ARCTIC ASTRO NAUTICS

Kitsat Extension Interface

Work in progress - information is lacking and unverified

Version history

Date	Author	Edits
June 28th 2022	Samuli Nyman	Initial document
June 28th 2022	Tessa Nikander	Structure, minor additions to Section 4

Table of Contents

1. Extended Kitsats

This document describes the necessary mechanical, electrical, software and user interfaces that can be used to create 3rd party extensions to Kitsat. The document is not an exhaustive list of ways to connect to the satellite but describes the interfaces designed for extendability.

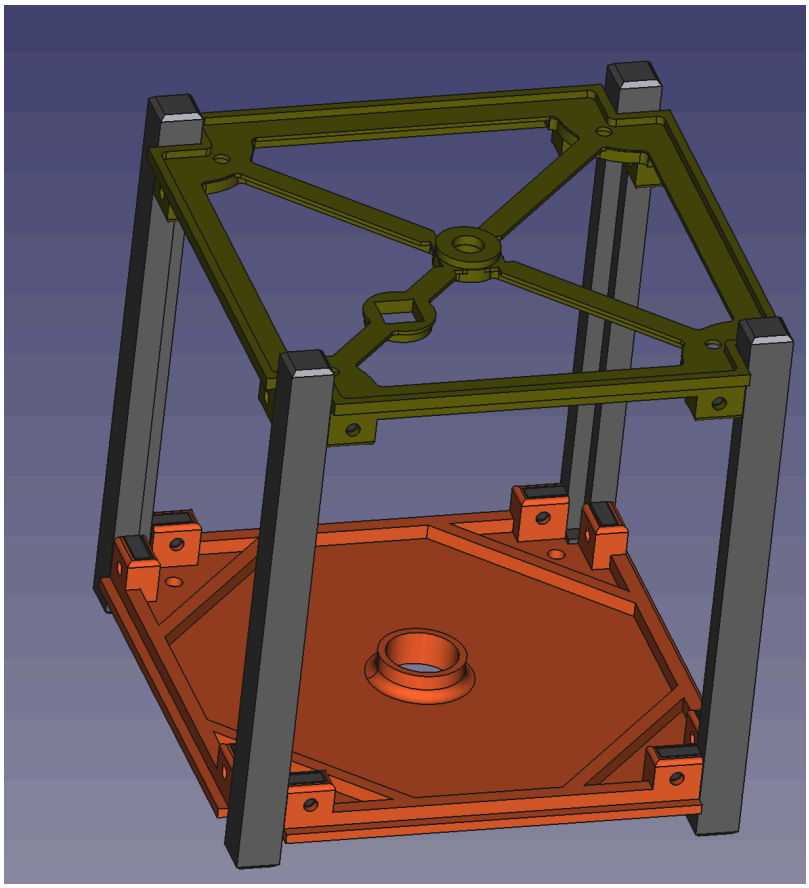
The Kitsat platform already implements many functionalities required by many simulatory or educational satellite platforms, such as a wired and wireless datalink, electrical power system, memory, camera, basic sensors for temperature, pressure, attitude and magnetometer, as well as a GPS for positioning. The Kitsat platform can be used as-is in many educational programs, but it is also possible to extend to suit a specific need or to serve as a platform for educational programs.

Technical materials can be found from the Kitsat extension repository available at:

2. Mechanical interface

2.1 Attachment options

The frame of Kitsat consists of 3 main parts - A plastic bottom plate and top frame and the anodized aluminum rails. These are used to support the electronics stack inside the satellite.



The basic options for attaching the extensions mechanically to the satellite revolve around these three main components - You could replace the top frame or the bottom plate with your custom parts and attach the payload to the plate interfaces. This approach is simple and would work well for a small payload. One option is also to utilize the attachment point in the center of the top frame, intended for hosting the Kitsat, to attach the payload.

Similarly, you could attach a payload to a side of the Kitsat, replacing one or more of the solar panels.

