DeSCENT ChipSat Ground Station

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University in Partial Fulfillment of the Requirements for the Degree of Master of Engineering, Electrical and Computer Engineering

Submitted by

Grace Tang

MEng Field Advisor: Dr. Van Hunter Adams

Degree Date: December 2025

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: DeSCENT ChipSat Ground Station

Author: Grace Tang

Abstract:

The DeSCENT mission aims to investigate the feasibility of gram scale spacecraft, known as ChipSats, for suborbital, atmospheric sensing. The goal of the mission is to deploy a fleet of 100 ChipSats from Blue Origin's New Shepard Vehicle at approximately 100km altitude. This project aims to design and implement a suitable ground station to receive and log transmitted ChipSat packets during the mission. The ground station must operate as a standalone system, requiring minimal setup and maintaining power for the duration of the setup and mission. A system was designed using the Raspberry Pi Pico microcontroller, the RFM95 LoRa module, a GPS, and an SD card reader/writer to store received ChipSat data. On the software side, the system was programmed using the Pico SDK, and the Radiolib library was used to implement LoRa communication. The LoRa radio and SD card functionality were completed, while the GPS still remains to be implemented. Other scoping was done for the system, including testing different packet formats and LoRa signal parameters, estimating power consumption, and range testing. Future work entails implementing a GPS module, further investigation into some of the errors that were encountered, and exploring some additional system features such as a motorized tracking antenna and remote data access.

Table of Contents

Executive Summary

The DeSCENT mission explores the use of gram-scale ChipSats for suborbital atmospheric sensing by deploying 100 ChipSats from Blue Origin's New Shepard vehicle at approximately 100km altitude. This differs from previous ChipSats which focused on their use in an orbital environment. The goal of this project is to design and implement a suitable ground station to receive and log transmitted ChipSat packets during the mission, allowing for easier analysis of the collected data afterwards.

The requirements of the ground station are as follows. It must be able to communicate over 915MHz LoRa with the ChipSats, log and store received packet data, maintain power as a standalone system for the duration of the setup and mission, and require minimal setup and intervention to deploy. The goal of this is to create an easy to use, scalable system for collecting mission data, as the significant size of the ChipSat fleet requires a more systematic approach.

The system itself consists of a Raspberry Pi Pico microcontroller for central control, an RFM95 LoRa module for radio communication, and an SD card reader/writer to allow for easy data logging and retrieval. The components were implemented in a breadboard design, and a GPS module was selected for the system as well, though it remains to be implemented. Software for the system was developed using the Pico SDK, and the Radiolib library was used for LoRa.

Additional system analysis was performed, including an estimation of power consumption to inform battery selection and the evaluation of different ChipSat packet formats and LoRa signal parameters. A system capable of receiving LoRa packet transmissions and simultaneously storing the data in these packets to an SD card was achieved, creating a baseline hardware and software foundation to build off of for future development. In addition to GPS integration and designing a power system, future work may explore other features such as a secondary, motorized tracking antenna and remote data access functionality.
Future Work and Recommendations

Introduction

Background

The goal of this project is to design a ground station for the upcoming DeSCENT mission. As a part of Cornell's Space Systems Design Studio, the mission builds off of previous ChipSat missions. A gram scale "satellite on a chip", missions such as Sprite, KickSat, and Alpha demonstrated their functionality as a device, communication ability, and survivability in space. The goal of DeSCENT differs from these in that it aims to determine their potential for suborbital, atmospheric use. The DeSCENT mission aims to deploy one hundred ChipSats from the New Shepard suborbital launch vehicle at approximately 100 km apogee, the Kármán line [10]. ChipSat sensor data shall be collected from their descent trajectories and landing positions, containing in-situ atmospheric sensor data and measurements of free-fall kinematics. This data will be used to characterize free fall dispersion dynamics. Additionally, in order to determine their survivability from terminal velocity impact with the ground, we want to recover as many ChipSats as possible after landing. The diagram in Fig. 1 shows a summary of this mission.

In order to both collect transmitted ChipSat sensor data during the mission and for ease of finding them after they land, a ground station is needed. The mission also requires designing a deployer to fit one hundred ChipSats and eject them from the New Shepard launch vehicle, shown in Fig. 2. The design and components of the ChipSats themselves are shown in Fig. 3. A STM32 based microcontroller with integrated LoRa radio is used for command and data handling, with a custom helical PCB antenna design. LoRa communication was previously used and verified for Alpha ChipSat, and its long range and low power capabilities also make it useful here. A PCB antenna instead of the previous dipole antenna design was used to optimize the number of ChipSats that can fit in the deployer. The ChipSats also have an IMU and environmental sensor, used for gathering atmospheric and kinematic data, as well as a GPS. This GPS data is valuable in that it not only allows for analysis of dispersion dynamics, but transmitted GPS data can also be used to help locate the ChipSats once they land. As a backup, the ChipSats have flash memory to store mission data in the case that transmissions fail. In order to power the ChipSats, they have both solar panels and a battery. Because this flight is much shorter in duration, approximately ninety minutes compared to a couple of days, the ChipSats should remain powered on for as much of the mission as possible. To avoid issues with unpredictable solar power, a backup LiPo battery is included.

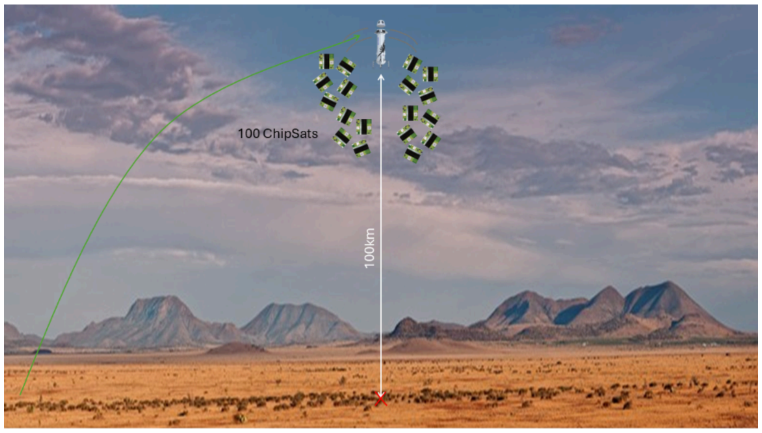
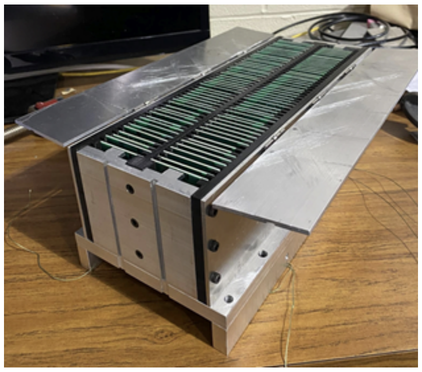


Fig. 1: Demonstration of Suborbital ChipSats Ejected from New Shepard Test Flight [10]



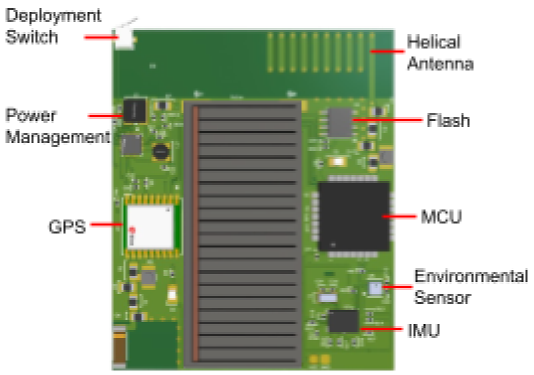


Fig. 2 (Left): Flight Unit Deployer, Fig. 3 (Right): DeSCENT ChipSat CAD Model [10]

Problem Statement and Requirements

In order to analyze ChipSat sensor data and evaluate mission results, a method is needed to to collect in situ transmit data from the fleet of ChipSats. This motivates the design of a ground station for the mission. The requirements are as follows:

1. Can communicate with multiple ChipSats over LoRa. This is the most basic functionality requirement, and will allow us to verify that the ChipSat transmit functionality is working. Only receive functionality is required, as the ChipSats are transmit only, and do not need to be issued commands. The ground station must be able to receive and decode 915 MHz LoRa packets, matching the following ChipSat transmit signal parameters: bandwidth (BW), spreading factor (SF), coding rate (CR), sync word (SW), and preamble length (PL). These signal parameters are to be determined depending on upcoming range testing, and are detailed in the Results: Debugging section. The station must also be able to receive and decode packets with multiple ChipSats transmitting as it is not practical to have a ground station for each ChipSat.

2. Can store and log received data packets. This is needed to allow us to retrieve and process ChipSat data after the mission. There should ideally be a convenient and reliable method to extract the logged data either during or after the mission.

3. Ease of setup and scalability. In order to receive ChipSat packets in previous testing, a laptop connected to a LoRa receiver module was used, and data was logged over serial using PuTTY. This method is time consuming to set up and maintain as a person is needed to configure the laptop and monitor the system throughout the mission duration. These limitations motivate the design of a ground station that requires minimal setup and can operate autonomously. Additionally, deploying multiple ground stations for this mission would be beneficial since it would help accommodate the size of the ChipSat fleet and allow for more data on signal strength and dispersion dynamics to be collected. It would also maximize the chances that all ChipSats are heard. Our current laptop based method would require excessive personnel, limiting scalability. As a result, a more scalable solution is also desirable, and having less setup would help with this.

