for more information.

Most of the MCU pins are exposed to the extension interface, which means that the students are able to connect any other sensors or devices to the board in addition to using the on-board sensors. The extension interface exposes I2C and SPI lines shared by the on-board devices, as well as an unused UART line, two ADC lines and several GPIO lines.



CanSat NeXT V1 hardware block diagram

CanSat NeXT V0.9 board ready to fly

3 Microcontroller and flashing

The CanSat NeXT main board and the ground station unit both use ESP32 MCUs. From the user's point of view, it is actually not that important as to which exact module they have, as the

Arduino system handles all MCU specific tasks in the background, and the user code is actually an application code running in a contained system. Currently the MCU on the main board is ESP32-WROOM-32U, but this will be replaced with the less power hungry ESP32-WROOM-32UE as soon as they are available on the PCB/A partner.

One important thing however to take into account when selecting the MCU module, is to make sure that it does not have a PCB antenna, as the performance of the modules with the PCB antenna is much worse than those with an antenna connector. There are also modules with both PCB antenna and an antenna connector, but those actually have the worst performance of all, as they handle the signal division in a completely incorrect way.

3.1 Microcontroller pins

Unlike with most MCU's, ESP32 boards have a significant drawback regarding which pins can be used. Many pins of the MCU are reserved for internal functions, but are still exposed to enable certain niche use cases. For example, the second ADC channel along with all the pins it uses is reserved for the WiFi radio system. This significantly reduces the number of pins CanSat NeXT can expose to the user. Anyway, the safety and usability of the pins is already handled internally on the CanSat NeXT, so the user does not have to take this into account.

All pins used by CanSat NeXT from the MCU are presented below. Only some of the pins are also exposed to the extension interface, while the rest are reserved for internal functions. In addition to this table, the pins are also listed in the cansatnext_pins.h file in the library.

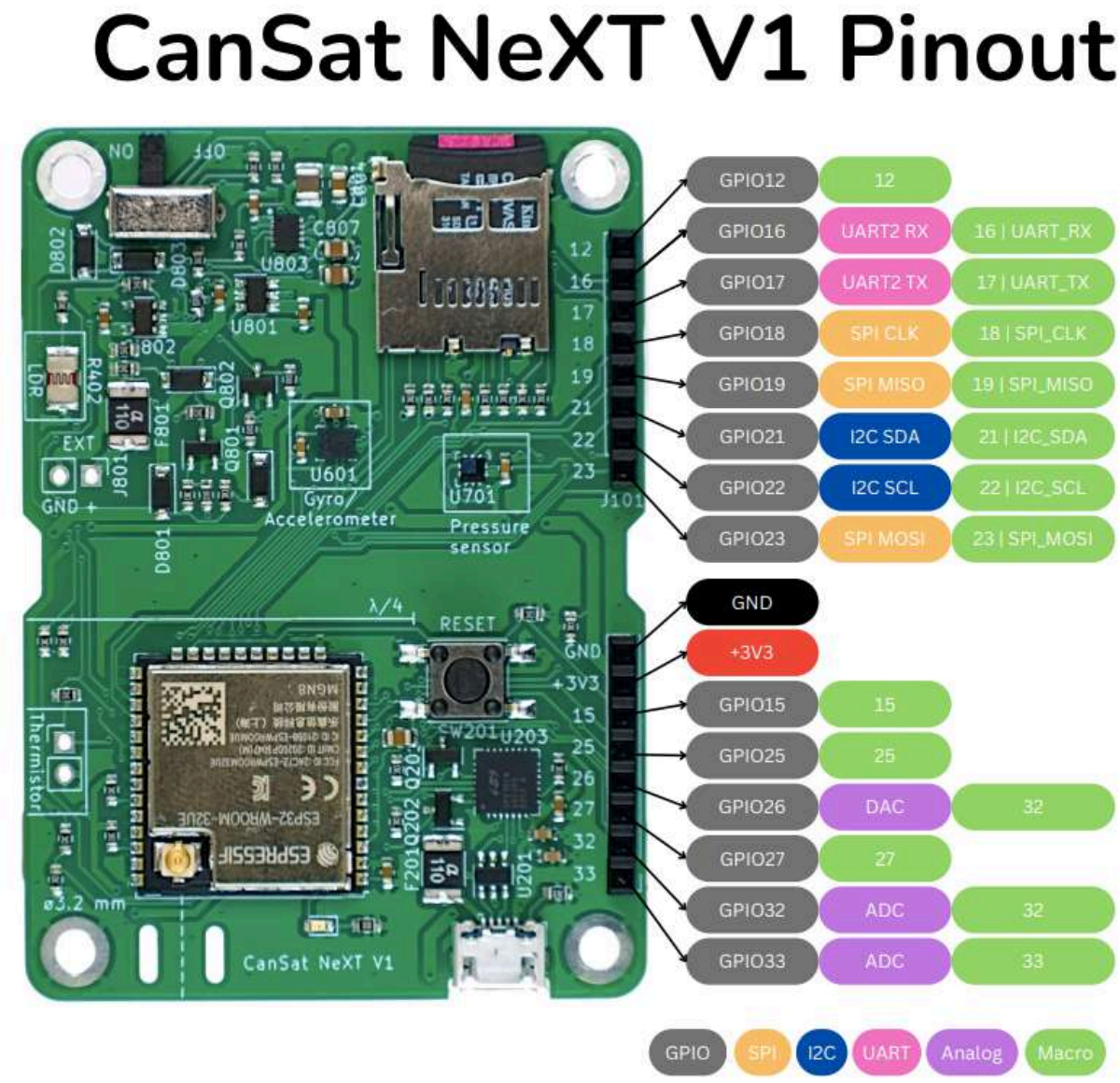
Pin Number	Library name	Note	Internal/External
0	BOOT		Used internally
1	USB_UART_TX	Used for USB	Used internally
3	USB_UART_RX	Used for USB	Used internally
4	SD_CS	SD card chip select	Used internally
5	LED	Can be used to blink on-board LED	Used internally
12	GPIO12		Extension interface
13	MEAS_EN	Drive 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
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
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