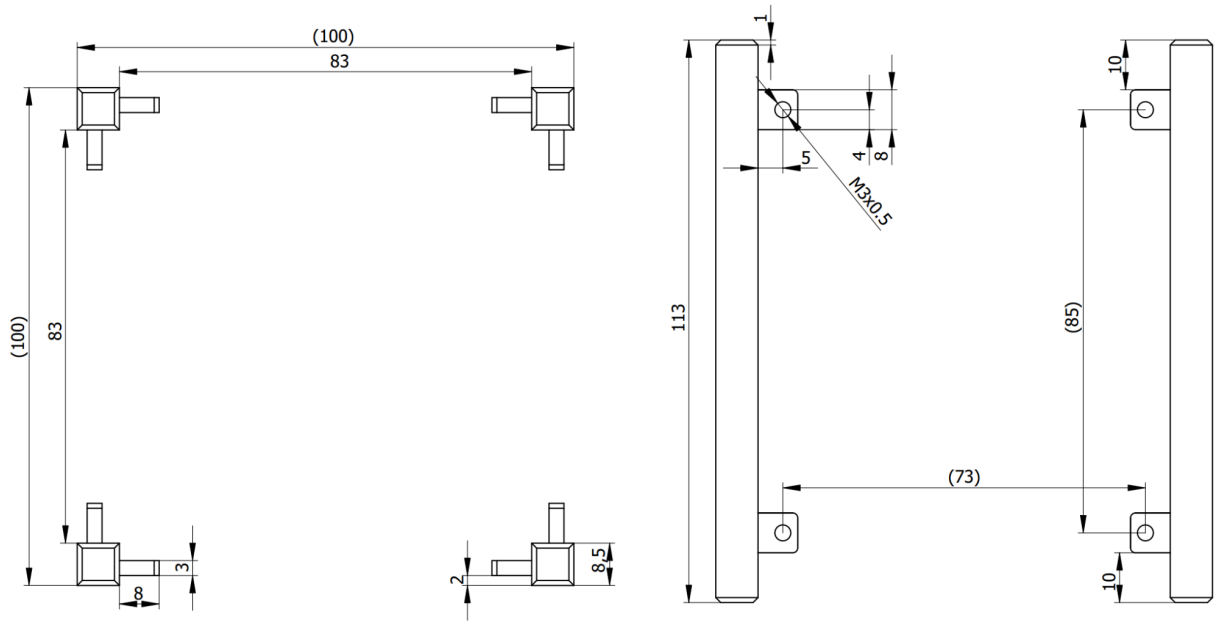
The most robust option for the modification is to extend the entire satellite, making it from 1U to 2 or 3U CubeSat. However, this is usually not cost-efficient as it requires ordering custom machined parts. Depending on the size, the cost of the rails is around 50-100 € per piece when ordered in small quantities. On the other hand, this allows for customized accurate interfaces and a robust end-product. Furthermore, engineering students often lack the experience of working with machine shops, making the process of designing and ordering the parts a valuable lesson in itself. That being said, 3d-printing the rail pieces could also be an option, depending on the project requirements.

The solar panels are of course also 1U-sized, but in many instances, it is possible to simply have multiple of them, or cover the sides with a custom-designed piece. Also, it is possible to design larger Kitsat-style solar panels as we've done with the 3U-Kitsat.

Overall, there are multiple ways to attach your payload to the Kitsat, and creative engineers can surely come up with other ones. Below is a quick recap of the methods described here.

Attachment method	**Kitsat parts replaced**	**Suitable for**	**Disadvantage**
Plate interface - top	Top plate, antennas	Medium size payloads	The antennas should be reattached or reimplemented
Plate interface - bottom	Bottom plate	Medium size payloads	The payload will probably block the camera
Solar panel area	One solar panel	Thin payloads	Removes solar panel, moves the center of mass to the side
Top plate hoisting ring	None	Small payloads	Blocks the main way for hosting Kitsat
Extension	Rails	Large payloads	Cost of manufacturing

In the drawing, below, the rail dimensions are presented in the configuration they will be in the assembled Kitsat.



Additionally, a step-model for the basic frame is available in the Kitsat extension repository.