4. Maintain power for the duration of the setup and mission itself. We estimate the mission to be about ninety minutes, and the setup to be done six to twelve hours beforehand. Because the ground stations need to be as autonomous as possible, they must not require manual configuration after they are set up. As a result, they are expected to remain powered and functional for the duration of the setup in addition to the mission. Since we aren't limited by size or form factor, there aren't many additional constraints in implementing this power requirement beyond a standalone system.

Below are some additional features that were explored while scoping the project. These would be nice to include, but are not required for mission success.

1. Remote data access. In addition to being able to physically retrieve the data after the mission, having the ability to remotely access the data would add an additional level of convenience. This could be implemented by transmitting the received ChipSat packets via some wireless method as it arrives at the ground station. Potential methods include via cellular or a LoRa WAN network, depending on the required range. This data could then be accessed from a remote dashboard, where we would be able to see it in real time. However, implementing this would add additional layers of complexity not required for mission functionality, so this was shelved for later.

2. Secondary tracking antenna. This would entail adding a secondary, highly directional, motorized antenna that can point to specific ChipSats based on GPS data from received packets. To implement this, the heading and location of the ground station, as well as location of the ChipSats must be known. Since this design would be heavily mechanical and require tuning an additional complex system, it was not the focus of this project. In

the future, it may be expanded on by the DeSCENT mechanical team or developed in parallel with the current design.

Design

System Design

Based on the requirements outlined in the previous section, the design shown in Fig. 4 was initially proposed. The system consists of a LoRa radio module for ChipSat communication, some storage element with read and write functionality to both log ChipSat packets and access them afterwards, and a GPS for real time and location data. The system also requires a central control unit, most likely a microcontroller, for handling peripheral components. An additional block was added to depict data transmit for remote data access functionality. This would include some method of data transmit and a more powerful compute element such as a Raspberry Pi with greater built in capabilities such as multithreading, wireless communication, and more I/O options. However, as the scope of the project was narrowed to exclude remote data access for now, the data transmit block was omitted, as shown in Fig. 5.

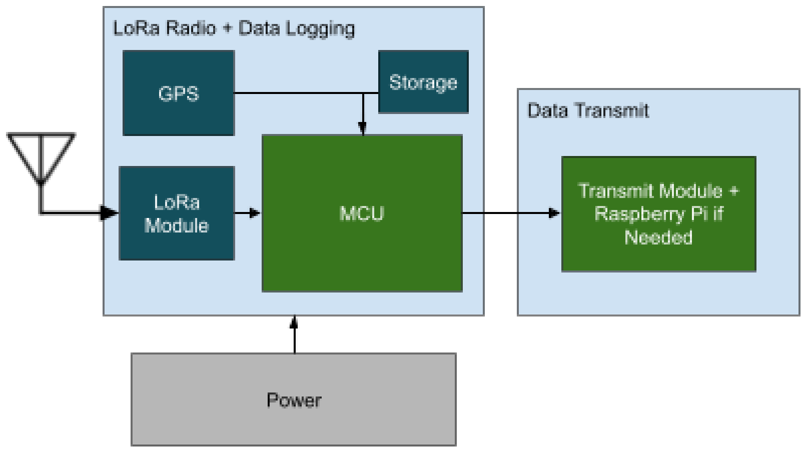

LoRa Radio + Data Logging
GPS
Storage
LoRa Module
MCU
Data Transmit
Transmit Module + Raspberry Pi if Needed
Power

Fig. 4: Ground Station Block Diagram with Data Transmit Block

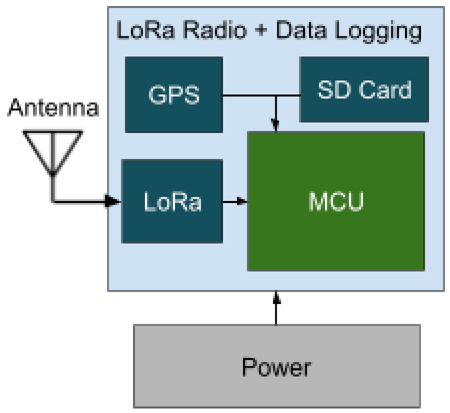


Fig. 5: Ground Station Block Diagram without Data Transmit Block

Component Selection

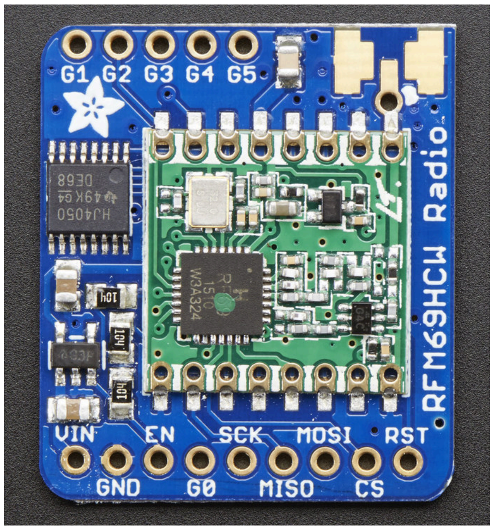
A Raspberry Pi Pico 1 was selected as the microcontroller for the design, and it was programmed using the Pico SDK. The key factors that motivated this choice were flexibility, comprehensive documentation, and availability. This setup, compared to microcontrollers programmed with Arduino IDE, allows for additional library configurability, and the Pico itself has multiple cores. Another benefit of using the Pico is that it is very well documented, and extra support and resources would be available via the expertise of my advisor, Dr. Adams. Lastly, the Pico is both a cheap yet relatively powerful component, and was readily available. However, a downside of this setup is that the library management must be done manually, and is a bit less "plug and play" compared to Arduino. Additionally, since the Pico SDK is meant specifically for the Raspberry Pi Pico, additional work is needed to port this code to other microcontrollers. Despite these drawbacks, the Pico was still determined to be the current most suitable option.

Another alternative that was considered, was to use a Raspberry Pi. This would be beneficial due to additional built in functionality and increased multitasking capability. However, for the basic ground station design, a Pi exceeds our functional needs and a Pico microcontroller was opted for instead due to its simplicity, lower cost, and lower power consumption. However, in the future, if a tracking antenna or remote data access are implemented, it may help to have the extra computing power of a Pi. As such, it was delegated to a separate module, as shown in Fig. 4. If remote data access is desired later down the line, the system can be expanded by adding on a Raspberry Pi in addition to the pico, or the entire system can be ported to the Pi instead.

Next, Adafruit's RFM95W LoRa radio module was selected for the design. This module provides a breakout board for the corresponding HopeRF component based on the SX1276 LoRa transceiver chip for ease of breadboard use. A similar HopeRF LoRa module for 433 MHz instead of 915MHz was previously used for Alpha ChipSat, making it a familiar and well documented option. For programming, the Radiolib universal wireless communication library, as it was previously used for Alpha as well, is well documented, and has sample code for the Pico. The module itself communicates over SPI, with a couple additional software defined I/O pins for IRQ handling and GPIO configuration. It operates on 3.3V power and logic, making it compatible with the Pico. Adafruit components are open source and well documented as well, making them easy to work with and adapt to a custom PCB later on. In addition to the LoRa module itself, the system also requires an antenna to enable radio communication. Unlike previous ChipSat missions that operate in LEO, DeSCENT will reach a suborbital apogee of approximately 100km, resulting in a significantly shorter transmit distance. Due to this, we have a much more flexible link budget than the Alpha ChipSat mission, and combined with a lack of size and form factor constraints, allows for a simplified antenna selection process. Its pinout is shown below in Fig. 6.

Next, a MicroSD card was used to store received ChipSat data. This method was chosen since it allows for convenient data retrieval. To extract mission data, one can simply remove the

SD card from the ground station and extract the data from it onto a laptop to be processed. While SD cards themselves are prone to failure over time as read/write cycles accumulate, for the purposes of the DeSCENT mission this is fine: the mission duration is relatively short, and the SD cards themselves are easy and cheap to replace. The size of the SD card also needs to be spec'd based on the packet size, transmission frequency, and number of ChipSats. This is discussed in more detail in the Packet Format section. Adafruit's MicroSD card breakout board was selected for this purpose. It also communicates via SPI, operates on 3.3V power and logic, making it compatible with the Pico, and again the same open source advantages apply here. Sample code from Bruce Land was referenced to write ChipSat data to a text file in the form of a CSV, making it easy to import into other data analysis software after the mission. Its pinout is shown below in Fig. 7 [9].



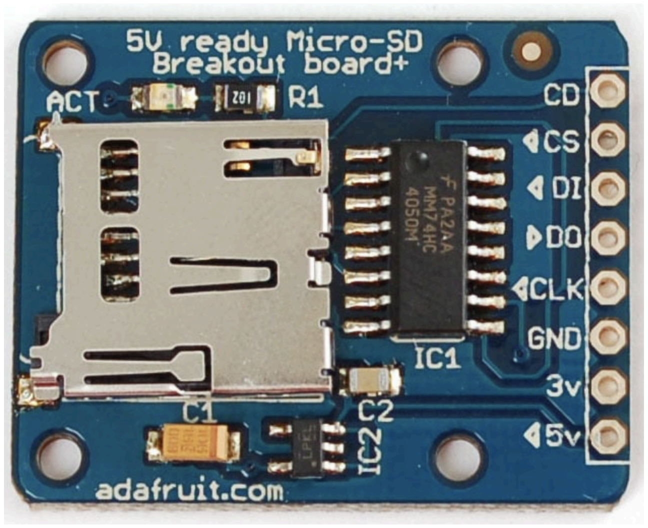


Fig. 6 (Left): RFM95 LoRa Breakout Pinout, Fig. 7 (Right): Adafruit MicroSD Breakout Pinout [5, 7]

Lastly, a GPS was also selected, not only to provide location data but more importantly to provide real time. This can add valuable context to the received ChipSat data, such as allowing for comparison between the last recorded time on the ChipSat with the last recorded time on the ground station. While this functionality could be implemented using just a real time clock, the additional location information may be useful even if the ground station itself is not moving, especially if a tracking antenna is implemented. An Adafruit Ultimate GPS breakout was selected for this purpose. It is both 3.3V and 5V power compatible, operates on 3.3V logic, making it compatible with the Pico, and it communicates over UART, allowing it to be separate from the other components communicating over SPI. It also comes with a patch antenna preinstalled. Its pinout is shown below in Fig. 8.

