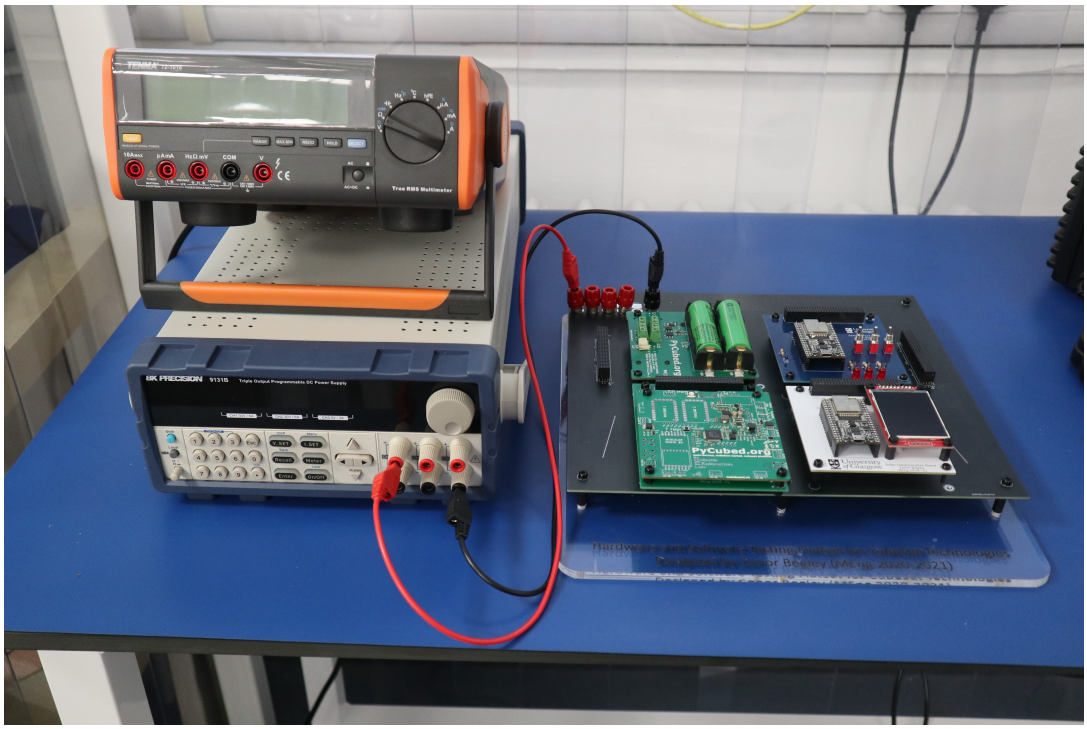
MEng Individual Project ENG5041P

A Hardware and Software Emulation Test Bed for CubeSat Design



MEng in Electronic and Software Engineering

Author: Conor Begley
Student Number: 2236693B
First Supervisor: Dr. Gilles Bailet
Second Supervisor: Prof. Marc Sorel

Integrated Space and Exploration Technologies Department
University of Glasgow
February 22, 2021

University of Glasgow

Abstract

CubeSats are praised for reducing the cost of access to space and increasing opportunities for universities and small organisations to carry out space related research. In principal, a small team with an innovative solution could rapidly develop and fly hardware in space using already existing 'building block' components. However, they still require technical electronic knowledge for the construction and testing of designs. While technical knowledge will always be required, especially on custom built CubeSats, this project aimed to reduce this barrier of entry, by providing a test bed framework to program and experiment with potential CubeSat designs.

The project has two main aims: the construction of physical devices for use in testing and developing accompanying software to help use these devices. The main physical device is a FlatSat, which aims to be generic and suitable for a range of projects. It uses PC/104 Plus connectors and has access to power and diagnostic connections. Alongside the FlatSat, there is a mainboard and battery board from an open source CubeSat design, which can be used to test other CubeSat subsystems.

The project software helps schedule tasks to be run using the mainboard and FlatSat system. These tasks then act as emulations of an entire CubeSat subsystem. This allows for a subsystem to be tested, as if an entire CubeSat had already been constructed. The framework for programming and using this software, strives to be simple enough for those inexperienced with programming to be able to be use, while still giving enough complexity that most aspects of a CubeSat can be emulated.

To demonstrate the usability of the system, two demonstration boards have been constructed. These boards provide example emulations of payload and communication boards. They also provide a template that can be modified to construct and emulate other subsystems. A demonstration task schedule has also been created to show how devices can interact with the software and mainboard from this project.

COVID-19 Constraints

14th January 2020

This main body of work for this project took place from the 24th of August 2020 to the 16th of December 2020. This was during the COVID-19 pandemic and multiple government lock-downs in Scotland. As a result not all practical project work intended was carried out. However, I appreciate all the effort Dr Bailet, Prof. McInnes, the Integrated Space and Exploration Technologies department and the university went to, to ensure it was possible to carry out some practical work. I am very grateful to have gotten that opportunity.

Lab access was limited to a total of two day sessions in November and four day sessions in December. In this time, the PyCubed Battery board and PyCubed mainboard was assembled but the FlatSat and the demonstration boards were not. The PyCubed mainboard, while soldered could not be tested before the closing of the lab, due to one missing component. Alongside this, errors in the FlatSat and demonstration boards that appeared in the first iteration of design, were spotted late, only when in the lab in December. These errors were fixed in the schematics and PCB designs for these boards. The PCB boards were reordered alongside the missing component, however, they did not arrive in time to complete assembly.

Both of these setbacks would have been mitigated by continuous lab access, allowing for faster development and multiple iterations of physical designs built in the lab. This very limited lab access meant the entire project could not be constructed by the report deadline. Work was carried out on breadboards and components in my residence in Glasgow. This allowed for the software of this project to be developed. However, despite virtual designs being made, the physical components produced for this project are severely lacking as a result of the impact COVID-19 had on this project.

Dr Bailet has agreed to allow the assembly of remaining components not done by the report deadline (15th January 2021) to be done before the final project presentation(*some time between 4th and 25th of February 2021*). However, with UK travel restrictions currently in place and another lockdown currently active in Scotland, this work may not be completed by the final presentations. However any documentation of any assembly done after the report deadline will be available on the Project GitHub(`https://github.com/C-Begley/masters_project`), should more information on a more completed version of this project be required.

Acknowledgements

Dr Gilles Bailet: For all guidance, advice and hands on help throughout this project.

Dr Kevin Worell and Dr James Beeley: For review of my electronic designs.

Prof. Colin McInnes and the Integrated Space and Exploration Technologies Department: For providing me with the resources I needed during the project.

Barbra Grant: For handling all my part orders and purchase requests.

Veronique Dekkers: For proofreading and being there for me.

My parents: For supporting me throughout.

Acronyms

ADCS Altitude Determination and Control subsystem.

CAN Controller Area Network.
CLK Clock.
COTS Commercial Off The Shelf.

ECSS European Cooperation for Space Standardisation.
EPS Electrical Power subsystem.
ESA European Space Agency.

GPIO General Purpose Input Output.
GPS Global Positioning System.

I^{2}C Inter-Integrated Circuit.
IAC Image Acquisition Unit.
IMU Inertial Measurement Units.
ISET Integrated Space and Exploration Technologies Department.

JSON JavaScript Object Notation.

MISO Master In Slave Out.
MOSI Master Out Slave In.
MRAM Magnetic Random Access Memory.

OBC On-Board Computer.
OPEN Open Prototype for Educational NanoSats.

REPL Read–Eval–Print Loop.
RMS Rate Monotonic Scheduling.
RTOS Real Time Operating System.
RX Receive Line.

SATNOGS Satellite Networked Open Ground Station.
SCL Serial Clock Line.
SDA Serial Data line.
SPI Serial Peripheral Interface.
SU Scientific Unit.

TX Transmission Line.

UART Universal Asynchronous Receiver/Transmitter.
UHF Ultra High Frequency.
UNICLOGS University Class Open Ground Station.

VHF Very High Frequency.

Contents

List of Figures

List of Tables

1 Introduction

CubeSats are a form of nanosatellite designed to reduce the cost of entry into space exploration and to make space research more accessible. A CubeSat is defined as a satellite of certain dimensions. This size is broken down into a unit measure, where 1U stands for a dimensions of 10cm x 10cm x10cm. A CubeSat can then be a multiple of this unit size e.g. 1U, 1.5U, 3U, 16U *etc*. CubeSats are typically included as auxiliary payloads on public and private sector missions to space, allowing them to be launched for significantly cheaper than a launch specifically for a CubeSat[1].

In recent years, as technology has become cheaper and more accessible, CubeSats have been used by private companies, universities, schools and even hobbyists. For the University of Glasgow, CubeSats are a strong focus of the Integrated Space and Exploration Technologies Department (ISET), with many PhD students and researchers working on this topic. This project aims to assist in this research and to design components of a CubeSat test bed for use in the department. The main intention is provide a system that can be used to design parts of a CubeSat and emulate other parts of a CubeSat system. This would allow for potential prototype designs to be tested in the early stage of development, without designing or needing an entire CubeSat system.

The first stage of this project focuses on providing a test bed for possible future designs by the ISET department. Currently a common industry practice is to design an entire CubeSat and test all of the components together [2]. This can be difficult, especially if a member of the ISET is researching or designing just a specific component for a CubeSat. To experiment and verify the design, this component would require a test bed, other CubeSat components and software programs. With this project, the aim is to produce a high level emulation of a command subsystem that can be used to test payloads, inter-board communication and subsystems of future CubeSat designs.

In creating this test bed, the project provides both a hardware and software solution. The final design provides a physical platform, where the user does not need to manage a series of connections and wires between prototypes devices. To complement this, the physical system is accompanied by easy to use program that handles the emulation of any missing components not included in a prototype design. This allows for rapid development of individual aspects of the system, while not compromising on considerations for design interactions with the entire system.

A secondary focus is the creation of a modular CubeSat ecosystem, that could allow for rapid CubeSat design. The ISET department has its own experiments they wish to carry out and would only require changing the payload and scientific units of a CubeSat. From this, the idea is to produce an interchangeable component for the new design. CubeSats are not designed for re-entry, reducing the benefit of modularity, but a modular system would simplify future designs and changes. As it stands, most experiments using open source designs often require a custom redesign of an existing design. This project then aims to reduce this design time to just that of the experiment payload and interface.

All researchers may not have experience with Real Time Operating System (RTOS) or programming in general. The current ISET department has a range of team members from various of engineering and science backgrounds. Looking further afield, as CubeSats become more and more mainstream,

anyone from any background - academic, industrial or otherwise - may have an idea for a CubeSat. However, they would not necessarily have the experience or know-how to program it. This project aims to provide a system that can be, at a minimum, used for design exercises to introduce the user to the concepts of programming a CubeSat. It may also have the potential to be used as the basis for flight software in future designs. With this consideration, the final design is not largely reliant on a coding language or software. It is intended to provide input to configure the system using a format closer to English than a standard programming language. This will then reduce the learning curve needed to get into CubeSat design.