The pins that are only used in the extension interface can be used safely for any GPIO purpose. A special case is the GPIO15 pin, which if driven high during boot, will cause the MCU to send a debug print to the USB. This has no other effect for the MCU behavior.

Some pins are shared by both CanSat NeXT and the extension interface. These can be used externally as well, but using them for other functions than for SPI and I2C will generally disable the use of that system on board. So for example, the pins 21 and 22 should generally

be used only for connecting to different I2C devices, and pins 18, 19, and 23 should generally be used only to connect with devices that also use SPI. The reserved I2C addresses are 0x6A and 0x5C. All other addresses can be used to connect new sensors and other devices.



3.2 Flashing the MCU with Arduino IDE

Using the CanSat NeXT, ground station and other ESP32 development boards with Arduino IDE is relatively straightforward. Here are step-by-step instructions that can be used to get started with Arduino IDE and CanSat NeXT.

1. Install Arduino IDE. Both V1 and V2 work, use your preferred one.
2. Install ESP32 support to Arduino IDE
    a. In Arduino IDE, click Tools->Board->Boards Manager
    b. Search for ESP32
    c. Install esp32 by Espressif
3. Select ESP32 platform in Arduino IDE
    a. Tools->Boards->esp32
    b. Select ESP32 Dev Module
4. Connect CanSat NeXT to your PC
5. Select the board from Tools->Ports
6. You can now install programs onto ESP32 or CanSat NeXT!

While Linux and Mac don't require drivers for using ESP32, Windows is usually able to install the drivers required for ESP32 automatically. However, it's reported that in some cases such as secure corporate setups, the drivers are not installed automatically. In such cases, you may need to manually install the drivers from https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers.

Many ESP32 development boards have a write protection feature, where you have to manually press the BOOT on the dev board when initiating a flash from the IDE. This tells the board that you want to rewrite the software. This can be useful on the ground station, but it would have been a bit annoying to do on CanSat NeXT every time you want to reprogram it. This is why CanSat NeXT automates this and "presses" the button for you so there is no need to manually handle the device while flashing. In the future, over-the-air updates can possibly be supported as well.

4 Software library

The recommended way to use CanSat NeXT is with the CanSat NeXT Arduino library, available from the Arduino library manager.

4.1 Getting started

4.1.1 Arduino IDE

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

4.1.2 Add ESP32

CanSat NeXT is based on 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.

4.1.3 Install CanSatNeXT library

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

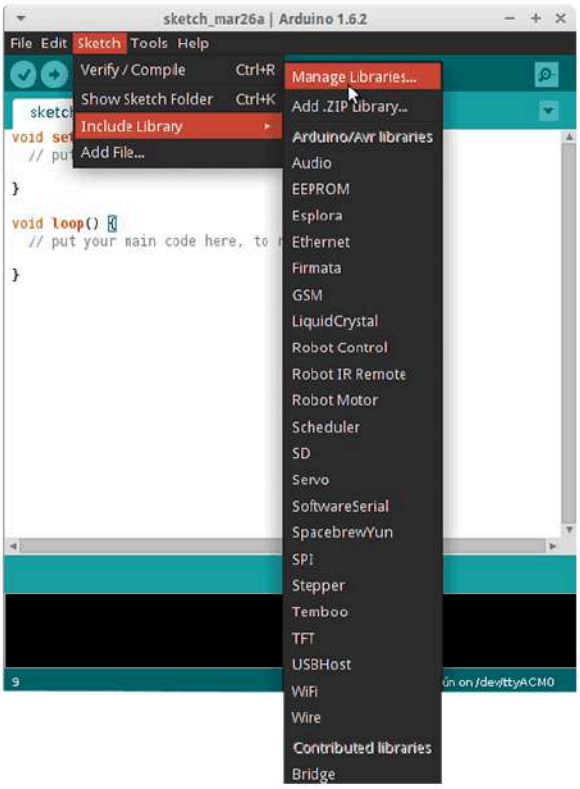


Image source: Arduino Docs,
https://docs.arduino.cc/software/ide-v1/tutorials/installing-libraries

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*.