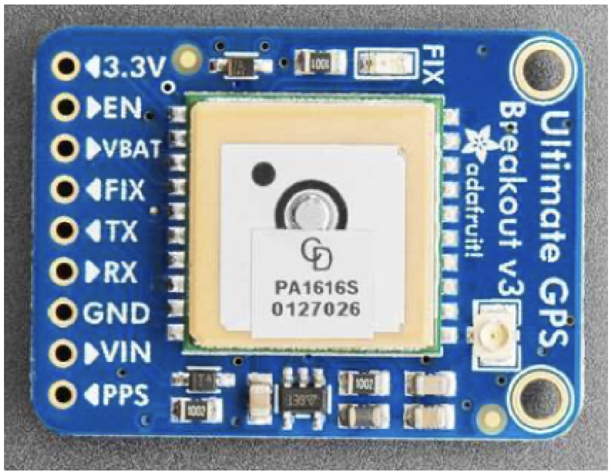


Fig. 8: Adafruit Ultimate GPS Pinout [6]

Hardware Implementation

Component	**Description**	**Quantity**
Raspberry Pi Pico	Microcontroller	1
Adafruit RFM95W LoRa Radio Transceiver Breakout	915 MHz LoRa radio, based on the SX1276 chip	1
Adafruit MicroSD Card Breakout Board	SD card reader/writer, for data logging	1
Adafruit Ultimate GPS Breakout	GPS, 66 channel w/10 Hz updates, MTK3339 chipset	1
Edge-Launch SMA Connector	For 1.6mm / 0.062" Thick PCBs, to connect antenna to LoRa Module	1
Antenna	915MHz, with SMA connector, for LoRa	1
47uF Capacitor	For power smoothing/decoupling	1

Fig. 9: Tentative Bill of Materials

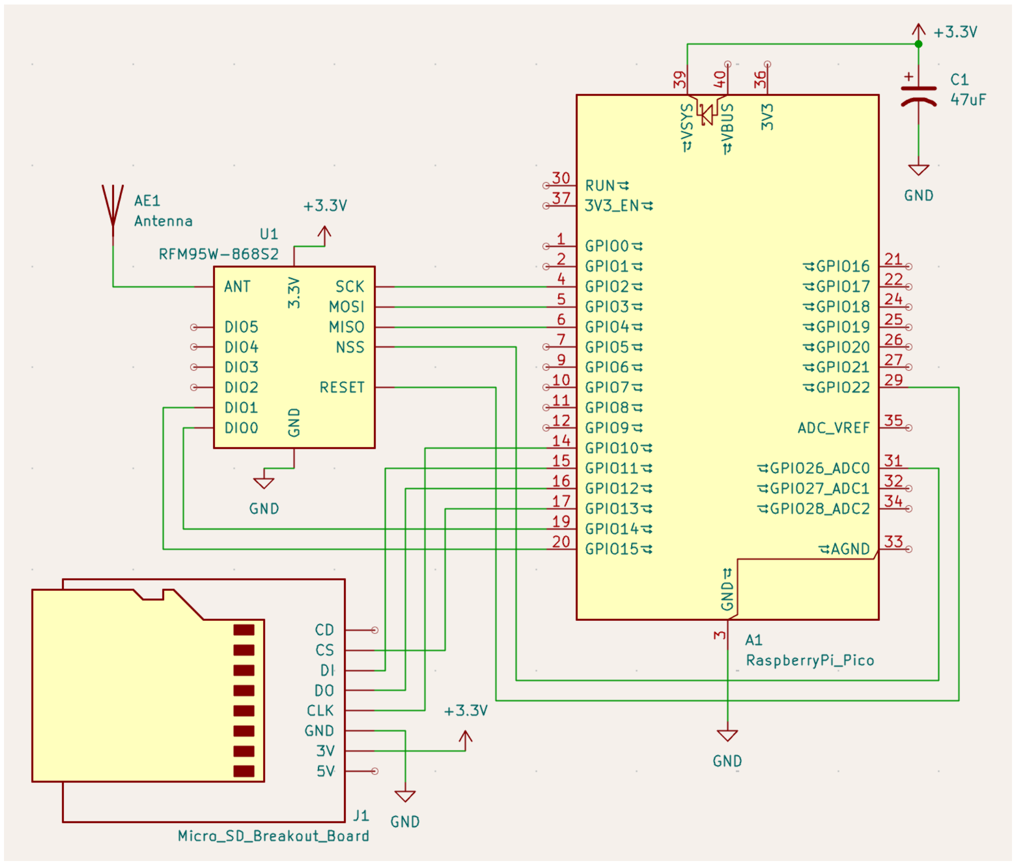
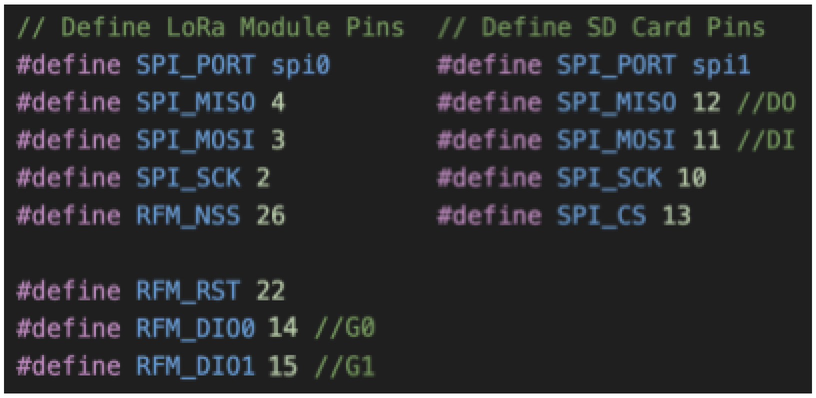


Fig. 10: Schematic of Current Design, GPS yet to be implemented

When powering the system, if it is powered via the Pico then its 3V3 pin should be connected to the 3.3V power line. Otherwise, if powering with an external power source, connect the Pico's VSYS line to the 3.3V power line, as shown in Fig. 10. DO NOT connect the pico 3V3 pin when powering with an external supply.


```
// Define LoRa Module Pins     // Define SD Card Pins
#define SPI_PORT spi0          #define SPI_PORT spi1
#define SPI_MISO 4             #define SPI_MISO 12 //DO
#define SPI_MOSI 3             #define SPI_MOSI 11 //DI
#define SPI_SCK 2              #define SPI_SCK 10
#define RFM_NSS 26             #define SPI_CS 13

#define RFM_RST 22
#define RFM_DIO0 14 //G0
#define RFM_DIO1 15 //G1
```

Fig. 11: Pin Assignments in Fig. 10

Software Implementation

For the software, first an initial LoRa transmit test was implemented. This was adapted from the Pico example in the Radiolib library and simply transmits an arbitrary string [3]. The goal of this was to verify that the Radiolib LoRa library and the LoRa module used are actually compatible with the Pico, and it took some time to configure the Cmake file to include the library correctly. To actually confirm that the module was transmitting, a USB software defined radio (SDR) was used to confirm there was a signal at 915MHz. Once this was working, the script was modified to receive instead of transmit. The core loop was changed such that instead of transmitting a string every few seconds, it instead starts listening and enters a waiting loop until it receives a packet. Once a packet is received, it triggers a callback function that sets a waiting flag to true, causing the code to exit the waiting loop. The code then reads contents of the received packet from the LoRa module into a buffer, where it can then be handled accordingly. Additionally, the LoRa signal parameters (detailed in the Problem Statement and Requirements) need to match the transmitter side in order for the received packet to be decoded properly, otherwise the result will be gibberish. This script was tested using a ChipSat programmed with a functional test script to transmit using the default LoRa parameters.

Once the LoRa module was working, an SD card test was implemented to simply write an arbitrary string to a text file based on sample code from Bruce Land [9]. For the purposes of this mission, SD card read functionality is not needed for the ground station since it won't need to interpret any of the received packet data after it is logged, though this functionality can be implemented. Once this was verified to be working by confirming that a string was written to a text file on an SD card, both the LoRa script and the SD card script were combined. The final code structure is detailed in Fig. 12. The main script, receive.cpp first initializes both hardware modules and their SPI interfaces. A module object is defined for the LoRa radio, the SD card is mounted, and header is appended to the text file on the SD card to act as labels for the CSV packet data. Then the code enters its main loop: the same as before, it first starts receiving and enters a waiting loop as it listens for packets. When a packet is received by the LoRa module, it triggers a callback function that sets a flag to true, signaling to exit the waiting loop. It then reads the packet contents into a buffer (must be larger than the packet), and parses out each sensor value. Then, it concatenates these values into a string, with each delimited by a comma, which it appends in a new line to the text file on the SD card. This format is done to make it easy to import the data as a CSV for processing after the mission. The loop then returns to the top and starts listening for packets again.