In trying to create this ecosystem, the system's main component should be suitable for space. If any micro-controller was used and the project the ecosystem is successful enough to be deemed suitable for in use in a given mission, considerations would then have to be made to ensure the design would work in space. It could then be found too late into the design time, that the given micro-controller is not suitable for space. For this reason and to give more value to the project, the mainboard of the emulation for this project has to be suitable for use in space.

Given the time constraints of this project, alongside the high risk of designing a CubeSat from scratch, the project is based on open source designs. A range of open source designs are available, and part of this project was deciding on which one to use. A project aim was to modify and build upon an open source design. This would ensure the main design is space compatible, while also speeding up prototyping and testing of the project.

Open source is a strong driving force in the success and future development of CubeSats[3]. Without some CubeSats being a completely open source project, this project would not be possible. This project will follow in these footsteps and produce all designs to be open source. Alongside open source, this project will strive to produce accessible documentation on the entire project. This includes a user manual for programming the test bed and using the project in a CubeSat design. A tertiary goal of this project is for the potential future use in teaching at the University of Glasgow. Having interfaces and the designs well documented will help with this.

Ultimately this project's aim is to create a 'PlugNPlay' test bed for future designs, to further promote open source development of CubeSats and to increase the accessibility of CubeSats.

2 Literary Review

2.1 Current State of CubeSats

The first CubeSat was designed as a joint project between California Polytechnic State University's Multidisciplinary Space Technology Laboratory and Stanford's Space Systems Development Laboratory in 1999. The purpose of this CubeSat was to teach students how to design and test space craft. From there, a CubeSat standard was drafted that served as guidelines for other universities to follow and the first CubeSat was launched successfully in 2003, on a Russian "Rockot" launch vehicle [4].

Fast forward seventeen years and the CubeSat industry has bloomed. As of January 2020, 1200 CubeSats have been launched, carrying out a vast array of scientific experiments[5]. There have now been successful CubeSats missions that have even left Earth's orbit and travelled to Mars [6]. The scope and market for CubeSats is massive. It has gone beyond a simple university idea to a full fledged multi-billion dollar industry, with companies like the Glasgow Clyde Space [7] at the forefront of development. It is no longer a just research project designed to teach students principals of space design.

2.2 Overview of CubeSat subsystems

Most CubeSat follows a common structure when it comes to component design. A CubeSat will usually consist of "cards" that slot into a CubeSat structure(*see fig. 1*). Generally each card carries out a separate function but this may not always be the case [8]. These cards can then be considered the subsystems of a CubeSat.

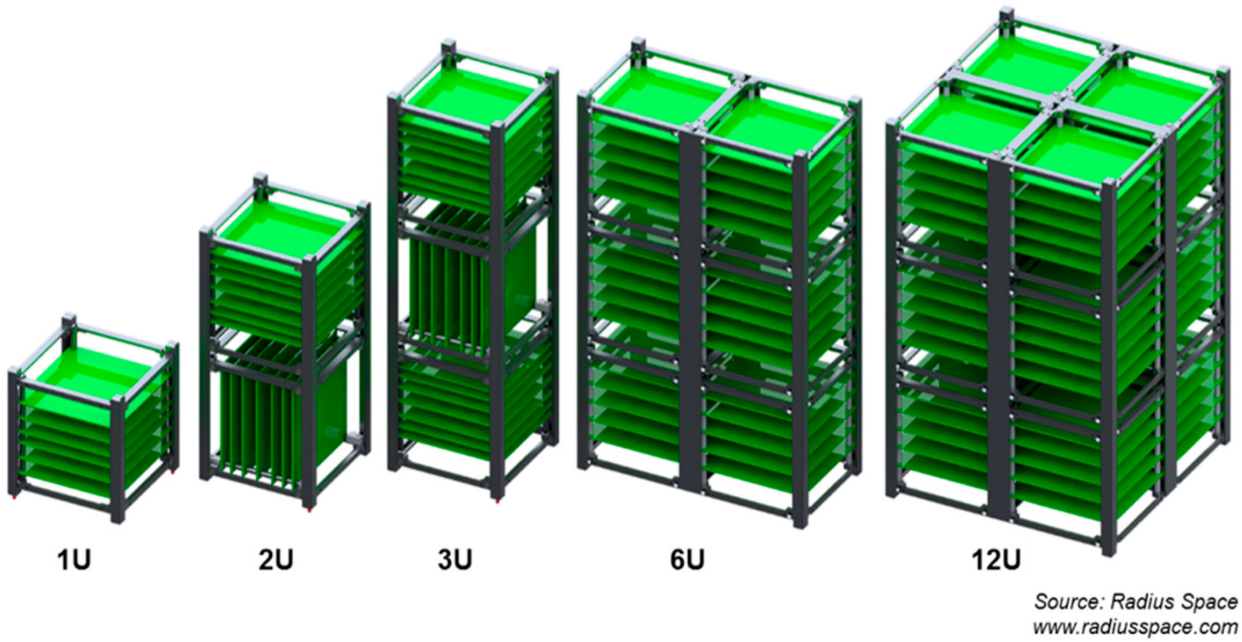


Figure 1: Sample of CubeSat Structures (*grey*) with Cards (*green*)

2.2.1 Command Subsystem

The command subsystem, often called the On-Board Computer (OBC), is the main controller of any CubeSat. The responsibly of an OBC varies depending on the autonomy of the other CubeSat subsystems and their complexity. Usually an OBC is responsible for the scheduling of system tasks, but this may be extended to power management, data logging and communication protocols, depending on the CubeSat design. As part of the CubeSat standard, antenna's have to be stowed before launch and the solar panels will normally be stowed also, to fit within the CubeSat dimensions. The command subsystem may be responsible for deploying these after launch [9].

2.2.2 Altitude Determination Control Subsystem

Most CubeSats will have some form of Altitude Determination and Control subsystem (ADCS). The complexity of the experiment, communication subsystem and sensors used directly influences the complexity of this subsystem. A CubeSat has various reasons for wanting to identify their orbit and adjust the direction relative to the Earth, the CubeSat is pointing. This can be for cameras, communication antennas and solar panel efficiency, among other things.

To identify the orientation, the CubeSat can use an array of sensors, with Inertial Measurement Units (IMU), star trackers and magnetometers being the most common ones. To control the orientation, reaction wheels and magnetorquer are the typically used as actuators, relying on the principle of conversation of angular momentum. When the actuator spins a direction, the satellite will spin the opposite direction to maintain angular momentum, allowing the orientation of the CubeSat to change [10].

2.2.3 Electrical Power Subsystem

Given the nature of CubeSats, they will need access to an Electrical Power subsystem (EPS), in order to keep their subsystems running. Early CubeSats used a non-rechargeable battery system but most modern CubeSats use solar arrays and batteries. CubeSats typically use Gallium Arsenide solar cells which have energy conversions of up to 30%. A low cost alternative is Silicon solar panels, but they come with a trade off in efficiency. The surface area of CubeSats are small and this limits the available energy output to a few watts. And so power consumption budgets must be made and adhered to.

On top of this CubeSats typically are exposed to the sun 66% of time on each orbit. As a result, there has to be a way to store power for the times the sun's energy is not accessible. Lithium Ion batteries are the favoured battery choice, benefiting from high energy density and no memory effect deterioration.

One final consideration is the CubeSat must be inert before launch, to prevent communication interference for safety of the launch vehicle and potential damage resulting from battery failure. It also ensures the CubeSat has power to at least deploy the solar panels and charge once launched. As a result, the OBC but more often the EPS must turn on the CubeSat after launch [9].

2.2.4 Scientific Unit

The Scientific Unit (SU) is often the primary purpose of a CubeSat mission. In the past, the primary mission of CubeSats was to build a CubeSat capable of launching to space or to demonstrate technology was usable in space. Nowadays, the focus of CubeSats has shifted to the possibility of carrying out experiments. These experiments can be anything, from a complex sensor array to a simple camera. Design considerations have to be made with a payload in mind. The complexity of the payload controls how complex the communication subsystems need to be: to send data back from the scientific unit, how much power needs to be generated by the EPS and what functions the command unit needs to be capable of carrying out [11].

2.2.5 Communication Subsystems

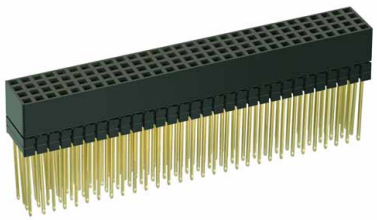
- **Ground Communication:**
  Ground Communications are expected when designing a CubeSat. Like most aspects of a CubeSat, they are not required by a CubeSat standard. However, launching a CubeSat without some form of communication renders a CubeSat's mission largely pointless. There would be no way to tell if it was successful or to retrieve data from the SU. Typically, CubeSats operate in the Very High Frequency (VHF) and Ultra High Frequency (UHF) ranges: 30 to 300 MHz and 300 MHz to 3 GHz respectively. This usually means a request to a governing body like the International Telecommunication Union needs to be made to operate a device at this frequency. The most common one appears to be in the range around 430-450 MHz. However, for cubesats requiring high data rates may use an S-band communication system in the range of 2.2 GHz [12] [13] [14].

- **On-Board Communication:**
  Normally, CubeSats will require communication between components. This is the case with the scientific unit and the ground station communication but there are many more examples. The EPS may shutdown components to conserve power or limit their usage. The ADCS may need to update the EPS when to start charging batteries or vice versa. The command card may schedule tasks for the SU, Communication or ADCS. Generally, the components are all interconnected in some way.

  For on-board communication, there are two considerations: the physical data bus and the communication protocol. Neither are defined in the CubeSat standard. By not limiting to any one method, this in part allows for a variety of Commercial Off The Shelf (COTS) products to be used in CubeSats. With that in mind, there is still some push towards an unofficial standardisation. The PC/104-style connectors have become the de facto industry standard, mainly for the connection style rather than the pin configuration. The PC/104 bus allows for stacking of cards to create a data bus from the pin headers, without the need for a backplane [15]. This can be seen in fig. 2. However, the PC/104 standard pinout is not conformed to within the CubeSat bus design. Most of these connections are unique to the individual CubeSat design, with the exception of power supplies and grounding connections [16].

(a) PC/104 Stack

Source: https://www.rtd.com/PC/104/default.htm



(b) Harwin PC/104 Connector

Source: https://www.harwin.com/products/M22-6003005/

Figure 2: PC/104 Connection

For communication protocols, Universal Asynchronous Receiver/Transmitter (UART) has been the most common protocol, in part due to it's simplicity, using just two wires and sending data asynchronously. SPI has also been used, albeit less frequently, due the initial 3-4 wire requirement, followed by an additional slave select line per device. However, in recent years there has been a push towards address based communication and some form of standardisation. One such example is Controller Area Network (CAN), promoted by organisations such as the European Cooperation for Space Standardisation (ECSS) [17]. CAN is a communication protocol used in the automotive industry. It has had various revisions and shown to work in safety critical systems like cars [18]. This reliability of CAN is one of the driving factors in introducing CAN into CubeSats. Another communication protocol that is being considered by groups like the European Space Agency (ESA), is Inter-Integrated Circuit (I^2C). I^2C also uses addresses, but uses masters and slaves, instead of the more communal node approach in CAN. I^2C does have the advantage of being adopted by many designs used in space already, and it's two wire simplicity over CAN, but is regarded as not being safe or reliable [16].