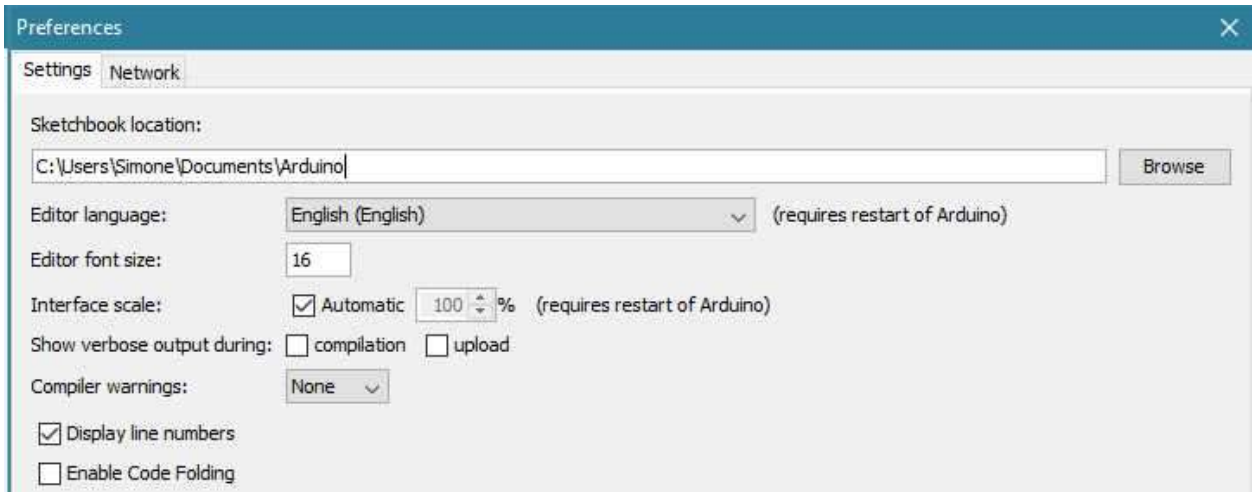


Image source: Arduino Docs,
https://docs.arduino.cc/software/ide-v1/tutorials/installing-libraries

4.1.4 Connecting to computer

Now you can plug in the CanSat NeXT to your computer. If it is not detected, you 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

4.1.5 You are ready to go!

You can now find CanSatNeXT examples from the Arduino IDE from Eile->Examples->CanSatNeXT

4.2 Functions

You can use all regular Arduino functionalities with CanSat NeXT, as well as any Arduino libraries. Arduino functions can be found for example 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.

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]`	6 byte mac address shared by the satellite and the groundstation. This is an optional parameter - when it is not provided, the radio is not initialized.
Used in example sketch	All

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 groundstation 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.

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 groundstation.
Used in example sketch	Groundstation receive

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.

readAcceleration(float &x, float &y, float &z);

Return type	`uint8_t`
Return value	Returns 0 if measurement was succesfull
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

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).

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 example sketch	IMU

This function can be used to read angular 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 angular acceleration. The acceleration is returned in units mrad/s.

float readPressure();

Return type	`float`
Return value	Pressure in mbar
Parameters	None
Used in example sketch	Baro

This function returns pressure as reported by the on-board barometer. The pressure is in units of millibar.

float readTemperature();

Return type	`float`
Return value	Temperature in Celsius
Parameters	None
Used in example sketch	Baro

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.

bool SDCardPresent();

Return type	`bool`
Return value	Returns true if it detects a SD-card, false if not.

Parameters	None
Used in example sketch	SD_advanced

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 if the SD-card is detected.

appendFile(String filename, String data);

Return type	`uint8_t`
Return value	Returns 0 if write was successful
Parameters	String filename, String data
`String filename`	Address of the file to be appended. If the file doesn't exist, it is created.
`String data`	Data to be appended at the end of the file
Used in example sketch	SD_write

This is the basic write function used to store readings to the SD-card.

printFileSystem();

Return type	`void`
Parameters	None
Used in example sketch	SD_advanced

This is a small helper function to print names of files and folders present on the SD-card. Can be used in development.

newDir(String path);

Return type	`void`
Parameters	String path
`String path`	Path of the new directory. If it already exists, nothing is done.
Used in example sketch	SD_advanced

Used to create new directories on the SD-card.

deleteDir(String path);

Return type	`void`
Parameters	String path
`String path`	Path of the directory to be deleted
Used in example sketch	SD_advanced

Used to delete directories on the SD-card.

bool fileExists(String path);

Return type	`bool`
Return value	Returns true if the file exists
Parameters	String path
`String path`	Path to the file
Used in example sketch	SD_advanced

This function can be used to check if a file exists on the SD-card.

uint32_t fileSize(String path);

Return type	`uint32_t`
Return value	Size of the file in bytes
Parameters	String path
`String path`	Path to the file
Used in example sketch	SD_advanced

This function can be used to read the size of a file on the SD-card.

writeFile(String filename, String data);

Return type	`uint8_t`
Return value	Returns 0 if write was successful
Parameters	String filename, String data
`String filename`	Address of the file to be written.
`String data`	Data to be written to the file
Used in example sketch	SD_advanced

This function is similar to the appendFile(), but it overwrites existing data on the SD-card. For data storage, appendFile should be used in its stead. This function can be useful for storing settings for example.