2.2 Existing solutions

We have built a couple of custom Kitsats using some of the options described in the previous chapter. Some have used a different design of solar panels, and one had a 3d-printed bottom plate which housed a motorized lens assembly for changing the zoom of the camera. The most versatile modification, however, is the 3U-Kitsat. It consists simply of longer rails and larger solar panels, as well as mid-attachment supports. This allows for a simple interface to attach user payloads.

2.3 Compatibility with the CubeSat standard

While close, the Kitsat frame design does not apply to the strictest definitions of a CubeSat frame. Firstly, the 1U frame pieces are 113 mm long, as opposed to 113.5 mm defined by the standard. Secondly, the margin of the end pieces is not tight enough to comply with the ±0.1 mm margin for the side length of CubeSats. This should be taken into account when planning

the use of Kitsats with test equipment designed for CubeSats. Furthermore, Kitsat has not been designed or tested to withstand the vibration loads required from CubeSats.

3. Electrical interface

Most payloads also require an electrical connection for power and the data link. The bus has an +3V3 line as well as two battery voltage lines - one that is on when the satellite is powered, and one that does not shut down when the satellite does, intended for backup power for memories, real-time clocks (RTC) or similar very low current devices. For data communication, the easiest solution is to use a USART line on the bus reserved for extension systems. The software support for extensions also uses the USART line.

3.1 PCB format

Kitsat uses a stackable PCB format, making it easy to add new subsystems as long as the dimensions match. The PCB format is not compatible with the standard PC/104 format. The easiest way to achieve compatibility is to use the Kitsat KiCAD template as the starting point, as that not only provides the PCB dimensions and the bus connector, but also the bus definition in the schematic. Kitsat uses Harwin M20-6102045 as the bus connector, but any general 2.54 mm pitch headers will work as well.

The Kitsat KiCAD template is available in the Kitsat extension repository.

The Kitsat subsystem boards are all 2-layer PCBs, and therefore the template is also configured for two layers. More layers can be added through the KiCAD board setup.