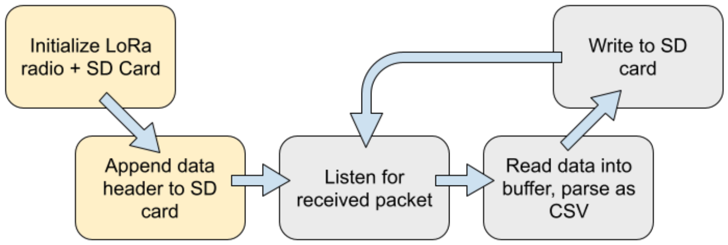

Initialize LoRa radio + SD Card
Append data header to SD card
Listen for received packet
Read data into buffer, parse as CSV
Write to SD card

Fig. 12: Flow Chart Demonstrating Code Functionality

Packet Format

Two packet formats were proposed for the mission, shown in Fig. 13. So far, strings of labelled values have been used for the packets for ease of parsing and testing, but for the actual mission it is ideal to switch to using bits with the labels removed. This results in a smaller packet size, allowing for more conservative bandwidth use and transmit time, which may help when coordinating multiple ChipSats transmitting at the same time. The mission packet format is still being finalized, so the ground station was designed to be easily modified to be compatible with either format, or other similar formats.

Option 1: Total size 61B							
ChipID	GPS	Gyro	Accel	Mag	Temp	Hum	Press
1B	4Bx3	4Bx3	4Bx3	4Bx3	4B	4B	4B
Option 2: 37B							
ChipID	GPS	Gyro	Accel	Mag	Temp	Hum	Press
1B	4Bx3	2Bx3	2Bx3	2Bx3	2B	2B	2B

Fig. 13: Two Proposed Packet Formats, table is in bytes, values are GPS XYZ, gyroscope XYZ, accelerometer XYZ, magnetometer XYZ, temperature, humidity, and pressure

Results

Debugging

The following section details some of the challenges and errors that were encountered during the process of designing and implementing the ground station:

1. Error code -2: RADIOLIB_ERR_CHIP_NOT_FOUND.

   This error occurred when trying to test out a simple transmit script for the LoRa Module, specifically when initializing the component, and the Radiolib documentation points to a few different possible causes [8]. One possible issue is that the chip type of the module instantiated in the code does not match the actual component. I verified that both the Adafruit breakout version uses the SX1276 chip, and that this matches the version defined in the code. In case of faulty documentation, I tried defining the module as a few other similar chips as well, such as the SX1278 and the SX126x chipset, but the issue still persisted.

   The next possible cause could be due to incorrect SPI pin definitions. To check for this, first, the pinout was checked, then the MOSI, MISO, CS, and CLK lines were probed using an oscilloscope. The behavior on the Pico side seemed correct: the CS pin was pulled low to select the corresponding device, and the MOSI and CLK lines showed activity while CS was pulled low. However, there was no response from the LoRa module on the MISO line, indicating further issues. Another check that was done was to verify that the LoRa module was even functioning in the first place. To do this, it was connected to a different microcontroller, specifically an Arduino Nano, and it was programmed with a test script via Arduino IDE. This setup was confirmed to be working using an SDR listening on 915MHz, meaning that the issue is not with the LoRa module itself.

   A potential issue not mentioned in the Radiolib documentation was that the Pico could not supply enough current to the LoRa module. Thus, the LoRa module was powered via an external power supply separate from the Pico to test this. While the issue still persisted, the external power supply continued to be used to prevent other potential related power issues. In later tests, it was found that the Pico can supply enough power to the LoRa module when receiving but not when transmitting, meaning that an external power supply was not required. However, ultimately an external power supply will need to be spec'd to power the entire system, which is detailed in the Results: Next Steps section. Ultimately, the actual fix for this problem was much simpler: the Pico pin numbering was interpreted incorrectly, and the CS pin was not actually connected. As a result, while the pinout was checked multiple times, this error was missed. However, in the process of debugging this issue, I gained a deeper understanding of the Radiolib LoRa library.

2. SD card f_mount error: "The physical drive cannot work".

   An issue that was encountered when testing the SD card breakout was an error when mounting the SD card. This seemed to be due to an issue with the SPI communication issue as well. Ultimately, the issue was due to having the unpowered LoRa module connected on the same SPI bus while testing the SD card, likely causing electrical interference with the logic values. Using different pins such that this was not the case resolved the issue. As an additional check, the SD card was also reformatted to ensure that it is FatFS, the intended format of the library. Other considerations when working with the SD card were ensuring that the SD card write parameters are configured to append to the end of the text file rather than overwriting it (via FA_OPEN_APPEND), and when to mount/unmount the SD card [2]. Initially, the SD card was mounted and unmounted with every write to avoid potential issues with not unmounting when code quits unexpectedly (eg. when pico is powered off). However to prevent excessive wear on the SD card this was changed to simply mount when the code is initialized and never unmount.

3. Unexpected register value errors when integrating LoRa and SD card.

   An issue that was encountered when integrating the LoRa and SD card test scripts came when combining their SPI buses: there were multiple errors indicating that unexpected register values were being read, or that the modules were unresponsive. The cause of this seems to be an issue with the CS configuration, most likely that the wrong device is being selected. This could be a potential pin or SPI definition issue, or it could be a deeper electrical issue. It is also distinct from the previous f_mount and radio module not found errors, since both modules were able to successfully initialize. However, for the sake of time the quickest solution was to simply use a different SPI channel. The Pico has 2 SPI channels available, so this was not an issue, but in the future if other SPI devices are added then it may need to be fixed. However, based on the current scope of the project, this should be fine, since the remaining module, the GPS, uses UART. spi0 was used for LoRa, and spi1 was used for the SD card.

4. Error code -7: RADIOLIB_ERR_CRC_MISMATCH.

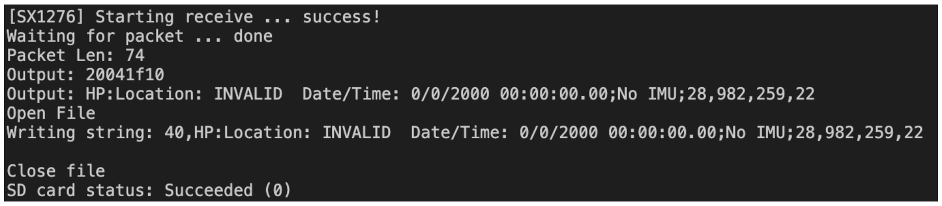
   When testing new LoRa signal parameters to optimize ChipSat transmit distance, there was an issue found with packet decoding for specific settings. A high spreading factor and low bandwidth serve to spread symbols chirps out over a longer duration and lower the noise floor, effectively increasing receiver sensitivity, allowing for a better signal over a longer distance. However, when testing with the ground station setup with our target bandwidth of 62.5 kHz, we were not able to use a spreading factor greater than 9 without getting a CRC mismatch error. This is much lower than the theoretical limit of the device, which is 12. The default and target LoRa signal parameters are shown in Fig. 14. With a higher spreading factor, the bandwidth must be increased to 500kHz for successful packet decoding, which is significantly higher than the default. Fig. 15 and 16 depict what successfully received and unsuccessfully received packets look like.

Based on Fig. 16, the RSSI (signal strength) and SNR (signal to noise) values are also nonsensical: typical RSSI is between -30 and -140 dB, while typical SNR values may be between -20 and +10 dB. This indicates that the problem here extends beyond a CRC mismatch, and likely may be some issue with bits getting shifted. Because shifted bits would disrupt the decoding of packet contents, hence the invalid characters, as a result this would also cause the CRC bits to be incorrect. The first few characters of the unsuccessfully received packet being correct also seems to corroborate this (eg. “HP:L” in Fig. 15 and 16). The decoding of the received packet is successfully getting through the preamble and a few characters before it gets too desynced, resulting in the invalid characters that follow. This also confirms that the LoRa module is still able to hear and recognize packets, but is unable to decode them since data is still being read. One possible cause of this could be a hardware timing issue: the new parameters result in a much longer airtime, meaning that the receiver may be progressing ahead of the transmitter. A slower Pico clock and SPI speed was tested, though this did not resolve the issue. This could also be the result of a register writing issue, though this would require deeper digging to be done in the future. It was also confirmed that this was not an issue with the transmit signal itself since the same LoRa parameters were successfully tested with an ESP32 based receiver.

Going forward there are a few different options to consider in order to address this issue. The best option would be to find a fix, but this requires further exploration that I did not have time to get to this semester. If it is determined that the problem is caused by an incompatibility issue with the Pico, another option would be to change the mission LoRa parameters. This would necessitate an evaluation of the ChipSat transmit link budget and communications requirements, and possibly additional hardware or system changes. Lastly, a third option if the issue is incompatibility would be to port the system and code to a different microcontroller. Radiolib has support for a wide variety of microcontrollers including the Arduino family, ESP32 and Raspberry Pi, and the SD card library FatFS is platform independent, only requiring a working SPI channel [2]. Implementing a lower coding rate was also discussed. This parameter determines how often a CRC check is conducted, and while this won’t fix the issue above, it will reduce the constraints for what counts as a successfully received packet. Even if values may be missing, some data is better than no data.