String readFile(String path);

Return type	`String`
Return value	All content in the file
Parameters	String path
`String path`	Path to the file

Used in example sketch	SD_advanced

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(String oldpath, String newpath);

Return type	`void`
Parameters	String oldpath, String newpath
`String` oldpath	Original path to the file
`String` newpath	New path of the file
Used in example sketch	SD_advanced

This function can be used to rename or move files on the SD-card.

deleteFile(String path);

Return type	`void`
Parameters	String path
`String` path	Path of the file to be deleted
Used in example sketch	SD_advanced

This function can be used to delete files from the SD-card.

void onDataReceived(String data); // CALLBACK

Return type	`void`
Parameters	String data
`String` data	Received data as an Arduino String

Used in example sketch	Groundstation_receive

This is a callback function, so the user code should not call this function. Rather, if you have a function with this name, the CanSat NeXT will call “back” this function when it receives data. The use is really simple - just add a function with this name anywhere in your code and use the data as you wish. This is also demonstrated in the Groundstation example, but the same concept can be used to receive data in the satellite.

void onBinaryDataReceived(const uint8_t *data, int len); // CALLBACK

Return type	`void`
Parameters	const uint8_t *data, int len
`const uint8_t` *data	Received data as uint8_t array
`int` len	Length of received data in bytes
Used in example sketch	None

This is similar to the function onDataReceived, except that the data is provided as binary instead of a String object. This is provided for advanced users who find the String too limiting.

void onDataSent(const bool success); // CALLBACK

Return type	`void`
Parameters	const bool success
`const bool` success	Bool indicating if data was sent successfully
Used in example sketch	None

This is another callback function which can be added to the user code if required. It can be used to check if the reception was acked by another radio, i.e. if another radio actually received the data or not. This can be useful for implementing resending or advanced radio protocols.

sendData(String data);

Return type	`uint8_t`
Return value	0 if data was sent (doesn't indicate ack)
Parameters	String data
`String data`	String format data to be sent.
Used in example sketch	Send_data

This is the main function for sending data between the ground station and the satellite. The parameter is simply the data to be sent. Note that the return value does not indicate if data was actually received, just that it was sent. The callback onDataSent can be used to check if the data was received by the other end.

sendData(char *data, uint16_t len);

Return type	`uint8_t`
Return value	0 if data was sent (doesn't indicate ack)
Parameters	char *data, uint16_t len
`char *data`	Data to be sent as a char array
`uint16_t len`	Length of the data in bytes
Used in example sketch	Nones

A binary variant of the sendData function. This is provided for advanced users who feel limited by the String object.

float adcToVoltage(int value);

Return type	`float`
Return value	Converted voltage as volts

Parameters	int value
`int value`	ADC reading to be converted to voltage
Used in example sketch	AccurateAnalogRead

This function can be used to convert an ADC reading to voltage. The function implements a calibrated third-order polynomial to convert the readings more linearly than a simple conversion. To use this function, provide it with an analog reading and it will output the voltage. Note that this is the voltage at the input pin, so to calculate the battery voltage you need to also consider the resistor network.

float analogReadVoltage(int pin);

Return type	`float`
Return value	ADC voltage as volts
Parameters	int pin
`int pin`	Pin to be read
Used in example sketch	AccurateAnalogRead

This is a function that can be used to replace the analogRead and read voltage directly instead. Uses adcToVoltage internally.

4.3 Extension interface

The 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
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

Extension interface pin lookup table

4.3.1 Communication options

The UART2 line serves as an unallocated communication interface for extended payloads. Use of I2C and SPI is supported as well, but the user must bear in mind that other subsystems exist 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.

With SPI, this distinction is instead made with 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.