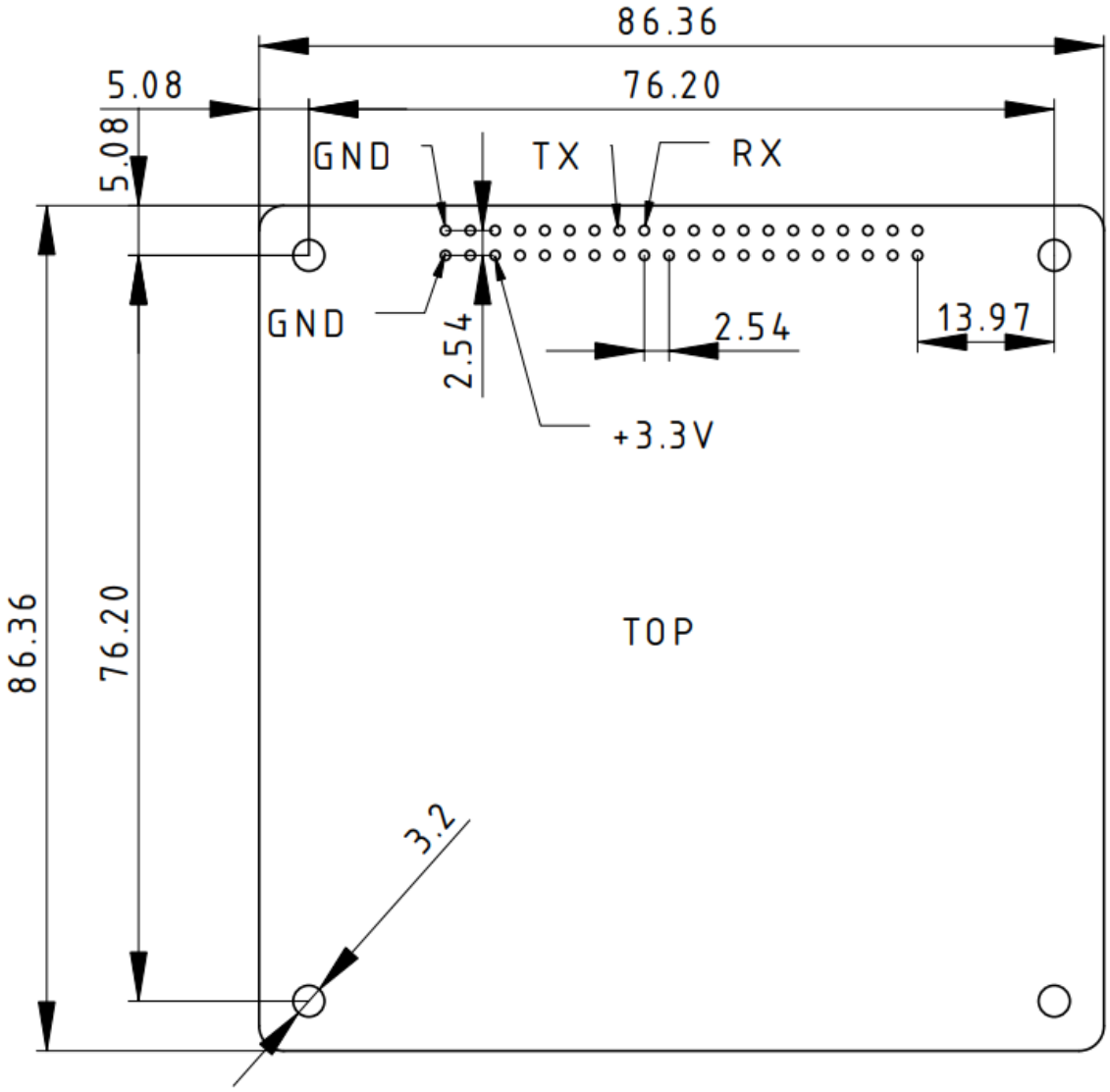


Figure 2: Kitsat board dimensions

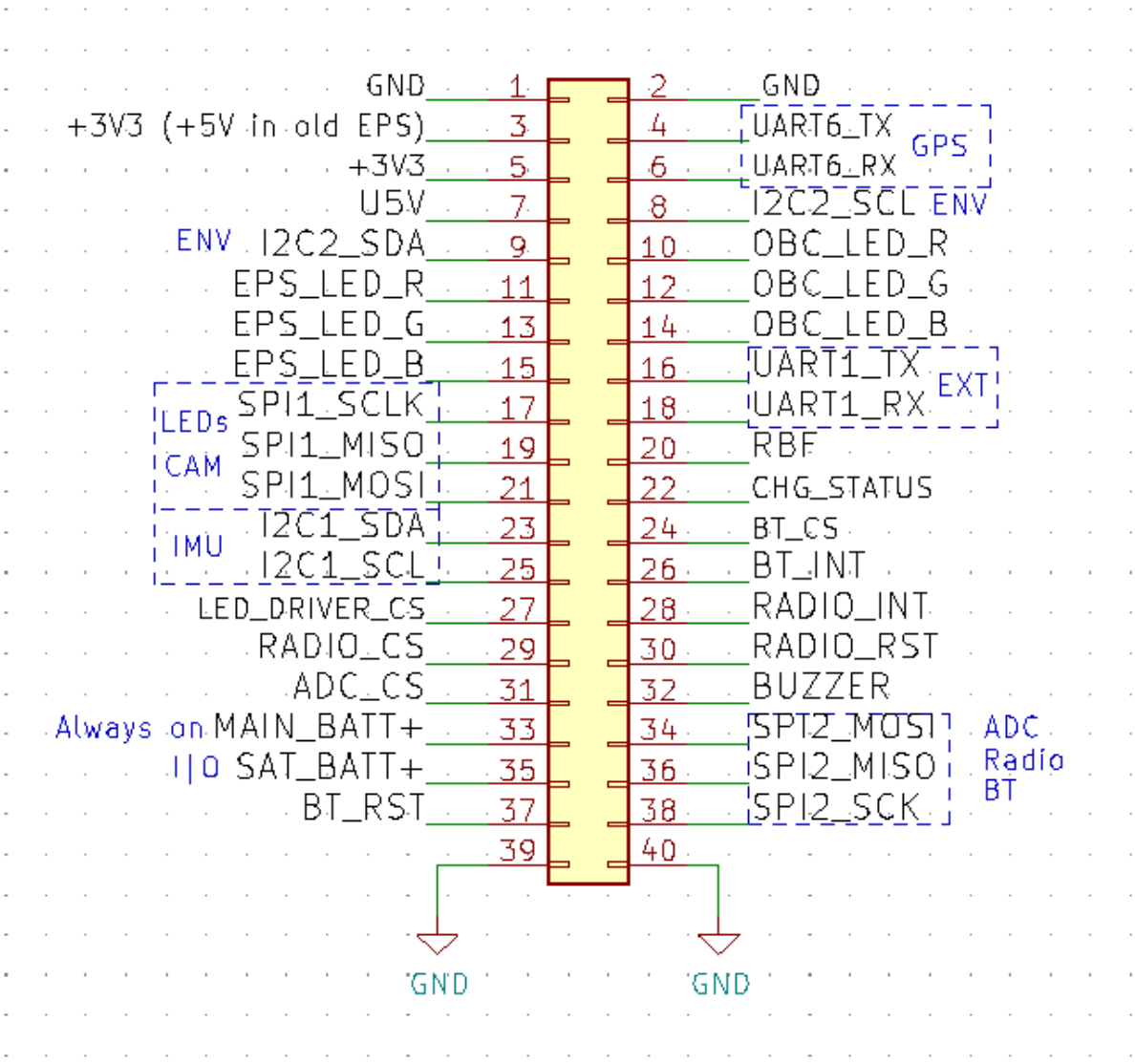


Figure 3: Kitsat pinout

3.1.1 Power lines

The Kitsat KiCAD template also shows the pinout for the bus power and data connections, as can be seen in figure 3. The other pins are reserved for the internal use of Kitsat. The MAIN_BATT+ line is connected directly to the battery through a self-resetting 1.5A fuse, while the BATT+ is only active while either of the Kitsat power switches is on. This also allows the extension to control the Kitsat power by connecting BATT+_bck to the BATT+ line, or even

providing the BATT+ voltage from a completely different source. The battery in Kitsat is a single-cell Li-Ion battery, with LVD at 2.8V and a max voltage of ~4.2V.

The Vbus line is connected to the USB power line and is used to charge the satellite. While this means that the line can be used to charge the satellite, it also makes it potentially unsafe if the USB is connected at the same time.

Pins 3 and 5 are +3V3 lines in Kitsat EPS:s after 1.4, while pin 3 is a +5V line in older versions. The +3V3 is regulated from the battery with an LD39150DT33-R LDO. The drop-out voltage is roughly 200 mV, so the +3V3 line starts to drop below 3.3V when the battery is empty. Other subsystems remain operational until LVD, but the camera is shut off while the battery is around 3.2V, or roughly 10% battery remaining. The maximum current output for the LDO is 1.5A, and the line is also fused through a self-resetting 1.5A fuse.