	Freq	BW	SF	CR	SW	PL	Power	CRC	LDRO
Default	915MHz	125 kHz	9	4/7	N/A	8	10 dB	N/A	N/A
High Power	915MHz	62.5 kHz	12	4/7	Custom Value	8	22 dB	On	On
Low Power	915MHz	62.5 kHz	11	4/7	Custom Value	8	14 dB	On	On

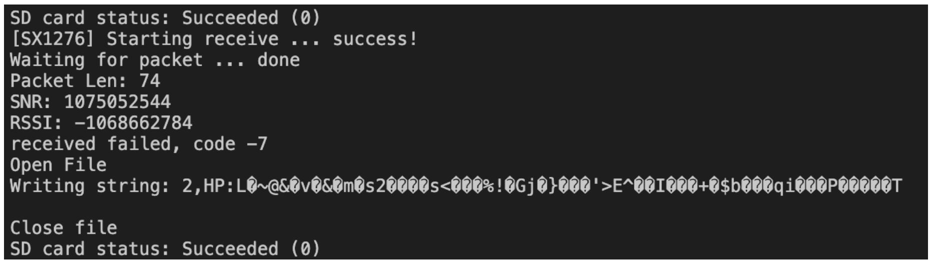
Fig 14: Default vs. New LoRa Signal Parameters. LDRO abbreviation for low data rate optimization and CRC abbreviation for cyclic redundancy check. See problem statement and requirements 1. for other abbreviations. Power does not matter on the receiver side.



```
[SX1276] Starting receive ... success!
Waiting for packet ... done
Packet Len: 74
Output: 20041f10
Output: HP:Location: INVALID  Date/Time: 0/0/2000 00:00:00.00;No IMU;28,982,259,22
Open File
Writing string: 40,HP:Location: INVALID  Date/Time: 0/0/2000 00:00:00.00;No IMU;28,982,259,22

Close file
SD card status: Succeeded (0)
```

Fig. 15: Successfully Received Packet (Default Parameters)



```
SD card status: Succeeded (0)
[SX1276] Starting receive ... success!
Waiting for packet ... done
Packet Len: 74
SNR: 1075052544
RSSI: -1068662784
received failed, code -7
Open File
Writing string: 2,HP:L�~@&�v�&�m�s2����s<���%!�Gj�}���'>E^��I���+�$b���qi���P�����T

Close file
SD card status: Succeeded (0)
```

Fig. 16: Unsuccessfully Received Packet (High Power, SF=12)

Next Steps

Due to the time limitation on this project being one semester instead of two, there are still a few components outlined in the system that need to be implemented. These are as follows:

1. Implement GPS.

   The data provided by a GPS module would still be useful for the mission due to the reasons outlined previously. As a result, even though a GPS was selected, it was not yet tested with and integrated into the system. This would be the next step for continuing the project. Since no actual integration has begun yet, a different GPS can be selected if problems arise. In addition to integrating the GPS, something else that needs to be determined is how often to update the GPS data, which will affect power consumption, and the accuracy and time to get a lock should be tested.

2. Implement power system.

   The power requirements of the system were also estimated using the parameters outlined in Fig. 17. Based on the analysis, a 1000mAh to 1500mAh LiPo battery (typically 3.7V) would provide sufficient power with a safe surplus while also allowing for a standalone system. Without GPS, the power consumption was mostly constant, with current hovering around 50mA. This value was measured by observing the current consumption while powering the entire system with a power supply. With a GPS, this current jumps up to 70mA, assuming an update frequency of once a second as detailed by its datasheet [6]. This frequency can also be lowered depending on power requirements and ChipSat packet frequency. Some other considerations include the maximum pulse current needed to support the system, for example if there are current spikes when it powers on, and providing voltage regulation or battery charging/monitoring.

Duration: setup, mission duration, retrieval	Voltage:	Current: No GPS	Current: With GPS	Estimated Total Power	Capacity Needed
6-12 hrs	3.3V	50mA	+ 20mA	70mA*3.3V= **231mW**	231mW*12hrs= **277mWh**

Fig. 17: Power Consumption Estimates

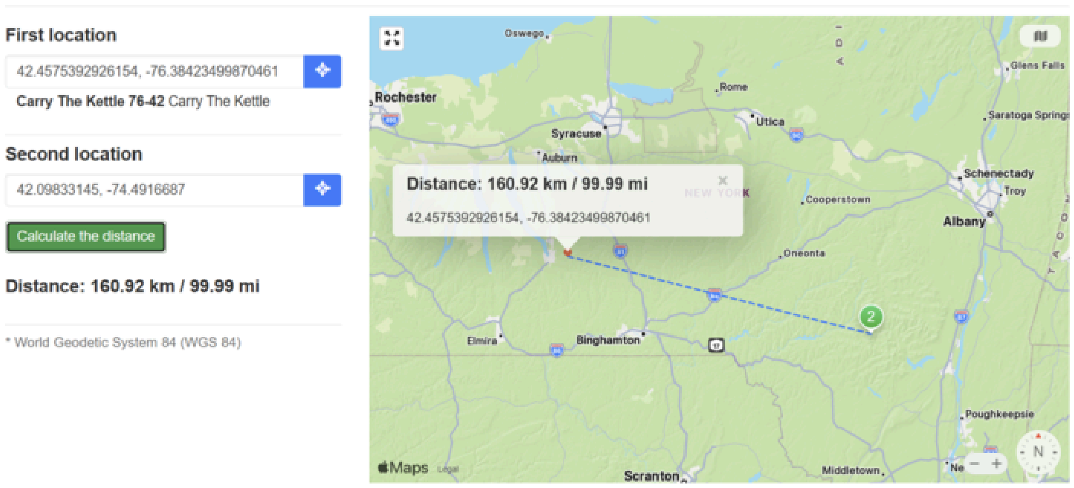
3. Board and mechanical design.

   For a more robust system that is less prone to connection issues, a PCB can be designed in place of the breadboard used. This can be as simple as just connecting the power system and different breakout boards used. A consideration might be to add header pins or slots for the breakouts rather than soldering them directly to the board, for ease of unit testing or replacing components. Ideally this should not be manufactured until the electrical system is finalized. Another related aspect to implement is a mechanical design for the chassis of the ground station. This can be very simple, though there are some factors to keep in mind: temperature may be a concern since the mission will take place in a hot sunny environment, and ease of access to components such as power connectors, programming connectors, and the SD card slot.

4. Antenna selection and range testing.

   A suitable antenna still needs to be chosen for LoRa. Because the mission has a suborbital launch, there is a shorter transmit distance, allowing for a more flexible link budget compared to previous missions. Some range testing has been done previously, including a shorter distance test using attenuators while listening across different parts of campus, and a weather balloon launch over a significantly longer distance. The furthest packet was received at a distance of 160km with a maximum RSSI of -130 dBm and an

SNR of -30 dB (see Fig. 18 and 19, SNR not shown). However, improperly decoded packets get more frequent with distance, as indicated by increasing CRC errors. Another aspect to consider when selecting an antenna is its radiation pattern. We do not know exactly where the ChipSats will be relative to the ground station, though the ground station itself can also be strategically placed to accommodate the antennas as well. Lastly, like the rest of the ground station, the antenna should ideally also require minimal setup.


First location

42.4575392926154, -76.38423499870461

Carry The Kettle 76-42 Carry The Kettle

Second location

42.09833145, -74.4916687

Calculate the distance

Distance: 160.92 km / 99.99 mi

* World Geodetic System 84 (WGS 84)

Distance: 160.92 km / 99.99 mi

42.4575392926154, -76.38423499870461

Fig. 18: Maximum Received Packet Distance from Dec. 11th, 2025 Balloon Test

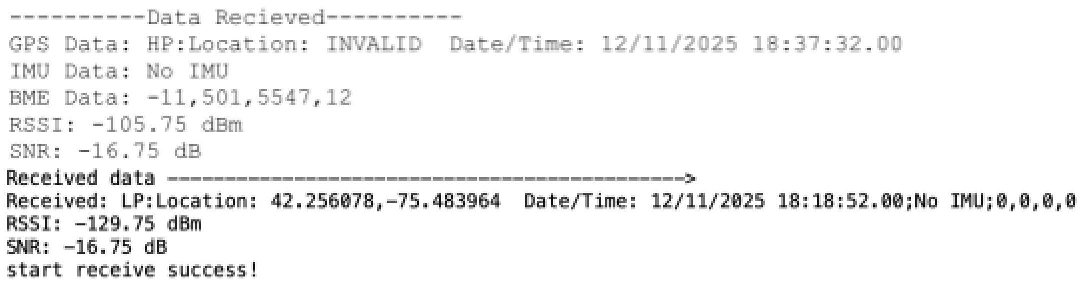

```
----------Data Recieved----------
GPS Data: HP:Location: INVALID  Date/Time: 12/11/2025 18:37:32.00
IMU Data: No IMU
BME Data: -11,501,5547,12
RSSI: -105.75 dBm
SNR: -16.75 dB
Received data ---------------------------------------------->
Received: LP:Location: 42.256078,-75.483964  Date/Time: 12/11/2025 18:18:52.00;No IMU;0,0,0,0
RSSI: -129.75 dBm
SNR: -16.75 dB
start receive success!
```

Fig. 19: Packet Data for Fig. 18 for High Power (HP) and Low Power (LP) Signal Parameters (shown in Fig. 14)

Conclusion

Progress Made

To summarize the progress made on this project are as follows. Due to this project taking place over the course of one semester instead of the anticipated two, not all system elements were implemented in time. However, a solid base was developed that can easily be built upon in the future.

1. The ground station requirements were outlined based on the DeSCENT mission goals, and a system was designed based on these requirements. Components were selected for the core functional aspects of the system, though a more detailed power system still needs to be implemented.
2. The LoRa module and the storage mechanism were implemented and tested successfully. These two components were then integrated into a system that can successfully receive ChipSat packets and save them to a text file on the SD card. An electrical system was built using a breadboard, and code was written for its functionality. Additional testing to scope the packet format and power requirements was conducted as well.
3. A working code framework was implemented that can be easily expanded on in the future. Documentation on how to use the system and modify it was completed as well.