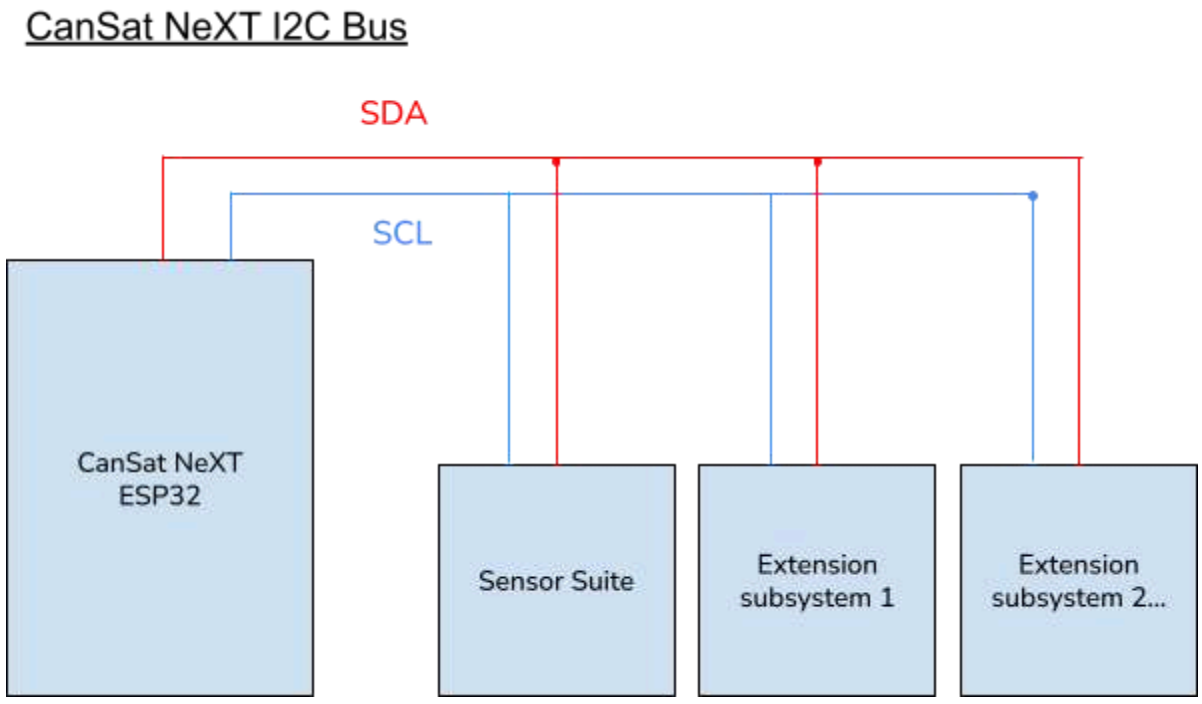


Image: the CanSat NeXT I2C bus featuring several secondary, or "slave" subsystems. In this context, the Sensor suite is one of the slave subsystems.

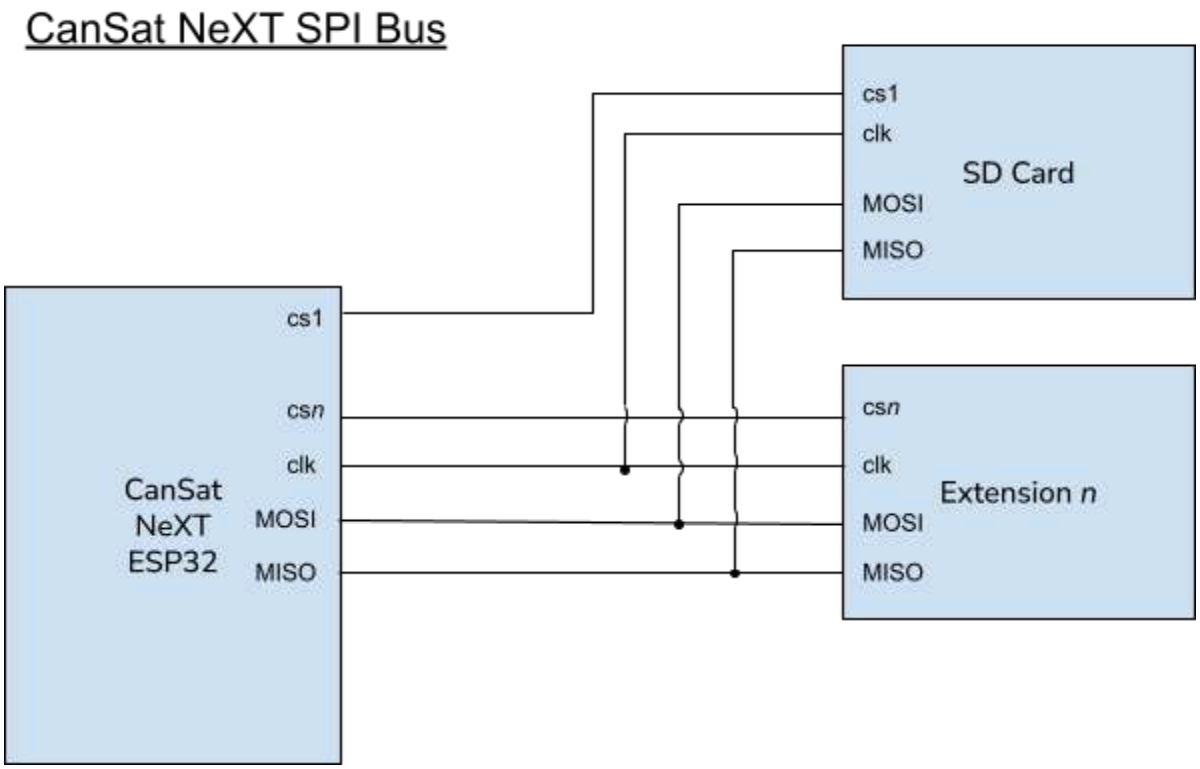


Image: the CanSat NeXT SPI bus configuration when two secondary, or "slave" subsystems are present. In this context, the SD card is one of the slave subsystems.

5 Communication system and antennas

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 will be available on the website.

6 On-board sensors

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.

6.1 IMU

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 will also be modifiable by the user.

6.2 Pressure sensor

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.

6.3 Analog to digital

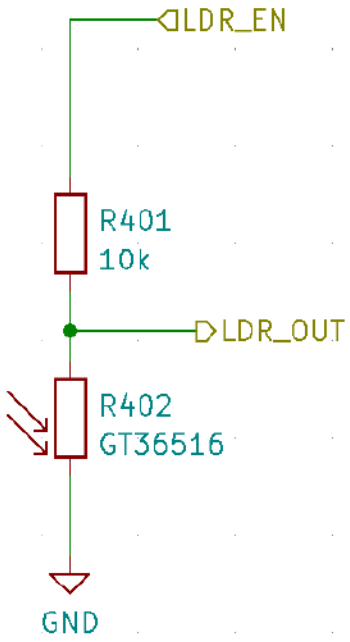
The 12 bit analog-to-digital converter (ADC) in ESP32 is notoriously nonlinear. This doesn't matter for most applications, such as using it to detect temperature changes or changes in LDR resistance, however making absolute measurements of battery voltage or NTC resistance can be a bit tricky. One way 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 AnalogIn. 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.