Furthermore, the battery also has internal over-current protection at 5A, which can be reset either by removing the battery and putting it back in or by disconnecting and reconnecting the battery-reset jumper in EPS versions 1.4 and after.

3.1.2 Data lines

The only data line reserved for extensions is the UART1 line in the satellite. The TX and RX are named from the Kitsat OBC perspective, so TX = Main output, subnode input and RX = Main input, subnode output.

The UART parameters are 8-N-1 115200 baud.

3.2 Cables

The satellite has some cabling inside of it. One 3-wire cable for each solar panel, one going to the top panel for the small power switch, and a coaxial RF cable from the radio to the antenna. The modified Kitsat should also implement these lines or replace their functionality in another way to keep all the functions operational. Special attention should be given to the coaxial RF cable, as operating the satellite without a cable might damage the radio. In practice, however, this doesn’t seem to occur due to the low power of the radio.

4. Software interface

The software interface is designed to be as flexible as possible to support different kinds of payloads. The interface simply passes the commands from the users directly to the payload and allows the user to customize the protocol and commands to their specific mission. The example communication section shows one way to command a payload, but the user is encouraged to design a system best suited for their own use case.

The Kitsat protocol has several commands implemented to enable communication to the payload via the UART line. A table of commands is below. The commands can be issued through the USB or via the radio interface.