Future Work

In addition to the next steps section, there is exploration to be done for some of the additional features discussed in the Problem Statement and Requirements section. For remote data access, this could include researching deeper into ways to implement a method for transmitting ground station data. Based on initial research, a potentially promising solution would be to implement a LoRaWAN network, which can be done using Raspberry Pis. In addition to this, a web dashboard could be designed for convenient data access. Overall, this entails a more software and networking heavy project. Additionally, some initial exploration of a tracking antenna was done this semester by the DeSCENT mechanical team, which can continue in parallel with development on the current design.

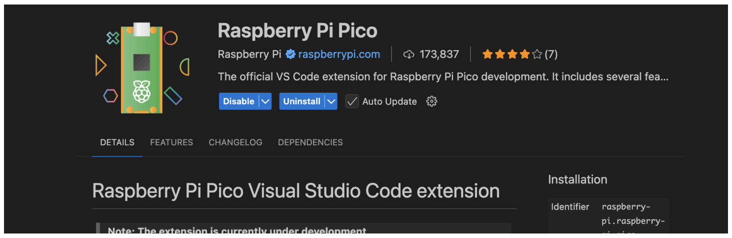
References

1. Adams, V. H. (n.d.). *ECE 4760/5730, Cornell University¶*. ece4760. https://ece4760.github.io/
2. *FATFs - generic fat filesystem module*. FatFs - Generic FAT Filesystem Module. (n.d.). https://elm-chan.org/fsw/ff/
3. Goddard, C., & Gromes, J. (2024). *RadioLib/examples/nonarduino/pico/main.cpp at master · JGROMES/RadioLib*. GitHub. https://github.com/jgromes/RadioLib/blob/master/examples/NonArduino/Pico/main.cpp
4. Hurford, M., Peters, B., Umansky-Castro, J., & Heywood, C. (2025). ChipSat Design for the DeSCENT Mission. *Small Satellite Conference*. https://digitalcommons.usu.edu/smallsat/2025/RA-S2-2025/6/
5. Industries, A. (n.d.-a). *Adafruit RFM95W Lora Radio transceiver breakout - 868 or 915 MHz*. Adafruit Industries . https://www.adafruit.com/product/3072
    - Additional link: https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts/overview
6. Industries, A. (n.d.-b). *Adafruit Ultimate GPS breakout - 66 channel W/10 hz updates*. Adafruit Industries. https://www.adafruit.com/product/746
    - Additional link: https://cdn-shop.adafruit.com/product-files/746/CD+PA1616S+Datasheet.v03.pdf
7. Industries, A. (n.d.-c). *MicroSD card Breakout Board+*. Adafruit Industries. https://www.adafruit.com/product/254
8. Gromes, J. (n.d.). *JGROMES/RadioLib: Universal Wireless Communication Library for embedded devices*. GitHub. https://github.com/jgromes/RadioLib/tree/master
    - Additional Link: https://jgromes.github.io/RadioLib/index.html
9. Land, B. (2025, August 12). *Cornell University ECE4760 microSD card reader Pi Pico rp2040 and rp2350*. RP2040 SD card. https://people.ece.cornell.edu/land/courses/ece4760/pi_pico/sd_card_reader_pico2/index_sd_card.html
10. Lohatepanont, M., Umansky-Castro, J., Zheng, R., & Chadwick, A. (2025). (tech.). *DESCENT Engineering Requirements Document (ERD)*. Ithaca, NY.
11. *Pico-series microcontrollers - raspberry pi documentation*. Raspberry Pi. (n.d.). https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html

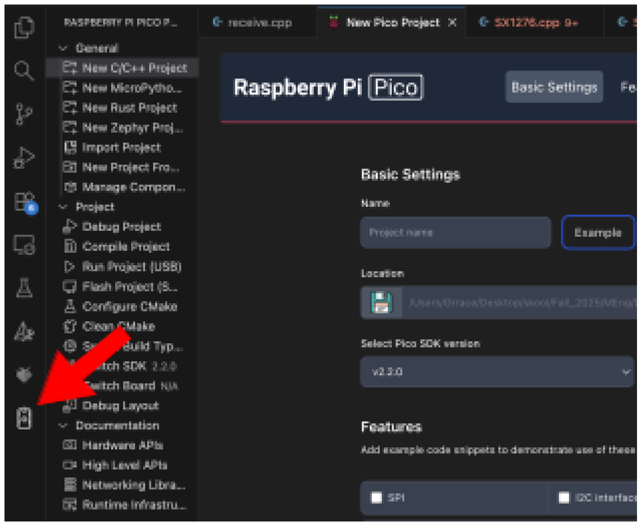
Appendix - User Manual

Using the Pico SDK

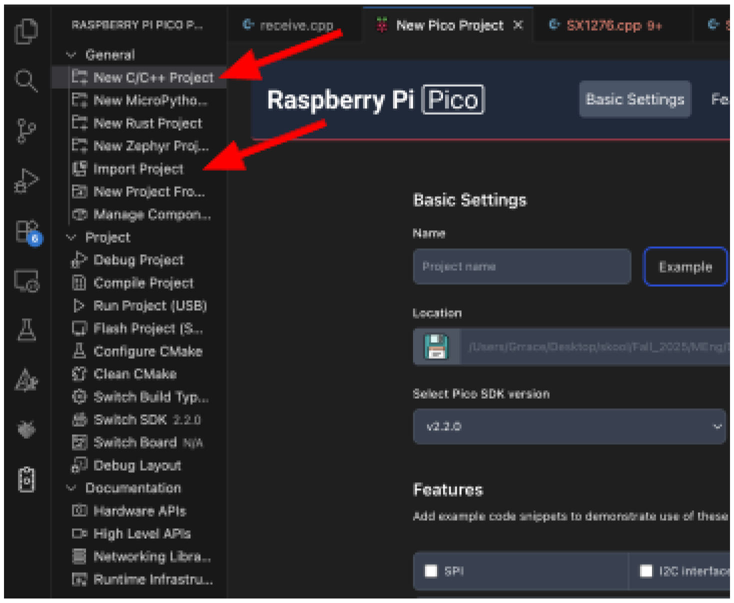
This section is meant to be a guide on using the Pico for potential students continuing this project, some of whom may mainly have experience working with Arduino. To set up the SDK, first download the Raspberry Pi Pico VSCode Extension:


Raspberry Pi Pico
Raspberry Pi raspberrypi.com 173,837 (7)
The official VS Code extension for Raspberry Pi Pico development. It includes several fea...
Disable
Uninstall
Auto Update
DETAILS
FEATURES
CHANGELOG
DEPENDENCIES
Raspberry Pi Pico Visual Studio Code extension
Installation
Identifier raspberry-pi.raspberry-
Note: The extension is currently under development

You can then access it from the sidebar in a VSCode window:



To create a new C/C++ project select the *New C/C++ Project* option from the sidebar. This will open a new tab in your main VSCode window area. Name your project, select its location, and configure any other settings as needed. The Pico SDK includes options to include sample code for common applications such as a number of different communication interfaces. Once these are set, you can then scroll to the bottom and select the *Create* button. This will generate a new directory for your project with a *CMakeLists.txt* file and C/C++ script with your main function. Alternatively, if you want to open an existing project or import one, you can select the *Import Project* option from the sidebar instead. Select the directory your project is located in and and select the *Import* button.



Basic Settings

Name: Project name

Location: /Users/Grace/Desktop/skool/Fall_2025/MEng/

Select Pico SDK version: v2.2.0

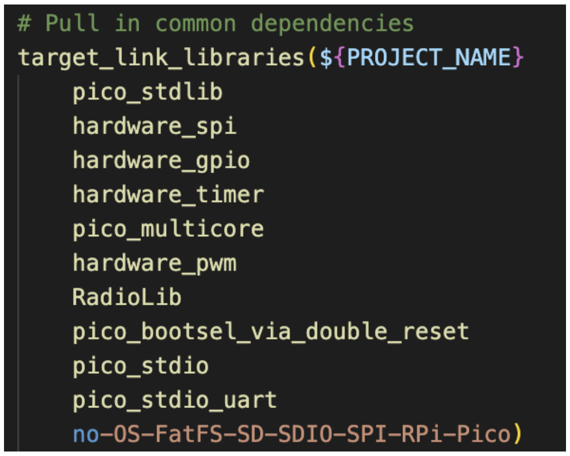
Features

Add example code snippets to demonstrate use of these

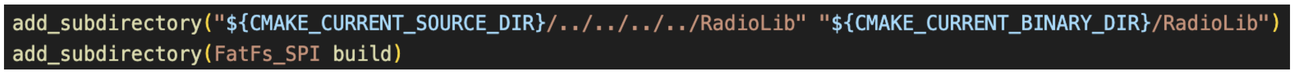
The Pico SDK also has options for using Micropython, and some other options to explore. For more details and possible errors see the documentation [1, 11].

CMake Basics

To include additional libraries, they need to be manually added to the *CMakeLists.txt* file. If adding one of the default Pico libraries, simply add it to the *target_link_libraries* command. If adding a more specific library, such as FatFS or Radiolib, you must first add the subdirectory with a path to the source folder for the library. This can be done using *add_subdirectory(source_dir [binary_dir])*.