6.3.1 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.



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

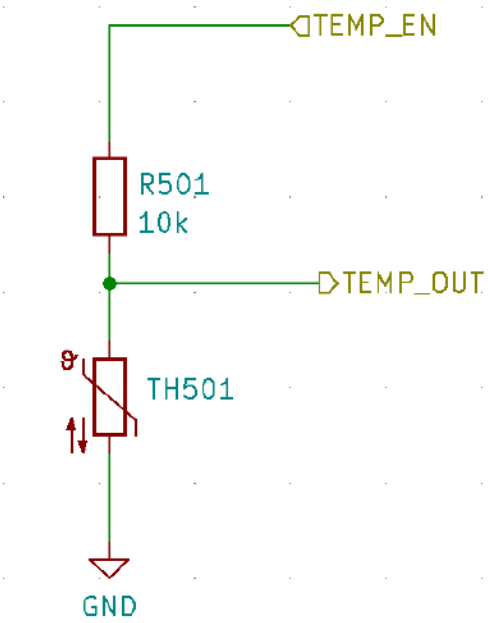
$$V_{LDR} = V_{EN} \frac{R402}{R401+R402} $$.

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.

6.3.2 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.

7 Power system

The power system of CanSat NeXT is somewhat more complicated than might seem necessary at first. The reason for the complexity is to make the use of the power system simple and safe, and have all the complexity in the circuit so that the user does not need to think about it any more than necessary. This is why this chapter is divided into two parts - the first part describes how to use the power system, and the second part dives into detail about how it works “behind the scenes”.

7.1 How to power CanSat NeXT

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. 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.

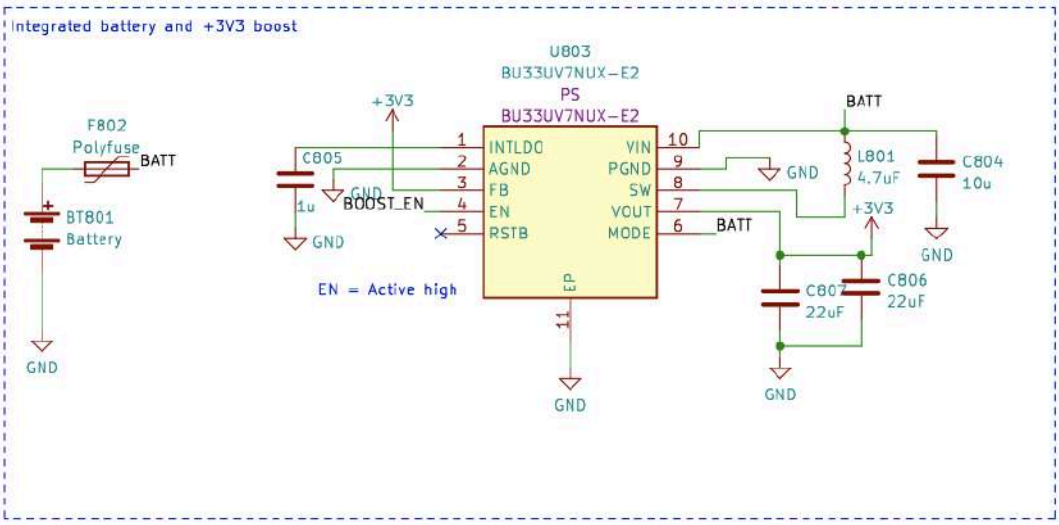
7.2 Adaptive multi-source power scheme

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.