2.3 Current State of CubeSat Testing

An important part of CubeSat design is the use of testing. It can costs up $300,000 to launch a 3U CubeSat [2], so having the CubeSat fail once it is orbit, is not a desirable outcome. This is even before considering the wasted money and time on the design of the CubeSat on top of the launch cost. To prevent this, robust testing on Earth is carried out before launching.

2.3.1 Design Testing

To test the electronic design and debug both inter-card communication and ground communication, a prototype of the design is made. Often this takes the form of a FlatSat. FlatSats are a way of laying out the CubeSat in a single layer arrangement, instead of stacking them like in the CubeSat structure. This allows for easy access to signals and connections for debugging, alongside prototyping

using breadboards. This is often done ad hoc. A custom structure may be made, or breadboards and jumper wires can be used. It often requires a certain level of design to be achieved, so that inter board communication and interaction can be tested. Once the FlatSat has been proven to work, an engineering test unit is usually made which is the FlatSat converted to work in the CubeSat structure. This is used for final debugging and verification of the design in a CubeSat structure [2].

2.3.2 Radiation Testing

In order to verify designs for use in Space, considerations have to be made for sensitive electronics being exposed to solar and cosmic radiation. This radiation can disrupt and change charges on particles in electronic components which causes errors. Typically, this can lead to two main types of problems within electronics [19]:

- **Single Event Effects**: When a high-energy particle passes through a semiconductors, it can cause a localised ionisation. This ionisation can cause a range of errors, usually caused by a transistor latching due to the ionisation. Small errors caused in memory storage *etc* from this can usually be fixed.
- **Total Ionizing Dose**: This is a cumulative damage caused by exposure to radiation over a period of time. Depending on the component in question, it has a different effect, but ultimately, if continuously exposed, it is a irreversible problem.

However, CubeSats are relatively short lived and and are not exposed to much radiation. A year long mission experiences about 1.2krad, which is considered a benign amount of exposure. Often, design considerations leave out radiation protection and opt for Single Event Effect preventative measure like having a watchdog circuit. Some do not even include this and just opt to test the CubeSat using a radiation chamber or similar before launching [20].

2.4 Current State of Open Source CubeSats

For this project, given the time constraints, it would not be practical to design an entire CubeSat from scratch. Instead, initial designs had to be considered as a starting point for development. To date, there have been six open source CubeSats of note.

2.4.1 OpenOrbiter

OpenOrbiter is a project, created by the University of North Dakota, to demonstrate the Open Prototype for Educational NanoSats (OPEN), also created at the University of North Dakota. It seeks to validate and demonstrate the designs laid out in the OPEN design documents. It's secondary purpose is to build a sensing payload for the visible electromagnetic spectrum. The OBC in the original design was a Raspberry Pi board, but an additional processor, GumStix WaterStorm COM unit, was used for processing power required by the payload. OpenOrbiter appears to have never launched and is currently going under major redesigns. At the time of writing, these updated design documents are not available [21] [22].

2.4.2 ArduSat

ArduSat was an open source 1U CubeSat, funded by Kickstarter and was launched in November 2013. It re-entered the atmosphere and burnt up on re-entry in April 2014. The focus of ArduSat was to provide an Ardunio platform that was intended to allow people to conduct experiments in space. ArduSat was fitted with an array of sensors and a camera which could be accessed remotely from Earth. The ArduSat mainboard consisted of an ATmega2561 (a processor similar to Ardunio Mega) which was a supervisor of 16 ATmega3280s (processors similar to Ardunio Uno). These nodes could be programmed remotely. Ardusat used a half-duplex UHF transceiver, using the 435-438 MHz band to communicate with the ground stations. People could submit requests online with code they wished to run on the ArduSat. It would be reviewed and if approved, uploaded to the ArduSat, with the data gathered returned to the original requester. However, ultimately ArduSat was not very successful in carrying out this process. A second CubeSat, ArduSat-2 was launched, but has been understood to have failed. The makers of ArduSat have since formed a private company, Spire, and appear to have removed online designs for ArduSat [23].

2.4.3 UPSat

UPSat claims to be the first fully open source hardware and software satellite. This was designed by both the University of Patras and LibreSpace foundation. It was a 2U CubeSat and the primary purpose of UPSat was to use a Langmuir probe to sample electron density in Low Earth Orbit. There was a a secondary payload, using the Linux board DART-4460 and a camera that served as an Image Acquisition Unit (IAC). It successfully launched in April 2017 and re-entered atmosphere and burnt up in November 2018. The STM32 family of microprocessors formed the basis of the UPSat design. The STM32F4 was used as an OBC for the design. The STM32F4 was also used in the ADCS, SU and communications subsystems while STM32L1 was used in the EPS. The various subsystems use serial inter board communication, with the EPS monitoring “heartbeats” from each subsystem. A Global Positioning System (GPS) receiver, magnetometer, gyroscope and sun sensor were used for accurate altitude and orientation sensing, while reaction wheels and magnetorquers were used for control. Communication was done in the range of 402-470 MHz and used the Satellite Networked Open Ground Station (SATNOGS) framework for ground stations. The software for the entire system was written in C and used the FreeRTOS operating system [24] [25].

2.4.4 EQUiSat

Much like the original CubeSat, EQUiSat was a teaching exercise to build a 1U CubeSat designed by undergraduate students though this time at Brown University. EQUiSat’s other mission focus was to test a battery technology previously not used in space. As this was the focus, the payload was a rather simple light beacon (*visible on Earth*) and a ham radio signal. The satellite’s other components were relatively simple, with a passive altitude control (*using hysteresis rods to produce a magnetic torque*) and a command card (*Atmel SAMD21J18A processor*) to schedule tasks. EQUiSat launched on July 13th 2018 and at the time of writing is an ongoing successful CubeSat mission. The software for the

entire system was written in C and used the FreeRTOS operating system [26].

2.4.5 PyCubed

PyCubed is a Python based CubeSat designed by the Robotics Exploration Lab at Stanford University. It consists of two cards, the mainboard and battery board. The mainboard contains the OBC, communication and power management electronics. The battery board is essentially just 8 3.6V Lithium Ion rechargeable batteries. The PyCubed design was used successfully in the KickSat-2 CubeSat, a CubeSat that launched femto-satellites. The mainboard uses the ATSAMD51 as the OBC. This OBC implements the Circuit Python firmware, allowing the entire system to be coded in Python. This processor has also been demonstrated to work beyond 1Mrad of radiation exposure without failure, however a watchdog using MAX70X series of processors is also implemented on this for redundancy. The board has modular radio connections, allowing the radio to be customised, but in the case of KickSat-2, a transceiver around 433 MHz was used. As the PyCubed has no payload in it's own right, it provides access to a series of General Purpose Input Output (GPIO) pins on the OBC and libraries to access these [27].

2.4.6 OreSat

OreSat is an ongoing open source CubeSat lead by the University of Oregon. The primary aim of the project is OreSatLive, an education platform, designed to encourage primary school and secondary school students into aerospace design. OreSatLive is a live streamed camera, using DxWiFi technology, that can be received using a ground station in Oregon. The intention of this project is for students to build ground-stations to access this live feed. Alongside this payload, there is also a cirrus flux camera, designed to study cirrus clouds and the possible connections with climate change. The primary communication method for OreSat is DxWifi, but a secondary radio is available for redundancy. A branch of SATNOGS, called University Class Open Ground Station (UNICLOGS) was developed to be the ground-station for this project and uses L-band (1.2 GHz), UHF (436 MHz) and S band (2.4 GHz). The internal board communications is done using CAN, with two CAN buses being used, one for safety critical control. The processor in use is part of the ARM CORTEX M4 family and uses ChibiOS as the real-time scheduler. Star trackers, magnetometers, IMU and GPS are used in the ADCS, with reaction wheels and magnetorquers for control. Currently, OreSat hopes to launch OreSat-0, a simplified version of OreSat, with no payloads in Spring 2021. A later launch will be made for OreSat, with the complete original design [28] [29].

2.5 Comparison of Open Source CubeSat Designs

Each open source design has it's merits and demerits. ArduSat and OpenOrbiter were omitted from further comparison as no design documents are available online at the time of writing,. This makes it impossible to proceed with those designs as initial starting points for the project. In order to decide on what designs would make for viable starting points, comparisons had to be made. (*Where references are not given, it can be assumed the data came for the resources discussed in the sections above*.)

2.5.1 Documentation

This is an important aspect of deciding the chosen design as good documentation and resources will reduce time wasted trying to understand the design.

- PyCubed is well documented, with a website[30] and GitHub [31]. There is documentation for every feature available with the firmware and example code, making PyCubed very accessible.
- OreSat features a website[32] and provide access to their GitHub [33].As the project is ongoing, the documentation is maintained for use by the current team and not necessarily ready for use by an external user. However, as the project is active, contacting team members would be possible.
- UPSat features a website [34] and some repositories on LibreCube's GitLab [35]. However, there is no documentation alongside the designs and code.
- EQUiSat has a website [26] and a Google drive [36] and GitHub [37] where code and resources have been store. Like UPSat there is little guidance for the resources available.

2.5.2 Successful Missions

Only OreSat has no recorded successful mission. With no first hand proof of working in space, this makes using OreSat less attractive than than the other design. Although it can be assumed OreSat will not be launched without the expectation of a mission success, it still reflects negatively compared to the successful missions of the others.

2.5.3 Structure and Size

Given this project is focused on constructing a test bed, the structure and size is not an important feature for consideration. However, it is envisioned, that this project may be used in a 3U cubesat in the future, making some designs more attractive than others.

- As PyCubed, is PC/104 compliant, it can fit in most standard third-party CubeSat cases.
- OreSat can be used as a 3U CubeSat, using the CAD and backplane models provided by the design team.
- UPSat requires a custom backplane, so expanding to 3U would require additional mechanical redesign work.
- EQUiSat does not use a backplane, but due to the simplicity of design, would require reworking to create new data lines between additional boards.

2.5.4 OBC and Firmware:

Each design has it's own command card and firmware that is used to write the mission specific software. The capabilities of the processors do not vary greatly, with all designs using some version of

the ARM M Cortex series. The justification for the difference between each series of the M Cortex largely comes down to mission specific requirements for communications, processing of data etc. This is something that is not a consideration for generic test bed the project is developing. For this reason, a more powerful processor would be favoured [38].

- EQUiSat uses the SAMD21 family of processors. These are Arm Cortex M0+ processors. Other features like communication are handled in part by the peripheral devices in EQUiSat making using the M0+ acceptable.
- PyCubed uses a higher specification version of the EQUiSat processor, from the SAMD51 family of processors. This is a member of the M4 processor family. As PyCubed is a singe board that acts as an OBC and handles communication and power management, it does require a higher computational power than a M0 processor.
- UPSat also uses M4 processors, this time from the STM32 family of chips. The STM32 was chosen not necessarily for the additional instruction set features the M0 boards do not have, but possibly due to the historical use of STM32 in open source designs. However, the documentation for UPSat provides no justification of the decision to use M4 as opposed to M0. Each subsystem uses a STM32, so the mainboard should be under relatively low stress.
- OreSat also uses the STM32 family of boards. The command board also handles communication, so a more powerful micro-controller had to be used than an M0. The M4 processor has additional support for digital signal processing, making it suitable for a control board that also handles communication.