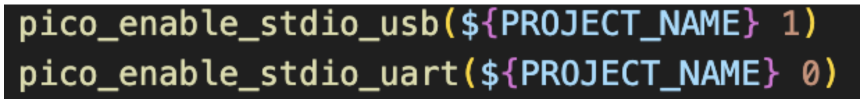


```
# Pull in common dependencies
target_link_libraries(${PROJECT_NAME}
    pico_stdlib
    hardware_spi
    hardware_gpio
    hardware_timer
    pico_multicore
    hardware_pwm
    RadioLib
    pico_bootsel_via_double_reset
    pico_stdio
    pico_stdio_uart
    no-OS-FatFS-SD-SDIO-SPI-RPi-Pico)
```



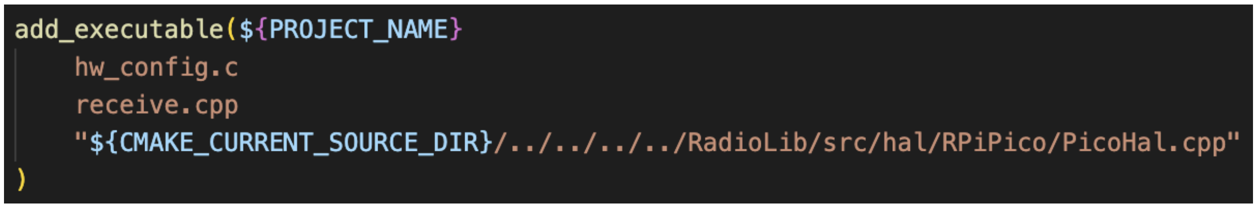
```
add_subdirectory("${CMAKE_CURRENT_SOURCE_DIR}/../../../../RadioLib" "${CMAKE_CURRENT_BINARY_DIR}/RadioLib")
add_subdirectory(FatFs_SPI build)
```

To enable use of the serial monitor, ensure that the following settings are configured correctly via the following commands. For example, if using the serial monitor via USB, then the corresponding USB command should be set to 1.



```
pico_enable_stdio_usb(${PROJECT_NAME} 1)
pico_enable_stdio_uart(${PROJECT_NAME} 0)
```

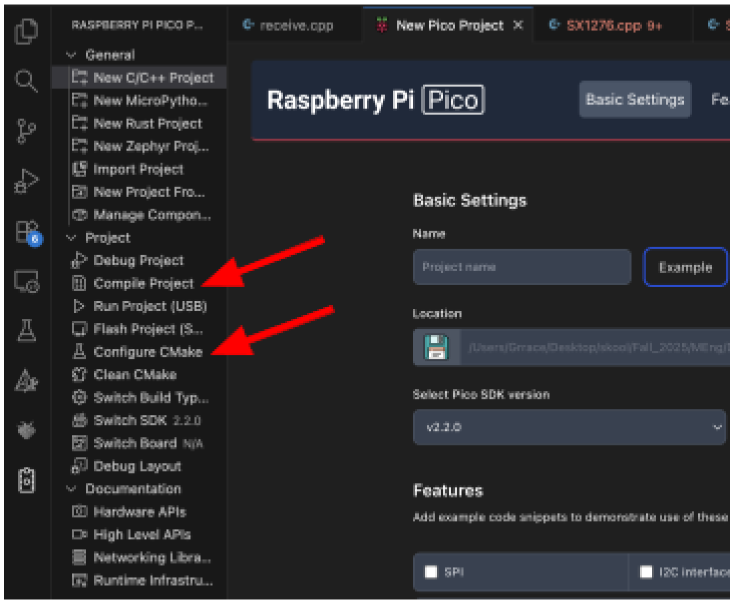
Additionally, you can configure CMake to run multiple scripts by modifying the following. The listed files after *${PROJECT_NAME}* are scripts to be run.



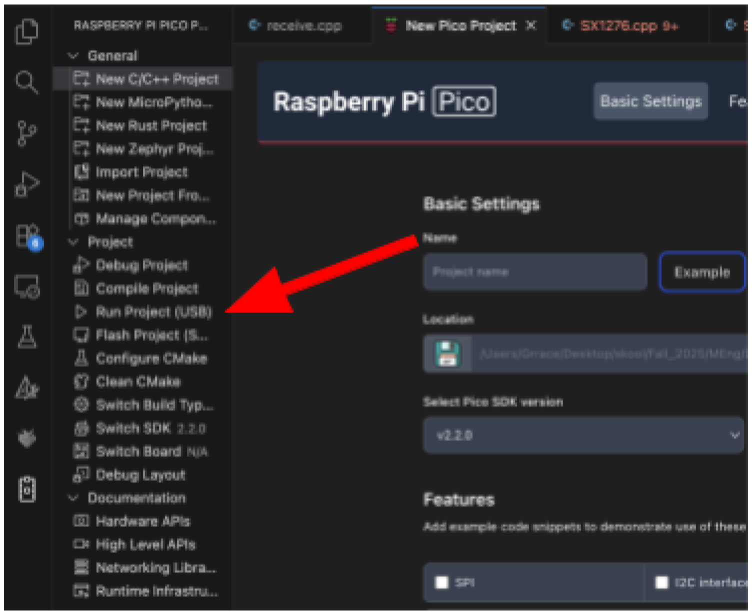
```
add_executable(${PROJECT_NAME}
    hw_config.c
    receive.cpp
    "${CMAKE_CURRENT_SOURCE_DIR}/../../../../RadioLib/src/hal/RPiPico/PicoHal.cpp"
)
```

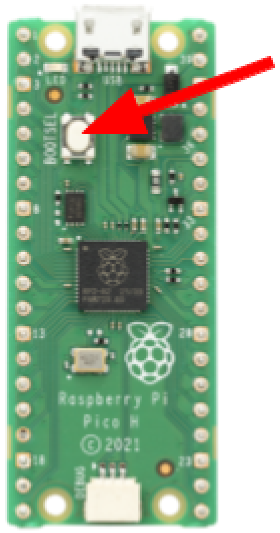
How to Program the Pico

Make sure your CMake configures without error and your code compiles before attempting to program the Pico. This can be done using the following options on the sidebar.

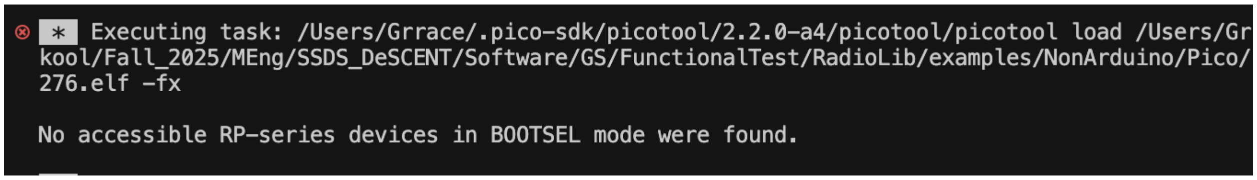


Once configured and compiled, you can start programming the Pico. When connecting the Pico to your laptop, hold down the white *BOOTSEL* button. This puts the Pico into boot select mode, allowing it to be programmed via USB. Once this is done, you can select the *Run Project (USB)* option from the sidebar to flash your code.





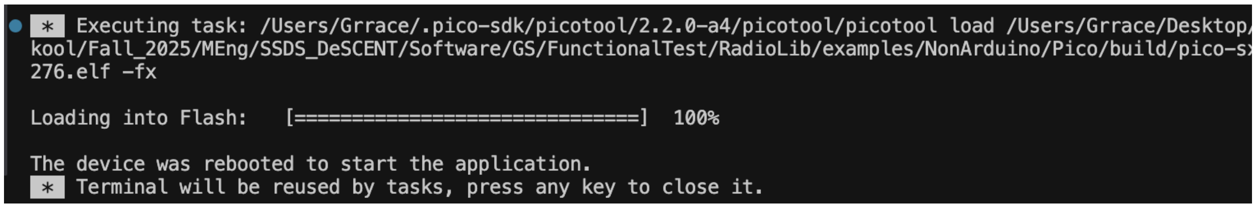
If the Pico was not set to boot select mode, or is not detected, the following error will occur. In this case, try reconnecting the Pico with the button held down, or checking its connection to the laptop.



```
⊗  * Executing task: /Users/Grrace/.pico-sdk/picotool/2.2.0-a4/picotool/picotool load /Users/Gr
 kool/Fall_2025/MEng/SSDS_DeSCENT/Software/GS/FunctionalTest/RadioLib/examples/NonArduino/Pico/
 276.elf -fx

 No accessible RP-series devices in BOOTSEL mode were found.
```

If successfully programmed, the terminal output should look like this.

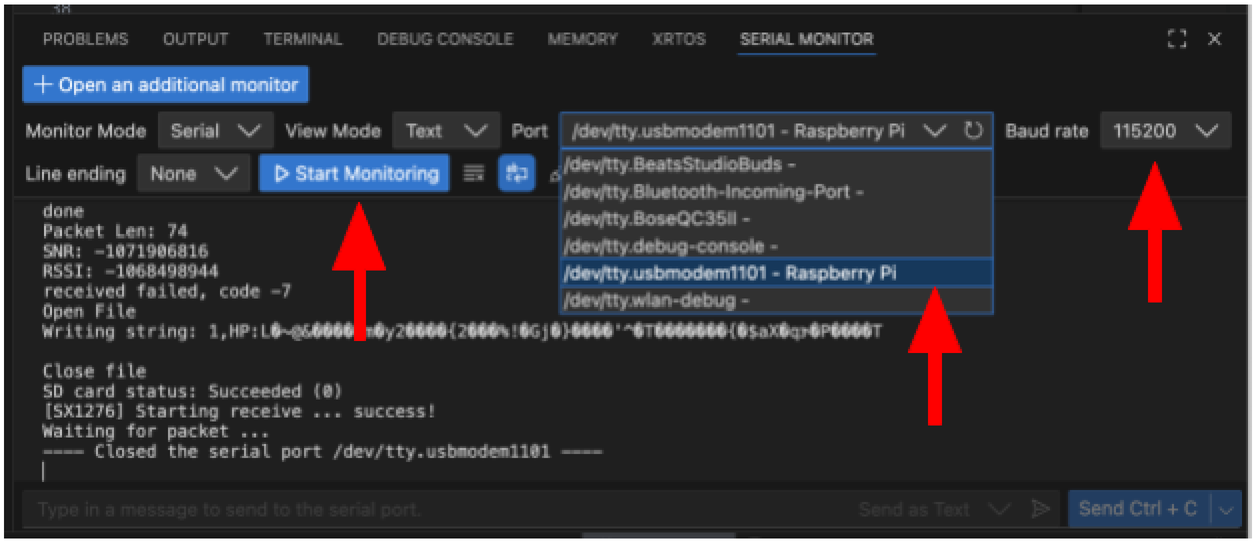


```
● * Executing task: /Users/Grrace/.pico-sdk/picotool/2.2.0-a4/picotool/picotool load /Users/Grrace/Desktop/
  kool/Fall_2025/MEng/SSDS_DeSCENT/Software/GS/FunctionalTest/RadioLib/examples/NonArduino/Pico/build/pico-s
  276.elf -fx

  Loading into Flash:   [==============================]  100%

  The device was rebooted to start the application.
  * Terminal will be reused by tasks, press any key to close it.
```