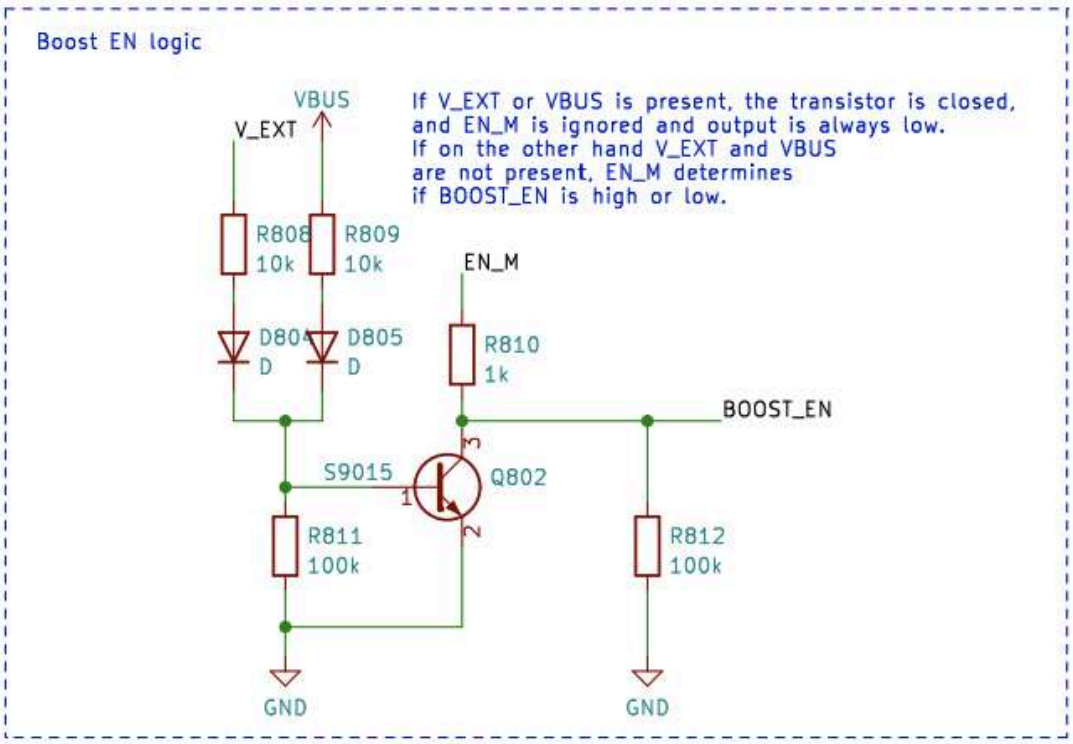
Below is the schematic for the boost converter. The IC is BU33UV7NUX, a boost converter specifically designed to give +3.3V from two alkaline batteries. It is enabled when the BOOST_EN line is high, or above 0.6 V.

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.

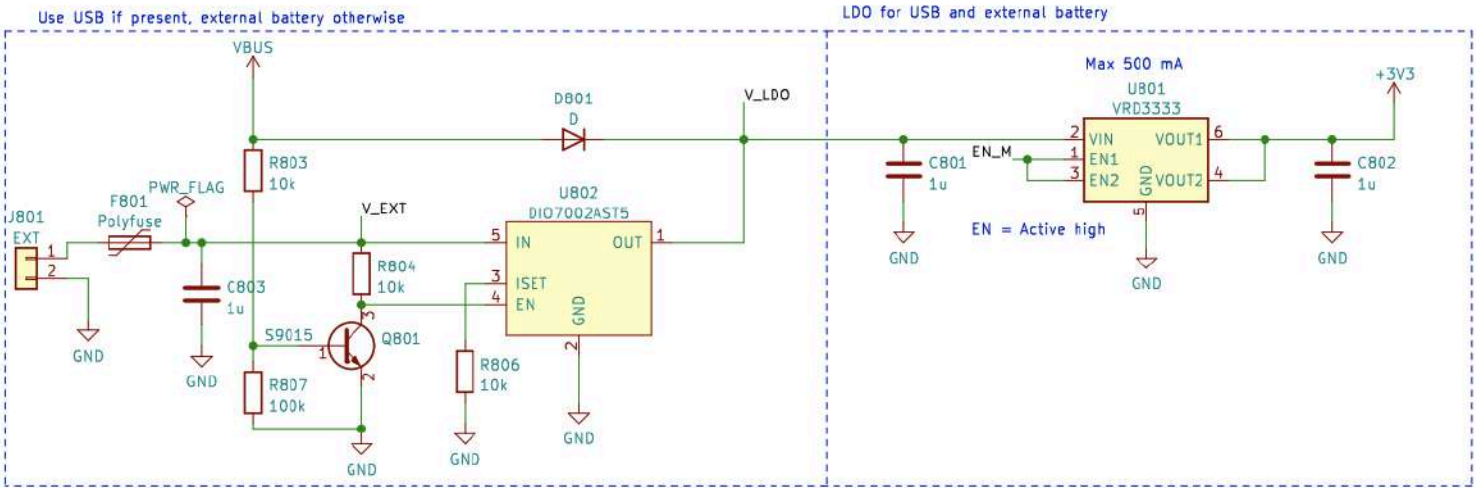


Or as a truth table:

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

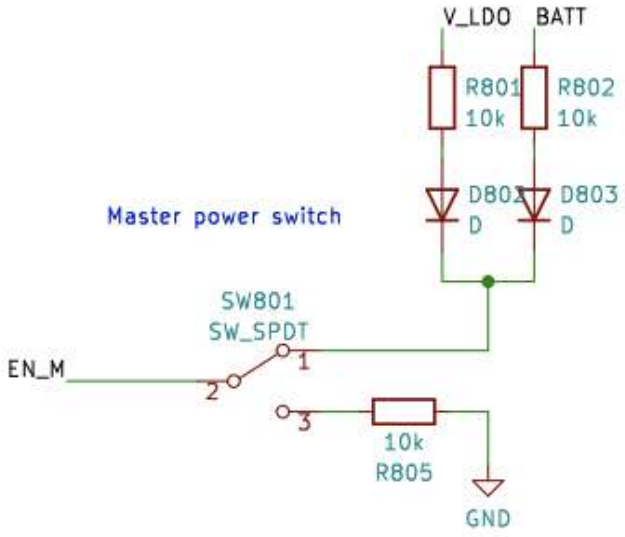
So BOOST_EN = EN_M ∧ !(V_EXT ∨ V_BUS).