The firmware is a far more important consideration. The style of RTOS varies, with the leading consideration for this, being to try and build a project that can be used generically:

- PyCubed is not a RTOS, instead just a programming platform. This brings both more freedom to add features and the option to build a simplified RTOS. However, it also means there is an initial programming opportunity cost to be considered for building a scheduler. It is maintained by the company Adafruit Industries, which ensures the software is commercial grade and well documented [39].
- FreeRTOS is used by both UPSat and EQUiSat. UPSat appears to have broken away the complexity needed by the OBC by using STM32s on each subsystem, including the payloads. FreeRTOS is well supported with documentation and countless books. It is also compatible with most boards. It features both pre-emptive and co-operative scheduling. However the code written by UPSat and EQUiSat as previously mentioned is not well recorded [40].
- ChibiOS is a used by OreSat. ChibiOS is lightweight fast RTOS designed for small RAM chips. It is also well suited to thread based scheduling and uses preemptive scheduling. It is a lesser know RTOS which does mean there is less support. However, the main designer appears to maintain good documentation [40].

2.5.5 Scientific Unit and Interchangeability

It should be possible to use large portions of the CubeSat designs already; just reprogramming the firmware/software and changing out the payload. However, with some designs changing the payload is easier than with others.

- PyCubed only has payload slots on the board, with access to open GPIO pins on the middle of the board. This means an adaptor card or rewiring is likely necessary to get the connections out to use with a separate payload.
- OreSat has two payloads, both of which should be possible to remove and replace with new designs, given it uses a backplane. The software would have to be updated to reflect any new CAN connections.
- EQUiSat's payload is just an LED beacon, so there are many connections available. Perhaps with a redesign of the mainboard the I^2C and SPI connections could be extracted.
- UPSat has two payloads, one that uses UART and the other SPI. In changing these, the software would have to redone to account for the change in 'heartbeat' that the EPS monitors. Care would have to be take with rewiring the new payloads, as no backplane or standard through connection is used between boards.

2.5.6 EPS and Power Budget

Each of these CubeSat designs, with the exception of PyCubed are made to spec and so the power available to a new payload depends on the spare budget available and whether the SU will be removed. This is not necessarily an important consideration for choosing the design to base the test bed off. The power needs of each CubeSat will change, so it may not be possible to have a generic one size fits all for power consumption on CubeSats. As a result, only quick estimates were made with the limited resources available:

- The power used by the PyCubed mainboard and the power budget is not available. The power depends on how low the batteries can be depleted, which is dependant on how the battery system is recharged.
- The original OreSat design EPS produces 37Wh, which cannot be discharged below 60%, leaving 14.8Wh available. The OreSat Live uses 2.3Wh and the cirrus flux camera uses 1.8Wh in 10 min sessions. The subsystems in stand-bye use 1.018 Wh per orbit(90 minutes). And the solar panels produce 1.634Wh per orbit. The OreSat payloads are expected to be turned on and off during orbit, so similar considerations would have to be made for any payload changes.[28]
- EQUiSat design has two battery systems, with a total power budget of 26.95 Wh. However for recharging battery usage is not expected to depleted more than 14%, leaving 3.773Wh available. At peak, the subsystems are expected to use 1.10Wh, with the LED flashing beacon at 2Wh. [41]

- UPSat has batteries that can store 44.4Wh and the only available power budget expects to use 22.5W. It is not clear what the battery limitations are, or the duration of the power budget (*time active per orbit*)), so the exact power available cannot be estimated without in-depth analysis.

2.5.7 Inter-Card communication

None of the designs use PC/104 style connectors. As mentioned before OreSat and UPSat have opted to use a custom backplane for power and inter-board communication. EQUiSat is essentially a two card CubeSat with battery packs, and so just uses custom connections for the few between boards. PyCubed does not use anything, as it is not a complete design, with all relevant communication happening internally on the PyCubed board. However, each board then uses different communication protocols for inter-board communication. Given the aim of this project, the more communication protocols available the more desirable the design is for reuse

- PyCubed is capable of SPI, I^2C, UART and Maxim OneWire, using the pins accessible on the payload section of the mainboard.
- The OreSat Command board is only natively capable CAN to handle inter board communication. However, there are additional pins available on the backplane, which could be reconfigured for some other communication protocols.
- EQUiSat uses SPI and I^2C to communicate with the Magnetic Random Access Memory (MRAM) and sensors.
- UPSat uses UART with the exception of SPI for the IAC, which uses SPI due to lack of UART pins on the OBC.

2.5.8 Radiation Protection and Testing

Radiation protection is not strictly necessary, as the design is expected to stay Earth bound. However, it is a benefit when considering potential future iterations. Each of the designs have undergone radiation testing, with the exception of OreSat which is still in development. However, OreSat has plans for radiation testing. Every design also appears to have made certain considerations, where deemed necessary, to use radiation hardened equipment instead of COTS.

2.6 Suitability of Open Source CubeSat Designs

Each design has it's trade-offs and no one design can be regarded as better. It is situational. EQUiSat, with its basic processor and low power EPS, is more suited to simple payload CubeSats. PyCubed provides a platform and the main components to further develop a full CubeSat. It is the most adaptable of the designs discussed. OreSat and UPSat are fully fledged CubeSats and powerful enough that they could be re-purposed for a range of payloads. The discussion in the sections above provide further reasoning to help decide what design maybe useful for a CubeSat project. The specific design chosen for this project is justified in section 3.1.2.

3 Methodology

Having reviewed the state of current CubeSats, two areas appear to be lacking. Firstly, there is not much common practice on designing FlatSats for CubeSats. A FlatSat is generally built using the components intended to be flown on the final project. This can leave testing part of a system to be quite late in development and dependant on the other components of a CubeSat. While a complete CubeSat will always have to be tested together, there is a missed opportunity for modular testing. If a component or card is finished before the rest of the system, emulation has to be done on the fly, or the inter-board testing delayed until the other cards are complete.

Secondly, it appears every open source CubeSat strives to be an exercise in designing a new CubeSat. These designs are started from scratch and are custom-made to a specific payload. One of the main advantages of open source is the ability to "stand on the shoulders of giants", i.e. to take a starting point and advance it further. However, this is often missed with the open source CubeSat designs. Re-purposing a CubeSat for a new payload can be a lot of work and at a minimum requires substantial coding. This raises the level of entry into CubeSats, especially for those less familiar with electronic and software design.

Bearing these two points in mind, the project focused on creating a high level emulation of a CubeSat that can serve as a configurable test bed. This presents itself as a FlatSat and mainboard, which allows individual components to be tested without the need for the rest of the CubeSat design to be finished. The mainboard forms the center of this test bed, which is programmed to schedule and run tasks.

Alongside this, other hardware boards are included in the design to emulate other CubeSat subsystems not currently built or tested. The boards largely use software emulation, with a physical data lines for inter-board communication. Using this system, there is the opportunity to create a framework that could be expanded and used as a modular open source CubeSat environment, where the user, especially non-engineering based users, can add a payload with relative ease and limited design changes required to the rest of a CubeSat design.

Following the literary review, and identifying the purpose and features of this project, a series of design decisions, outlined below, had to be made in order to achieve this. These decisions can broadly be split into hardware (section 3.1) and software (section 3.2) categories. Following from these decisions are the construction of the hardware in section 3.3 and software implementation in section 3.4.

3.1 Hardware

3.1.1 Choice of Test Bed

The FlatSat test bed design will use PC/104 Plus connectors to match the de facto industry standard. The design went through multiple versions and interactions. Initially the board was going to be two layers and fabricated by the university. However, the space required to route all connections on two layers was too great, leading to the decision to design the FlatSat to use four layers. Various spacing and the number of connectors were experimented with, with the final design decisions outlined below.

The final FlatSat has 2 X 2 PC/104 connectors, with an additional two for diagnostics and bridging to other FlatSats. Banana clips are used to allow for connecting to CubeSat power lines to power sources. With the PC/104 connections, it will also be possible to connect a battery card to the powerlines, instead of using the banana connectors. The design can be seen in fig. 3.

As CubeSats do not follow an internal standard, just an external dimension specification, compliance with the PC/104 standard, is often hand-waved or disregarded [15]. Many designs opt for connector of 104 pins, using 2x26 headers, which matches neither the PC/104 standard [42] or the PC/104 Plus standard [43]. It was decided to follow the PC/104-Plus standard, which uses 120 pins in a 4x30 connector configuration. The design decision behind this was: should someone decided to follow the PC/104 Plus standard, the FlatSat will be compatible and should someone just use the two 2x26 style headers or some other variation, this board should still be compatible (*dependant on pin spacing, a simple adaptor PCB may needed to be made*).

Some design aspects of the FlatSat do not completely follow open source methodology, but rather were chosen to suit the needs of the ISET department at the University of Glasgow. However, neither of these decisions should affect the use of this design in an open source setting. The power and ground lines were chosen to match Clyde Space's pinout. Clyde Space is a private space technology company, also based in Glasgow. The ISET department has had dealings with them previously and so this decision was made to help integrate future components that may be potentially bought from Clyde Space.

The orientation, number of connections, and space between boards was chosen to match another private company. This time it was ISIS - Innovative Solutions in Space. The ISET department has intentions of using CubeSat structures from this company. The FlatSat was then designed to match these measurements, making it possible to use the cards in one of their structures from 1U up to 16U, while still supplying power from the FlatSat [44].

The size of the tracks and clearances were initially determined using the IPC2221A standard [45] and the fabrication limitations of chosen manufacturer, JLCPCB. Deciding on values for a generic FlatSat was difficult, as it is not clear what future uses there may be. For this reason, maximum thickness values values were used that still maintained JLCPCB's and IPC2221A's clearance requirements and could be routed inside the dimensions for the 16U CubeSat structure. The signal lines in the final version are 0.2 mm and the power lines are 1mm, with a 0.2mm clearance between wires.