Command	ID	Purpose
ext_tx str data	1	Send a char array with a max length of 53 bytes to the payload
ext_rx	2	**Used internally** for data received from the payload when passed through the Kitsat command interface.
ext_passthrough int state	4	Enables “uart passthrough” so that any data received from the payload is passed to the Kitsat command interface, i.e. to the GUI. 1 to enable, 0 to disable. Enabled by default.
ext_start_recording str filename	5	Starts logging any data received from the payload to the sd card, to the file named in the command. If the file exists already, the data is written at the end of the existing file. The filename can be any ASCII string, such as “data.dat” or “asdf.csv”.
ext_stop_recording	6	Disables the logging of payload messages to the sd-card.
ext_recording_size	7	Queries the log file size in bytes

ext_last_output	3	Get the latest recorded data.
ext_command int state	8	Enables command interface from the external payload to the satellite. 1 to enable, 0 to disable. Enabled by default.

4.1 Protocol

The protocol depends on the payload interface implementation. If the Kitsat GUI is used for commanding the satellite and the payload, all commands and responses should be in ASCII format to be visible in the GUI terminal. The data stream can still be in binary format when saved to the SD card, but an ASCII format such as CSV can also be used.

4.2 Example communication

As an example, let's consider a simple imaginary payload - a dosimeter. The dosimeter has commands to check its status, query simple data items like the average and maximum and a data stream. Here is one possible way to operate this payload.

Command	**Parameter**	**Response**	**Explanation**
ext_tx	start	-	Passes the "start" string to the payload, which turns the sensor on
ext_passthrough	1	-	Enable the passthrough of uart rx data.
ext_tx	status	"Active"	Passes the parameter to the payload, which

			responds.
ext_tx	latest	75.125	Gets the reading
ext_start_recording	data.dat	-	Satellite creates the file “data.dat” to the sd-card and starts saving all received data to the file.
ext_passthrough	0		Disable passthrough so that we can rapidly transmit data without cluttering the command interface.
ext_tx	data_start	-	Tell payload to start streaming data
ext_tx	gain 1.0	-	Command the payload to switch a parameter
ext_recording_size	-	1337	Get the file size to see that you are receiving data
ext_tx	data_stop	-	Stop sending data
ext_stop_recording	-	-	Stop saving data to the SD-card
ext_passthrough	1	-	Enable passthrough to see the responses
ext_tx	status	“Active”	Request payload status
ext_tx	sensor_stop	-	Send command to the payload to stop

			the sensor
ext_tx	status	“Disabled”	Send command to the payload to see the status

4.3 Commanding the satellite from the payload

The payload is also able to send any command to the Kitsat, to allow more possibilities for the custom missions. The commands are sent over the same uart interface, and as long as they match the Kitsat command protocol, they are interpreted and executed identically to commands received from the radio or USB interface.

The command should be sent as a buffer, which follows the format shown here:

target_id	command_id	Data length	Data	FNV checksum
1 byte	1 byte	1 byte	Data length bytes	4 bytes
Subsystem specific ID	Command specific ID	Length of the data, i.e. parameters of the command	Parameters in the command. Typically ASCII data.	Fowler–Noll–Vo hash function - Wikipedia

The available commands with their corresponding target and command ids can be found from the Kitsat GUI by selecting Edit -> Open configuration files -> command_list.csv

As an example, let’s consider the command to make start the led- random pattern. From the command list, we can find that the target_id for LEDs is 9, while the command_id for the “led

show” is 2. The parameter in this case is the delay between updates in milliseconds, written in ASCII form. Let’s use “100” in this case.

target_id	command_id	Data length	Data	FNV checksum
9	2	3	“100”	The checksum
0x09	0x02	0x03	0x31, 0x30, 0x30	0x66, 0x4b, 0xd7, 0xde