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.



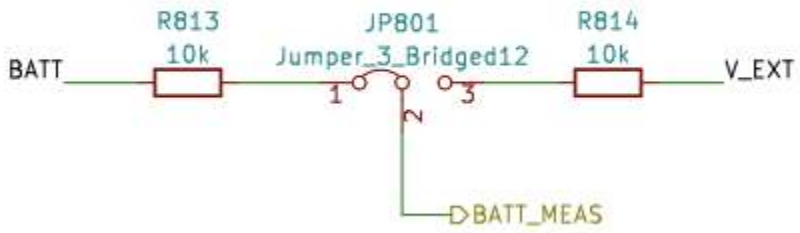
Overall, this circuit leads to a functionality where USB power is used if present, and V_EXT used when USB is not present. Finally, the EN_M is used to enable or disable the LDO.

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.

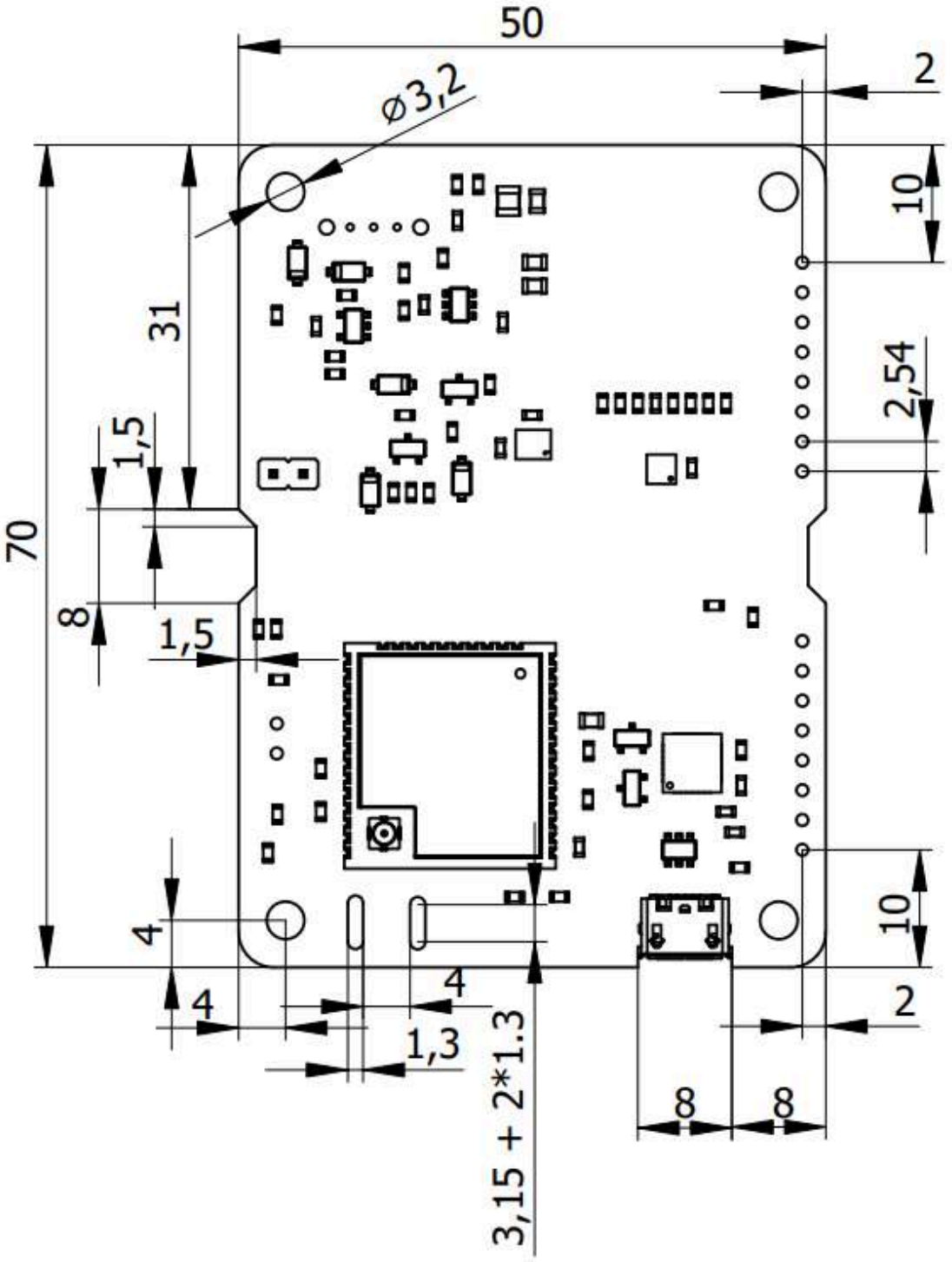


The jumper can be found from the bottom side of the CanSat NeXT main board. The jumper is quite easy to solder, so don't be afraid to cut the 1-2 line if you are using an external battery. It can always be resoldered to again use 1-2 instead.

8 Mechanical design

The CanSat NeXT main board is built on a 70 x 50 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 10 mm.



CanSat NeXT PCB dimensions