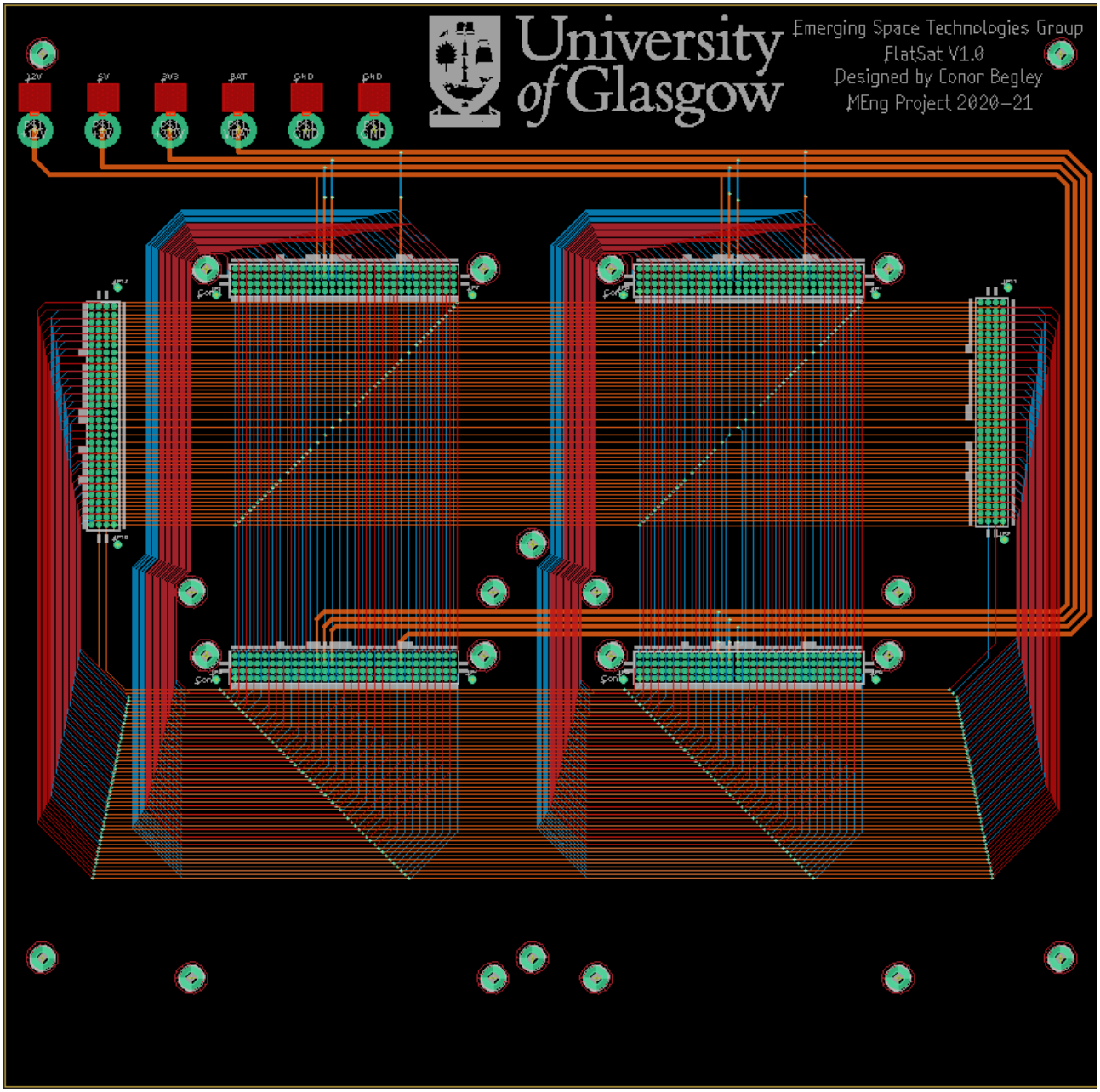


Figure 3: FlatSat PCB - *(with ground plane omitted)*

3.1.2 Choice of Open Source CubeSat Design

The two forerunners, following the literary review for choice of initial design, were OreSat and PyCubed. This was largely due to complete designs and demonstrable use in a space environment. UPSat was initially removed from the selection, as while the designs are available on their website, there is very little supporting documentation or papers on the subject. This will bring about a steep learning curve, that would consume too much time for this project. This was also true for EQUisat, in addition to the fact that there would be major redesign works needed to have a new payload. Initially it was decided both OreSat and PyCubed could be used. The project would try to be system agnostic, giving the user the choice between firmware and language. However, following correspondence with Andrew

Greenberg of the OreSat project, their command card is undergoing revisions and is not complete. At the time of correspondence, *(October 2020)*, OreSat was behind schedule and major redesigns were being done to produce a scaled back CubeSat that could still launch in Spring 2021. Deciding not to waste resources on a board that may have errors or no longer be supported by OreSat as redesigns happen, it was then chosen to soley focus on designing a system using the PyCubed mainboard.

3.1.3 PC/104 Adaptor

PyCubed does not not have a PC/104 Plus connector and none of the viable open source designs do. In order to use PyCubed with the FlatSat or stacked with other PC/104 boards, an adaptor had to be made. This means the PyCubed board will no longer be PC/104 compliant as the adaptor card is longer than a PC/104 card. This was acceptable as this design is in the first iteration and if the system was to be used in a space bound design, the PyCubed board would be rewired. For the time frame of this project, rewiring was not a suitable use of time. The adaptor plate design can be seen in fig. 4.

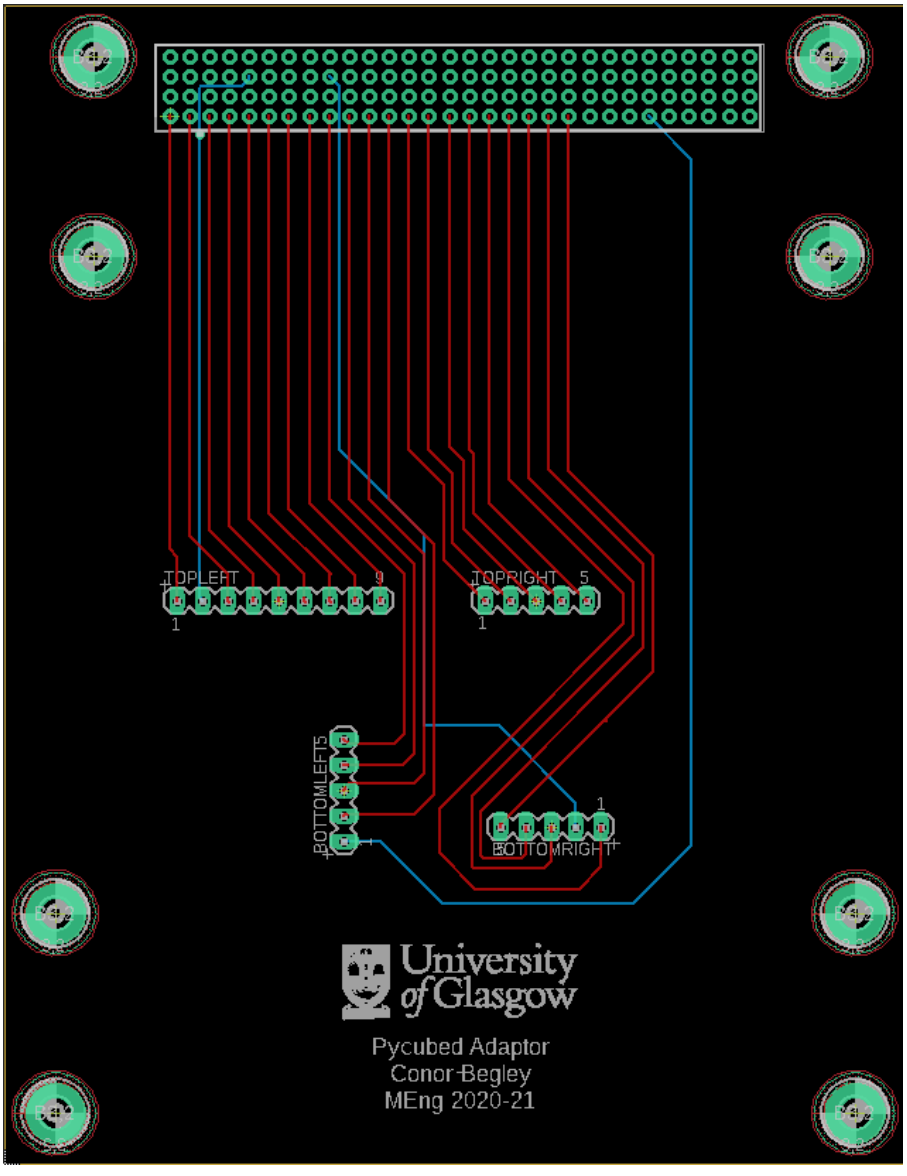


Figure 4: PyCubed Mainboard Adaptor

3.1.4 Demonstration Boards

The ESP32 was chosen to be used as the controller for the demonstration boards. The justifications for this are given in section 3.2.4. For the project, it was decided that the two demonstration boards would be designed. These boards emulate a SU and communications board. Considering other subsystems like ADCS or EPS, where the hardware will be performing most of the system function, there is not as much to emulate from a software perspective. However, these demonstration boards could be reprogrammed to act as any subsystem, producing the relevant incoming and outgoing signals. The demonstration boards are not intended to be fully functioning CubeSat ready designs. Designing payload and communication boards for a space ready CubeSat would be too time consuming for this project and also too costly. Instead these demonstration boards are intended to emulate real subsystem boards, demonstrating behaviour that could be expected in a real CubeSat system. In both cases, the boards are wired to run off a 5V power supply, using the same pins as the FlatSat power pins. However, it also can be powered via USB, which is often more desirable, as it allows for monitoring of the boards via a Read–Eval–Print Loop (REPL) style program.

3.1.5 Demonstration Payload Board

To be a good representation of a payload, there had to be some means of data input. The inputs for this board were chosen to be both passive and active. This allows for data to be constantly measured while also giving the user the options to interact with the payload. For the final design, this was chosen to be three switches and a thermistor. To help display states or outputs, three LEDs were also added to the board. The schematic is given below in fig. 5.

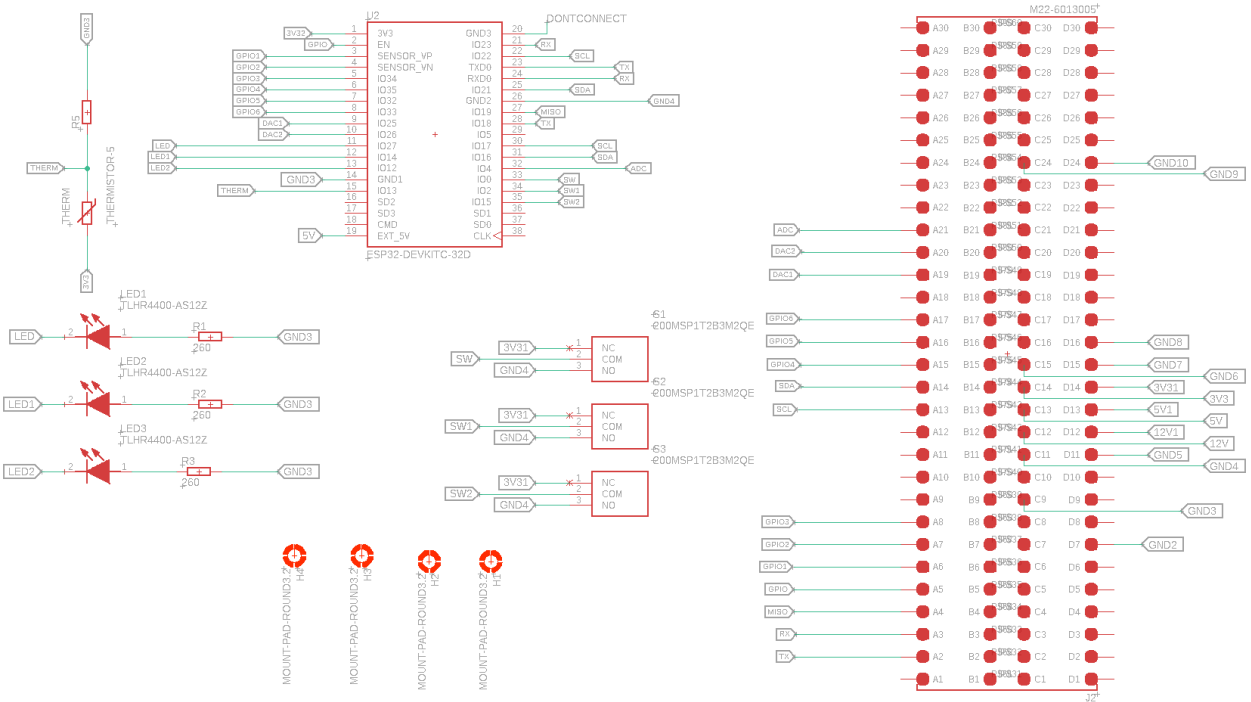


Figure 5: Demonstration Payload Board Schematic

The PCB version detailed is fig. 6 was designed to be PC/104 compliant, including mounting holes.