To access the Pico’s serial monitor, first open a serial monitor in VSCode. Then select the port corresponding to the Pico. This is usually labelled, but if it isn’t you can check by connecting and disconnecting the Pico, or checking your device manager. Press the start monitoring button to start displaying serial monitor output. If the port is not showing up, or failing to open, check that your code is not terminating, or running into an error that causes it to quit. Make sure to check that you are using the correct baud rate. If the serial monitor is not printing anything, it is possible that printed outputs occurred before the monitor was opened. Because of this, it can be useful to repeat error and status messages in a loop.



Monitor Mode: Serial
View Mode: Text
Port: /dev/tty.usbmodem1101 - Raspberry Pi
Baud rate: 115200
Line ending: None

```
/dev/tty.BeatsStudioBuds -
/dev/tty.Bluetooth-Incoming-Port -
/dev/tty.BoseQC35II -
/dev/tty.debug-console -
/dev/tty.usbmodem1101 - Raspberry Pi
/dev/tty.wlan-debug -
```

```
done
Packet Len: 74
SNR: -1071906816
RSSI: -1068498944
received failed, code -7
Open File
Writing string: 1,HP:L0~@&0000[illegible]0y20000(2000%!0Gj0)0000'^0T0000000(0saX0qr0P0000T

Close file
SD card status: Succeeded (0)
[SX1276] Starting receive ... success!
Waiting for packet ...
---- Closed the serial port /dev/tty.usbmodem1101 ----
```

Type in a message to send to the serial port.

Code

Main function: receive.cpp

```
// Default Pico + C libraries
#include <pico/stdlib.h>
#include <stdio.h>
#include <format>
#include <vector>
#include <string>
#include "hardware/clocks.h"

// SD card libraries
#include "f_util.h"
#include "ff.h"
#include "my_rtc.h"
#include "hw_config.h"

// LoRa libraries
#include <RadioLib.h>
#include "hal/RPiPico/PicoHal.h"

// Define LoRa Module Pins (see hw_config.c for SD card pins)
#define SPI_PORT spi0
#define SPI_MISO 4
#define SPI_MOSI 3
#define SPI_SCK 2
#define RFM_NSS 26 //CS
```

```
#define RFM_RST 22
#define RFM_DIO0 14 //G0
#define RFM_DIO1 15 //G1

// Create a new instance of the HAL class
PicoHal* hal = new PicoHal(SPI_PORT, SPI_MISO, SPI_MOSI, SPI_SCK);
// Create radio module using hal
RFM95 radio = new Module(hal, RFM_NSS, RFM_DIO0, RFM_RST, RFM_DIO1);
// Create new SD card object
FATFS fs;

// Signal Parameters
float freq = 915;
float bw = 125; // Bandwidth
int sf = 9; // Spreading factor
int cr = 7; // Coding rate
int sw = RADIOLIB_SX126X_SYNC_WORD_PRIVATE;
int pwr = 10; // Doesn't matter on receive end
int pl = 8; // Preamble length
int gn = 0; // Gain, doesn't matter in receive end
int bufferlen = 100; // Buffer size needs to be greater than packet size
int packetlen;
volatile bool receivedFlag = false;

// LED initialisation, for built in Pico LED
int pico_led_init(void) {
#if defined(PICO_DEFAULT_LED_PIN)
   // A device like Pico that uses a GPIO for the LED will define PICO_DEFAULT_LED_PIN
   // so we can use normal GPIO functionality to turn the led on and off
   gpio_init(PICO_DEFAULT_LED_PIN);
   gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
   return PICO_OK;
#elif defined(CYW43_WL_GPIO_LED_PIN)
   // For Pico W devices we need to initialise the driver etc
   return cyw43_arch_init();
#endif
}
// Turn the led on or off
void pico_set_led(bool led_on) {
#if defined(PICO_DEFAULT_LED_PIN)
   // Just set the GPIO on or off
   gpio_put(PICO_DEFAULT_LED_PIN, led_on);
#elif defined(CYW43_WL_GPIO_LED_PIN)
   // Ask the wifi "driver" to set the GPIO on or off
   cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, led_on);
#endif
}

// Write string to file in SD card
FRESULT write_data(char* str){
 // Open file
 printf("Open File\n");
 FIL fil;
 const char* const filename = "logdata.txt";
 FRESULT fr = f_open(&fil, filename, FA_OPEN_ALWAYS | FA_OPEN_APPEND | FA_WRITE);
 // FA_OPEN_ALWAYS - creates file if it doesn't exist
 // FA_OPEN_APPEND - read/write set to end of file, prevents overwriting existing
contents
 if (FR_OK != fr && FR_EXIST != fr){
   printf("f_open(%s) error: %s (%d)\n", filename, FRESULT_str(fr));
   return fr;
 }

 // Write string
 printf("Writing string: %s\n",str);
 if (f_printf(&fil, str) < 0) {
       printf("f_printf failed\n");
 }

 // Close file
 printf("Close file\n");
 fr = f_close(&fil);
 if (FR_OK != fr) {
   printf("f_close error: %s (%d)\n", FRESULT_str(fr), fr);
   return fr;
 }

 return fr;
}

// LoRa packet received callback function
void setFlag(void){
 receivedFlag = true;
}
```

```
int main() {
 // Initialize pico
 stdio_init_all();
 sleep_ms(100);

 // Initialize LED
 int rc = pico_led_init();
 hard_assert(rc == PICO_OK);
 pico_set_led(true);
  // Initialize radio with parameters set above
 printf("[SX1276] Initializing ... ");
 int state = radio.begin(freq,bw,sf,cr,sw,pwr,pl,gn);
 if (state != RADIOLIB_ERR_NONE) {
   printf("initialization failed, code %d\n", state);
   pico_set_led(false);
   while(1){
     printf("initialization failed, code %d\n", state);
     sleep_ms(2000);
   }
 }
 // Set LoRa receive callback function
 radio.setPacketReceivedAction(setFlag);
 printf("[SX1276] init success!\n");

 // Initialize SD Card Writer
 if (!sd_init_driver()) {
   while (true){
     printf("SD init driver failed\n");
     sleep_ms(1000);
   }
 }

 // Mount SD Card
 printf("Mount SD Card\n");
 FRESULT fr = f_mount(&fs,"0:",1);
 if (FR_OK != fr){
   pico_set_led(false);
   printf("f_mount error: %s (%d)\n", FRESULT_str(fr),fr);
   return fr;
 }
```

```
// Write header to SD card
printf("Writing header to SD card\n");
char header[] = "PLACEHOLDER HEADER\n";
// char header[] =
"PacketNum,ChipID,GPSlat,GPSlong,GPSalt,IMUgyroX,IMUgyroY,IMUgyroZ,IMUaccelX,IMUaccelY
,IMUaccelZ,IMUmagX,IMUmagY,IMUmagZ,Temp,Humidity,Pressure\n";
FRESULT sd_status = write_data(header);
printf("SD card status: %s (%d)\n", FRESULT_str(sd_status), sd_status);

// Count received packets
int packetnum = 0;

// Loop forever
for(;;) {
  // Start listening
  receivedFlag = false;
  printf("[SX1276] Starting receive ... ");
  int state = radio.startReceive();
  if (state == RADIOLIB_ERR_NONE) {;
      printf("success!\n");
  } else {
      printf("failed, code %d\n", state);
  }

  // Wait for packet to arrive
  printf("Waiting for packet ... ");
  while(!receivedFlag){
    sleep_ms(1);
  }
  printf("done\n");
  // Count received packets
  packetnum = packetnum + 1;

  // Print packet stats
  printf("Packet Len: %d\n", radio.getPacketLength());
  printf("SNR: %d\n", radio.getSNR());
  printf("RSSI: %d\n", radio.getRSSI());

  // Read packet data into buffer
  uint8_t str[bufferlen] = {0};
  int state1 = radio.readData(str,bufferlen);
  str[radio.getPacketLength()] = 0;
```

```
    if (state1 == RADIOLIB_ERR_NONE) {
      printf("Output: %x\n", str);
      printf("Output: %s\n", str);
    } else {
      printf("received failed, code %d\n", state1);
    }

    // Parse packet data (as CSV), format as char array
    char buf[bufferlen];
    std::snprintf(buf, sizeof(buf), "%d,%s\n", packetnum, str);
    std::string str_formatted(buf);
    // TODO: implement bit parsing, concatenate as comma separated values

    // Write to SD card
    sd_status = write_data(buf);
    printf("SD card status: %s (%d)\n", FRESULT_str(sd_status), sd_status);

    sleep_ms(1000);
  }
}
```