An easy way to create a valid Kitsat command is using the following C-functions:

```
uint32_t ufnv(char *bytes, int str_len)
{
  uint32_t hval = 0x811c9dc5;
  uint32_t fnv_32_prime = 0x01000193;
  uint64_t uint32_max = 4294967296;
  for(int i = 0; i< str_len; i++)
  {
    hval = hval ^ bytes[i];
    hval = (hval * fnv_32_prime) % uint32_max;
  }

  return hval;
}

uint8_t createKitsatCommand(char *buf, uint8_t subsystem_id, uint8_t command_id, char *parameters, uint8_t
parameters_len)
{
    buf[0] = subsystem_id;
    buf[1] = command_id;
    buf[2] = parameters_len;
    memcpy(buf+3, parameters, parameters_len);
    uint32_t fnv = ufnv(buf, parameters_len+3);
    memcpy(buf+3+parameters_len, &fnv, 4);
    return parameters_len+7;
}
```

For example, the led-random pattern used in the previous example would look like this. First, a buffer is created to store the command. Next, the command is generated using the createKitsatCommand function, where the parameters are buffer, target id and command id, then the parameters and length of the parameters. Finally, it is written on the UART line. The write command depends on the used environment.

```
    char msg[64];
    uint8_t len = createKitsatCommand(msg, 9, 2, "100", 3);
    satellite.write(msg, len);
```

4.4 Using Kitsat data in your payload

In some situations, it might be useful to use data from Kitsat sensors in your payload. Possible use cases are for example checking the environmental data such as surrounding air pressure, or for example reading time from the GPS. This can be done by sending a command to the satellite like described in the previous chapter, and then reading the output. As the data will be packetized and arrives as part of a stream of characters, reading it can be a bit tricky. The code below presents a struct called telemetry to hold the data, and then a parser function for that struct. The received data from UART or serial can be given to this function, and it returns an instance of the telemetry struct. The valid_telemetry flag can be used to check if reading the data was successful. Note that if using this implementation, the data buffer needs to be freed after the data is not anymore needed to avoid memory leak.

```c
struct telemetry{
    uint8_t target_id;
    uint8_t command_id;
    uint32_t timestamp;
    uint8_t data_length;
    char *data;
    uint8_t valid_telemetry=0;
};

struct telemetry parse_telemetry(char *buf, size_t len)
{
    struct telemetry tm;
    // a valid packet will have at least 11 bytes
    if(len<11) return tm;

    tm.target_id = buf[0];
    tm.command_id = buf[1];
    tm.timestamp = ((uint32_t)buf[2] << 24) | ((uint32_t)buf[3] << 16) | ((uint32_t)buf[4] << 8) |
(uint32_t)buf[5];
    tm.data_length = buf[6];

    // Check if the length of buf is long enough to hold the expected number of bytes for data field
    if (len < 7 + tm.data_length) return tm;

    // Copy the data from buf to the data field of the telemetry struct using memcpy
    tm.data = (char*) malloc(sizeof(char) * tm.data_length);
    if (tm.data == NULL) {
        tm.valid_telemetry = 0; // Memory allocation failed
        return tm;
    }
    memcpy(tm.data, &buf[7], tm.data_length);

    uint32_t received_checksum = ((uint32_t)buf[tm.data_length+10] << 24) |
((uint32_t)buf[tm.data_length+9] << 16) | ((uint32_t)buf[tm.data_length+8] << 8) |
(uint32_t)buf[tm.data_length+7];
    uint32_t calculated_checksum = ufnv(buf, tm.data_length+7);
    if(received_checksum == calculated_checksum){
        tm.valid_telemetry = 1;
    } else{
        free(tm.data);
        return tm;
    }

    return tm;
}
```

```
void interpret_command()
{
    struct telemetry tm = parse_telemetry(satellite_buf, satellite_buf_counter);
    if(tm.valid_telemetry)
    {
        printf("%.*s\r\n", tm.data_length, tm.data);
        free(tm.data);
        return;
    }
... ...
```

5. User interface

5.1 Kitsat GS GUI

The easiest way to send any commands to Kitsat is using the Windows ground station software. While this enables simple operation very easily, it is also the most limiting and requires manual operation of the payload, one command at a time,

5.2 Software solutions

Alternatively, the communication can be done through user-developed software. The protocol is pretty easy to use for the most part, but it has some confusing parts, especially regarding image data handling. Luckily, we have implemented a python library to take care of the application-specific tasks.