While the PC/104 Plus connector is required for the FlatSat, using the PC/104 dimensions strictly is not. However, by sticking to these dimensions, the hardware can also be stacked to make a CubeSat like structure. The design, itself has two layers so that while this design was manufactured externally, future versions could be made by the University of Glasgow's PCB fabrication equipment.

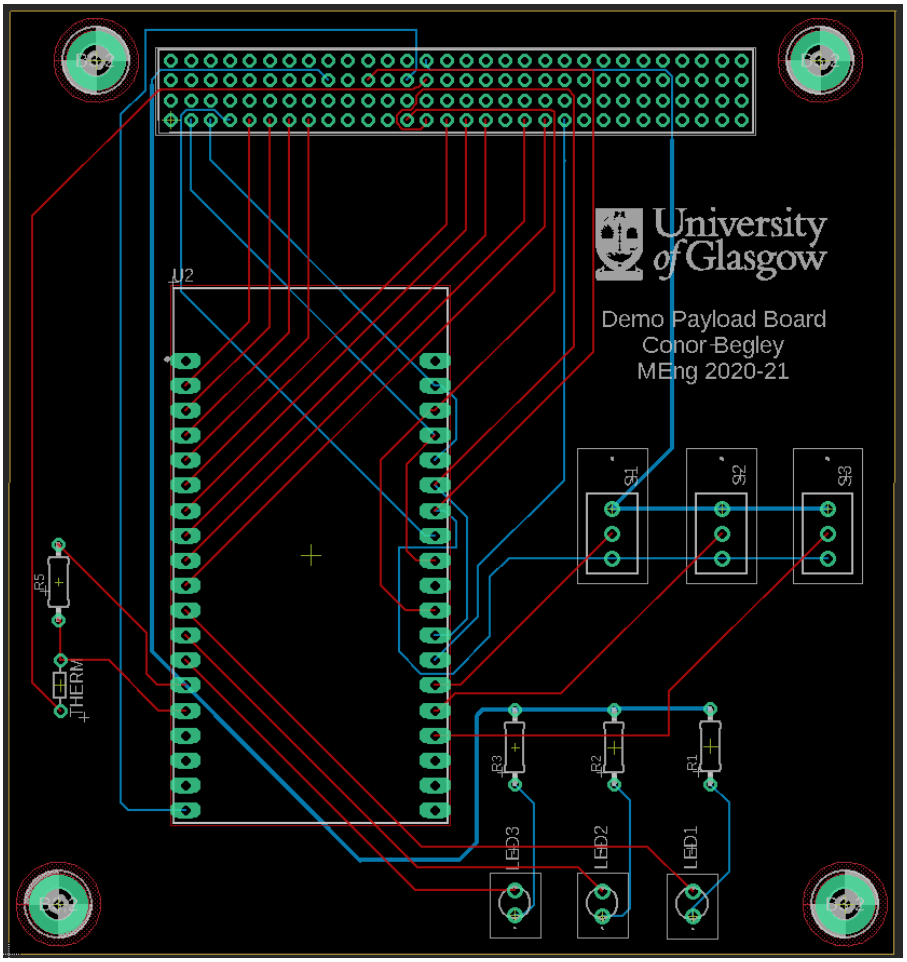


Figure 6: Demonstration Payload Board Schematic

3.1.6 Demonstration Communication Board

Given the possible need to acquire radio permission and licensing for testing, having a functioning radio card and receiver was not feasible. It would also require development and testing of the radio features, something that is not a direct focus of this project. Instead, a simple screen was chosen to represent the radio, displaying data that was 'sent' to a ground station. The screen chosen was a TFT 2.8 Inch LCD screen that uses a ILI9341 driver chip [46]. It uses SPI, but the ESP32 has two SPI peripherals, allowing for the other to still be free for other use. If data has to be sent to the communications board to represent ground station uploads, a custom function on the mainboard could be used to handle this. There is also the option of using the file storage system on the ESP32 or a REPL program to interact with the ESP32 from a computer. The schematic for the communication board is given below in fig. 7 and PCB in fig. 8..

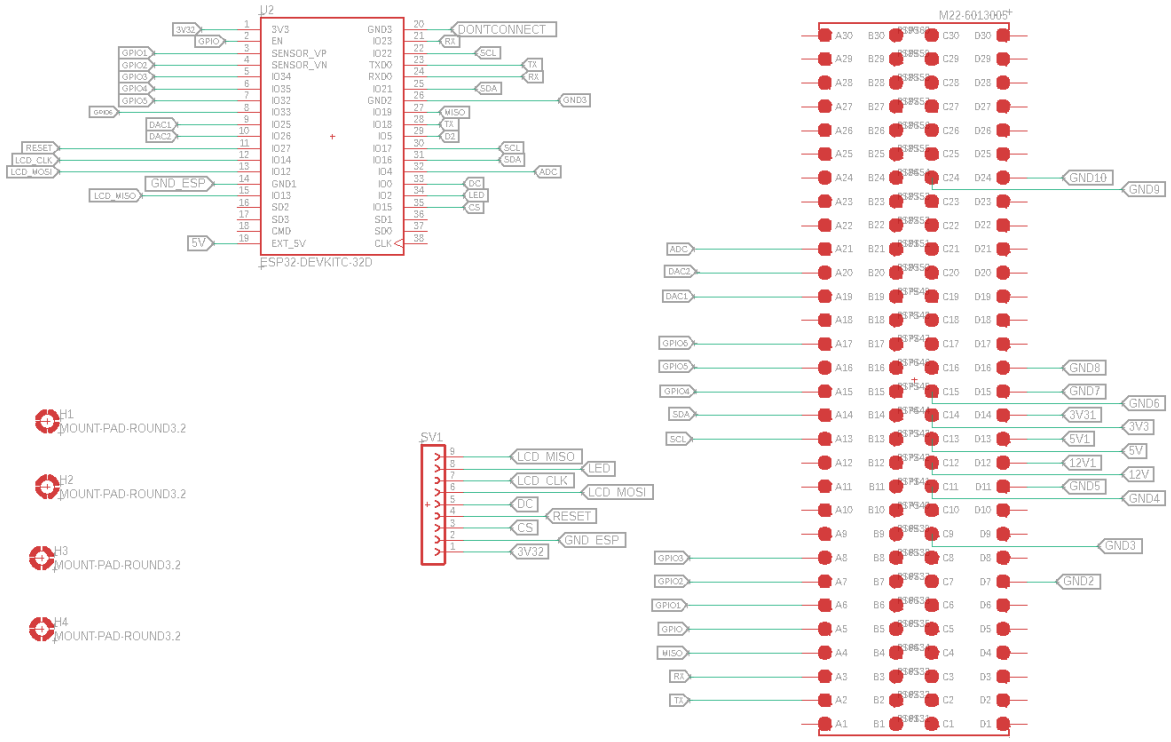


Figure 7: Demonstration Communication Board Schematic

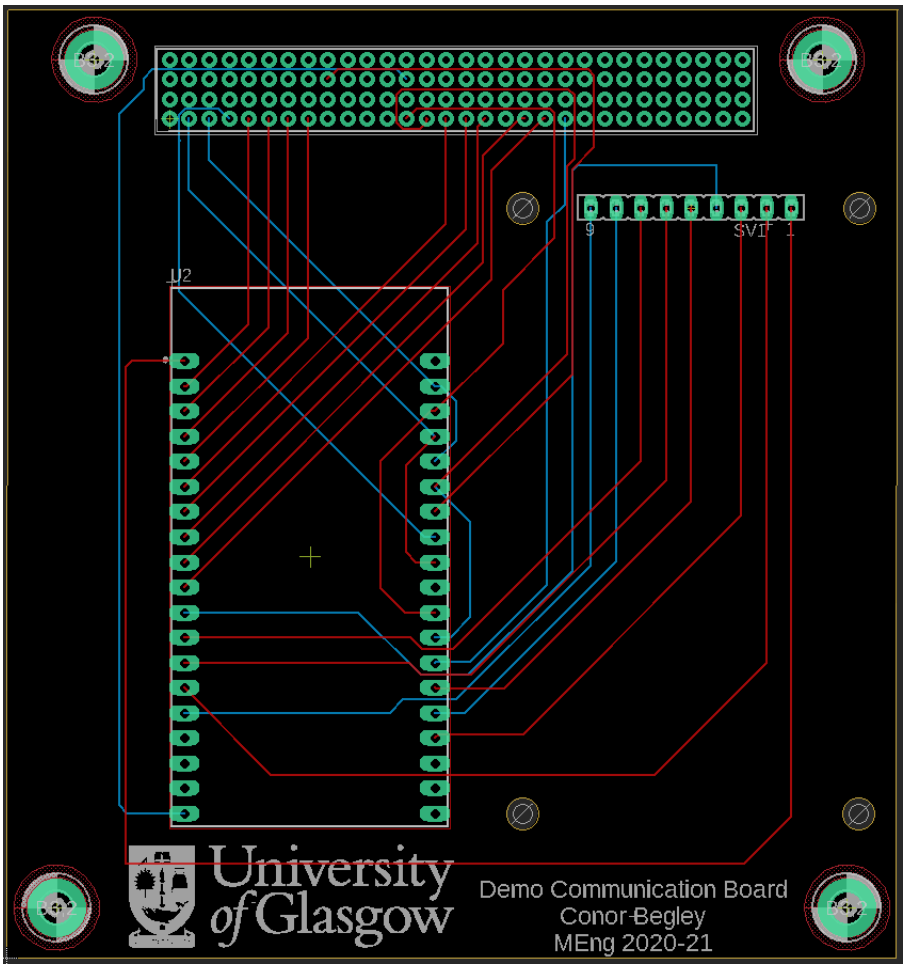


Figure 8: Demonstration Communication Board Schematic

3.2 Software

Before coding and implementing the software system, a series of design considerations had to be made. With these considerations made, the system could be designed quickly and less time wasted developing features and prototypes that would not be used.

3.2.1 User Input

The most important consideration was how the user would interact with the system. The user input has to be simple enough that someone new to the concepts of CubeSats or programming could use it, while giving experienced users the freedom to implement complex ideas. To help realise these possibilities, two features were considered: configuration files and a scheduler class.

For beginners there would a series of configuration files, with a range of parameters and customisation features. The user should then only have to enter a specification of task, device etc. into a file. These files have default parameters which should allow the user to only have to define the values for their tasks that change.

Meanwhile, advanced users would be assumed to have programming and design experience. On top of using the configuration file system, there is the option to run custom functions, defined in Circuit Python. Additionally, the project code was defined as a self contained object, so that a user could import the class and design some scheduler or series of functions around this class. As everything is open source, there will always have to be considerations for the option to modify the source code to fit an advanced user's need.

3.2.2 Scheduling

As this is an emulation of a CubeSat and by extension a RTOS, the tasks provided by the user should be able to be scheduled. There was a range of scheduling algorithms that were considered, with each one prioritising different features [47].

- **Round Robin:** Round Robin attempts to be the fairest scheduling system, allowing each task a short time slot to run before suspending the task, to run something else. Each task is revisited until it is complete. New tasks are added to the end of the queue.
- **First Come First Served:** As tasks are generated, they are added to the end of queue. When a new task is run, it is taken from the start of the queue. This ensures the task that was submitted first is run first.
- **Shortest Task First:** Whichever task has the shortest duration is run first. If short tasks are repeatedly queue, then long tasks are not chosen and are starved of run time. A fairer alternative version of that does not starve tasks, is the shortest elapsed time first, which runs the tasks that have be run for the least amount first.

- **Priority Scheduling:** Tasks are chosen by a priority that does not directly relate to the run-time of the task. This is usually a numeric value that denotes the importance of the task, ensuring the most valuable or critical tasks are run first. Again low priority tasks can be starved. A fix for this is 'aging' tasks that have not been run by increasing their priority value.
- **Rate Monotonic Scheduling (RMS):** This is scheduling algorithm that ensures all tasks are run for their full period at regular intervals. RMS assumes all tasks have a hard deadline, that is that all tasks must be done by their period(*deadline*). With RMS is possible that a series of tasks are too long to be scheduled as deadlines will be missed. This is the case even if some tasks would have soft deadlines i.e they could be delayed to run after their deadline without catastrophic results.

Having considered these types of scheduling, one stands out from the rest, RMS. Each other scheduling system uses context switching and some prioritisation. Space compatible RAM is an expensive resource in CubeSats, so not using context switching can have advantages by using less RAM. Additionally, most OBC tasks should be short and not require long time slots. The exception to this is calculations and data processing. However, it is reasonable to expect that the CubeSat will collect raw data and transmit this to a ground station. For what this project is envisioning, it is also reasonable to assume the ADCS will be responsible for calculating any course change necessary. RMS also generates a repeating loop of tasks, simplifying the code, as task generation is not necessarily needed.

In testing to identify errors in a function's runtime, it is beneficial to develop a system where a function can run to completion instead of using a context switch. The OBC will be routing signals to various cards, so it will be important that they cannot be interrupted. Additionally unless the user has experience in operating systems, they may not understand context switching, priorities, threading etc. This can add an additional challenge to using the system instead of just considering the order of tasks to be run. RMS does guarantee all tasks will be completed, which is beneficial for testing code behavior. Furthermore, as every task is considered in terms of duration and period, it is easy to determine the task order on paper. This can also be visualised as a timeline, something that may help inexperienced users instead of trying to understand different task states.

An issue with this decision is handling interrupts. If an RMS system is interrupted, deadlines will be missed and the guarantee of all tasks being run lost. For this reason RMS does not allow interrupts. However, Circuit Python currently does not support interrupts, so this was always going to be an issue with PyCubed even without RMS. The proposed alternative would be a pseudo interrupt handler that polls the incoming signals at intervals defined by the user. This is sub-optimal, but perhaps something that can be improved when Circuit Python adds interrupt handlers natively.

3.2.3 Input Options

There are three main types of file that define the project specification:

- **Tasks:** These files will be functions that the mainboard is expected to run. As previously mentioned, it should be possible to give a specification for a task, that runs custom code. However,

the main purpose of these tasks will be to define the routing of data to and from the mainboard. Duration and period will need to be provided for these tasks to be used in RMS.

- **Devices:** This represents any hardware other than the mainboard used in a system. The file will define the type and properties of the connections made to the mainboard.
- **Interrupt Handlers:** These files should define any functions that will be checked at regular intervals for important signals. As it is in a RMS system, they would not be true interrupt handlers, given they will be polling the signal.

3.2.4 Demonstration Board Firmware

In order to test this design but also to demonstrate the functionality and usability of the design, PC/104 cards emulating CubeSat subsystems were constructed(*d*iscussed in section 3.1.4). The programming language used on these boards did not have to match Circuit Python, as the boards would be communicating via a protocol, which would be standard across all devices. Looking at common programming languages for micro-controllers, there were four main choices:

- **Mbed:** An open source programming platform that uses a version of C++ and was created by ARM. As it is maintained by ARM, it is compatible with most ARM based processors [48].
- **Ardunio:** An open source hardware and software platform. There is direct support for any official Ardunio board, but there also exists ports for various other micro-controllers. The software runs are a version of C++ [49].
- **Micro-Python:** An open source platform that runs a stripped down version of Python 3, with not all libraries in Python 3 included in Micro-Python. There is an official Micro-Python board, but many ports exits for a range of micro-controllers[50].
- **Circuit Python:** A derivative version of Micro-Python that is also open source and maintained by Adafruit. Adafruit develops libraries for their peripherals and any processor breakout they have released in recent years can be used with Circuit Python [39]. This is the firmware the PyCubed mainboard uses.

Alongside deciding the platform, the functionality of the demonstration boards had to be considered. As they will be emulations, any demonstration board could act as any other board, from the point of view of software. However, there are some hardware considerations. First off, due to the cost of the mainboard and FlatSat components, keeping the demonstration boards cheap was a motivation, especially as it was unlikely these designs would ever be used in a CubeSat. This then did bring the benefit of not having to use radiation suitable components. At this point in the project, it had be decided and accepted that only PyCubed and not OreSat were going to be used. As CAN was not a built in technology of PyCubed, it became a tertiary aim of the project and not a consideration of the demonstration boards. An external CAN bus could be used, so it was not necessarily a direct requirement of the demonstration board or the micro-controller.

There is no clear advantage from a specific platform based on the hardware available. Similar boards can be used under each firmware. Instead the choice initially was between platforms using C ++ (*Mbed, Ardunio*) and platforms using Python (*Micro-Python, Circuit Python*). Each of these languages have advantages and disadvantages, summarised in table 1 below. Ultimately, the decision lies between an easier to use language with less low level customisation or a lower level, very customisable language that is harder to code [51].

Table 1: Comparison of Python and C++

Feature	Python	C++
Learning Curve	Python's syntax is quite similar to English and the control flow is dictated by white-spacing, reducing the user's concern for tracking brackets for control flow and semi-colons in end statements. It is clear what sections of code are in logical blocks	C ++ is a lower level language and so has more complex, less English syntax as well as needing special characters to keep track of control flow and end statements
Memory Safe	Python is completely memory safe, with dynamic sized data structures and a garbage collector to manage the heap.	C ++ is memory unsafe, with segmentation faults and array overflows as common risks. The concept of pointers as well as dynamic C ++ data structures using malloc for dynamic variables etc. may be hard for inexperienced users to program.
Complexity of Behaviour	As Python is high level, understanding certain features can be confusing if not understood. For example unlike simpler type variables, Python lists that are passed by reference, and can lead to unexpected behaviour if the user edits the list.	As C ++ is lower level, less functions are included in the default libraries and the complexity of data structures is more clearly defined and easier to reason about.
Optimisation	Python is high level, implemented in C. As a result, it can be hard to really fine tune the code, predict exact behaviour or memory usage.	C ++ is a lower level language, making predicting behaviour and optimising memory usage easier than higher level languages

The benefits of Python make it a clear choice. The project aims to design a system that those unfamiliar with programming can still use. Following that, if they hope to customise a feature, it should be easier to do so in Python. The choice then fell between Micro-Python and Circuit Python. For the demonstration boards, they would have to communicate properly with the mainboard. This meant the

firmware should be capable of allowing the device of acting as a slave in at least either I^2C or SPI.

Natively, Micro-Python or Circuit Python does not support being a slave. However, there is a branch of Micro-Python called Loboris Micro-Python [52]. This is a branch specifically for the ESP32 series of boards [53]. It improves on various features in the Micro-Python firmware, taking advantage of specific features of the ESP32. This firmware version allows for using the ESP32 as an I^2C slave. Using the board as a SPI slave would not be possible with this configuration, but is also not possible in the original version of Micro-Python and Circuit Python. However, an SPI sensor or similar could be used for testing the SPI features. Choosing to use an ESP32 also has the added benefit that it can be used as a C++ platform.

3.2.5 Design Testing

As with any software system, some form of testing will need to be available to ensure the code is working correctly. Alongside this, in the spirit of open source, these test cases should allow a user to modify the source code while ensuring the modifications would not break any existing code. However, this is not so easily done in Circuit Python, or any micro-controller OS for that matter. For communication protocols like I^2C and UART, errors are produced if no message is received, connected devices can not be found etc. Unlike Python, which has the capability to mock objects and tests, this is not really possible with internal libraries of Circuit Python. This ultimately means that current testing will have some unit cases, with a number of physical test cases for testing communication methods functions. It may be possible, in a future iteration of the project, to design libraries and mock objects for communication protocols in Circuit Python, but this is beyond the time and scope of the project.

3.2.6 Tools and Programs

One final consideration for the software side of this project was deciding on what development tools to use. Not only did they have to perform certain functions, but had to ensure the maintainability of the project for future iterations. In addition, the tools also had to be suitable for less experienced users.

For programming the PyCubed board, the PyCubed designers recommend using the editor Mu [30]. Mu can be used on Windows and macOS but Mu is not fully supported on Linux. However, it is likely Linux users will have their own chosen IDE already. Regardless of the system, programs can be added to the file system by mounting the drive as a USB device and transferring the relevant files. Mu acts as a REPL terminal, giving access to the PyCubed device and allowing programs to be run. For Linux any REPL programme like Picocom can be used to communicate with the board.

For the demonstration boards, the software chosen was a combination of Ampy, Picocom and Mu (*or another IDE*). Ampy and Picocom were recommended by Loborris [52]. Ampy is used to load programs onto the ESP32. The ESP32 cannot be mounted as a drive, so instead the user needs to transfer files using serial connections and this is what Ampy is used for. As for Picocom, this is used as a REPL, allowing for programs to be run in a Python shell on Linux. For macOS users the screen program can be used and for Windows users Putty can be used as well as Mu.

3.3 Construction

3.3.1 Breadboard Designs

In order to design initial test schematic designs, breadboards were constructed for the demonstration Payload and demonstration Communication boards, using the same components as the final PCB designs. When COVID-19 delayed the PCB assembly, these where then used to test and design code. These boards can be seen in fig. 9 and fig. 10

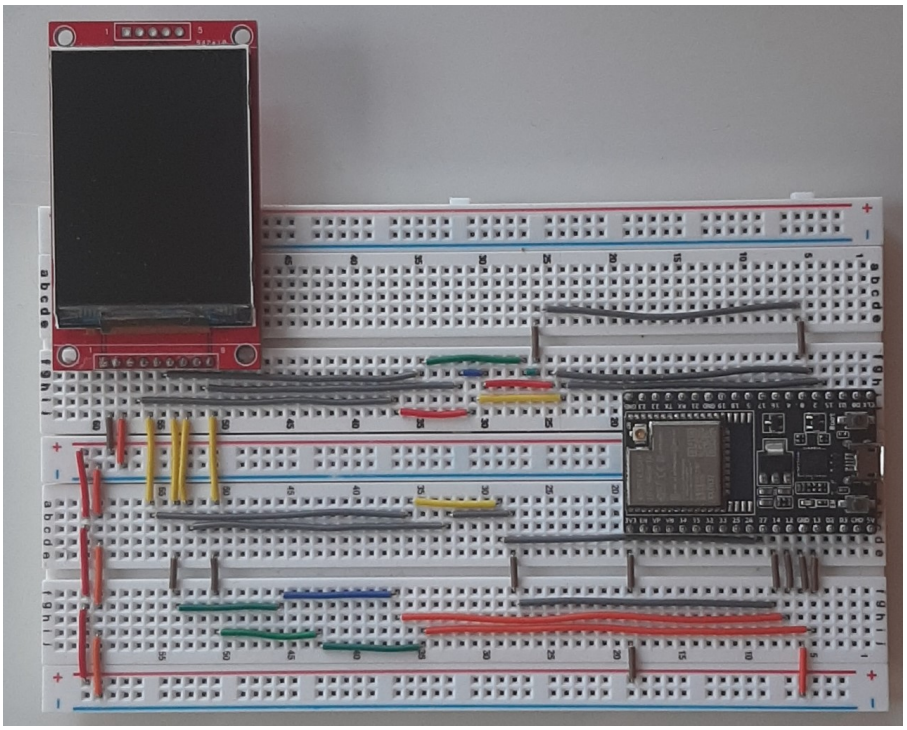


Figure 9: Breadboard Demonstration Communication Board

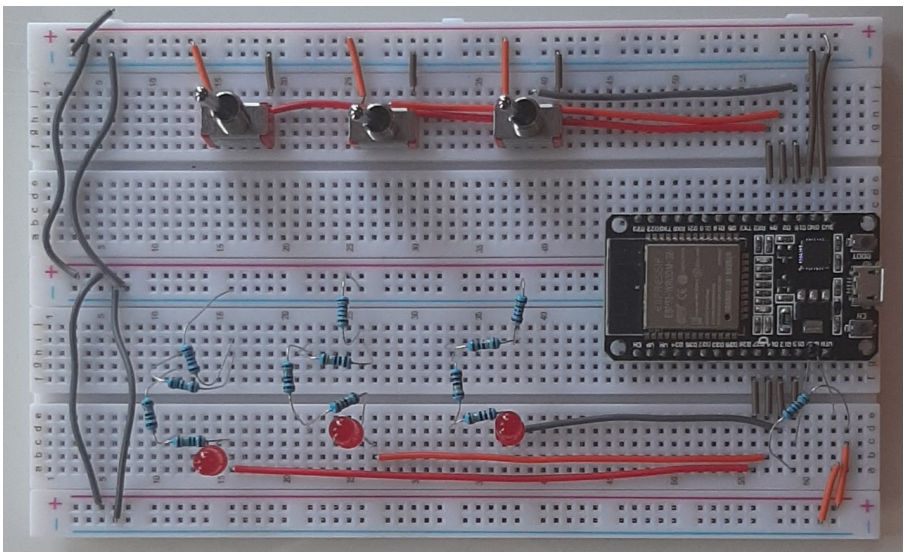


Figure 10: Breadboard Demonstration Payload Board

In order to replicate the PyCubed board, the Adafruit Grand Central Express M4 breakout board was used [54]. This was chosen as it uses the same SAMD51 family of processors as PyCubed and is Circuit Python compatible. As this was a temporary solution, in-depth analysis of boards was not needed, so long as the chosen board used the same firmware version and had the same functionality as the PyCubed mainboard.

As the FlatSat could not be used, an additional breadboard was used to act as the FlatSat. However, only the necessary connections were wired to the boards. This was also done, accounting for the limitations of connections available to the PyCubed board, with the UART being shared by all payloads. In fig. 11, the wires used are as follows:

- **Yellow:** Serial Data line (SDA)
- **Green:** Serial Clock Line (SCL)
- **Brown:** Mainboard Receive Line (RX) / Demonstration Board Transmission Line (TX)
- **White:** Mainboard TX / Demonstration Board RX
- **Blue and Purple:** GPIO pins for interrupt handlers, ready signals etc.

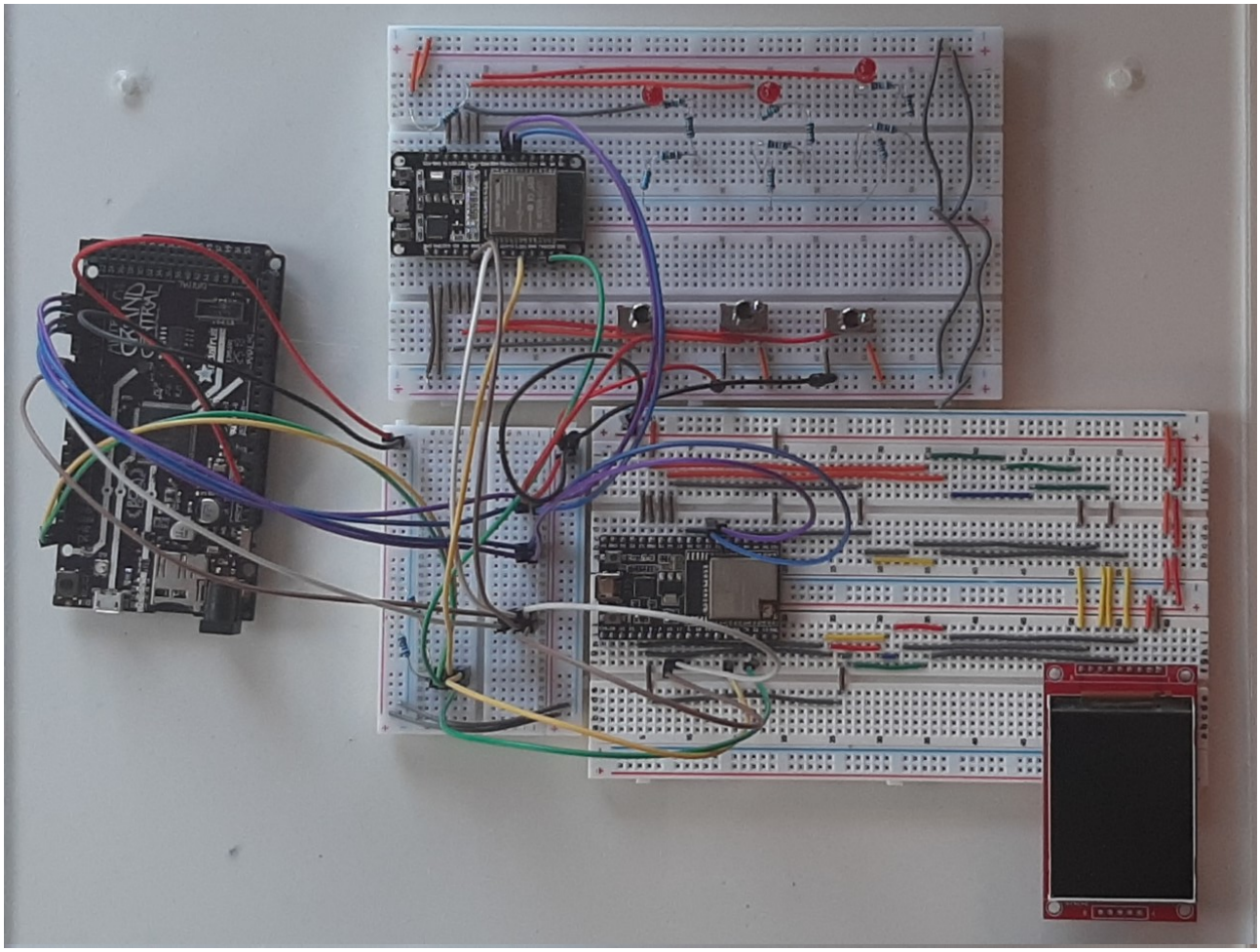
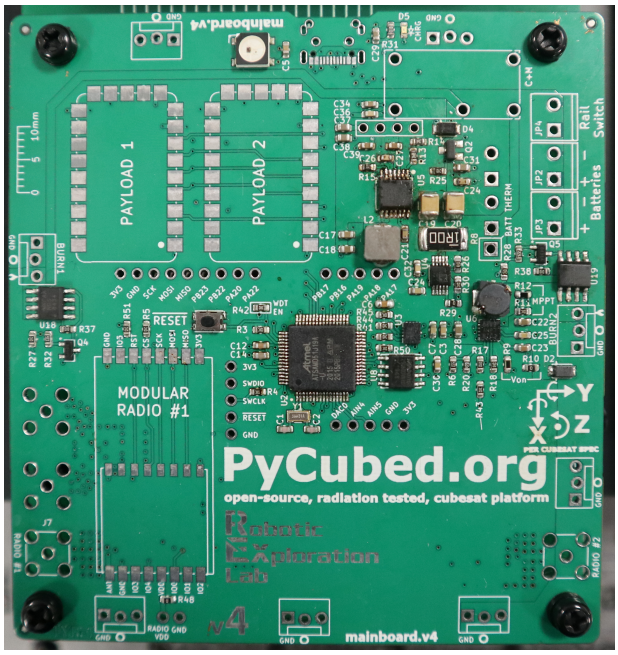


Figure 11: Breadboard FlatSat with Adafruit Grand Central Express M4 as the Mainboard

3.3.2 PCB Designs

All boards were ordered and fabricated at various stages throughout the project. However, PCB soldering and assembly was not completed by the report deadline, as explained in the COVID-19 forward. The PyCubed mainboard and battery boards were soldered and built. However, due to a part order issue the mainboard could not be tested in time. The battery board was fully assembled and appeared to be producing the correct voltage when initially test. A fault may have developed, causing batteries to heat up, but this could not be investigated due to lab closure. They can be seen in fig. 12.



(a) Soldered PyCubed Mainboard

(b) Soldered PyCubed Battery Board

Figure 12: Soldered PyCubed Boards

The FlatSat and demonstration boards were not assembled in time for the report deadline. There was an issue with the initial footprint design exacerbated by COVID-19 lab delays. The PC/104 Plus connectors pins are square and the footprint designed was circular. With that, there was not sufficient margin left in the hole size to account for this size difference and the pins could not fit. The boards had to be re-ordered in December, after realising this issue when the labs reopened after lockdown. However, these updated versions of the PC/104 cards and FlatSat were not assembled before the labs closed at the end of December. A mock up of unsoldered components on these boards can be seen in fig. 13.

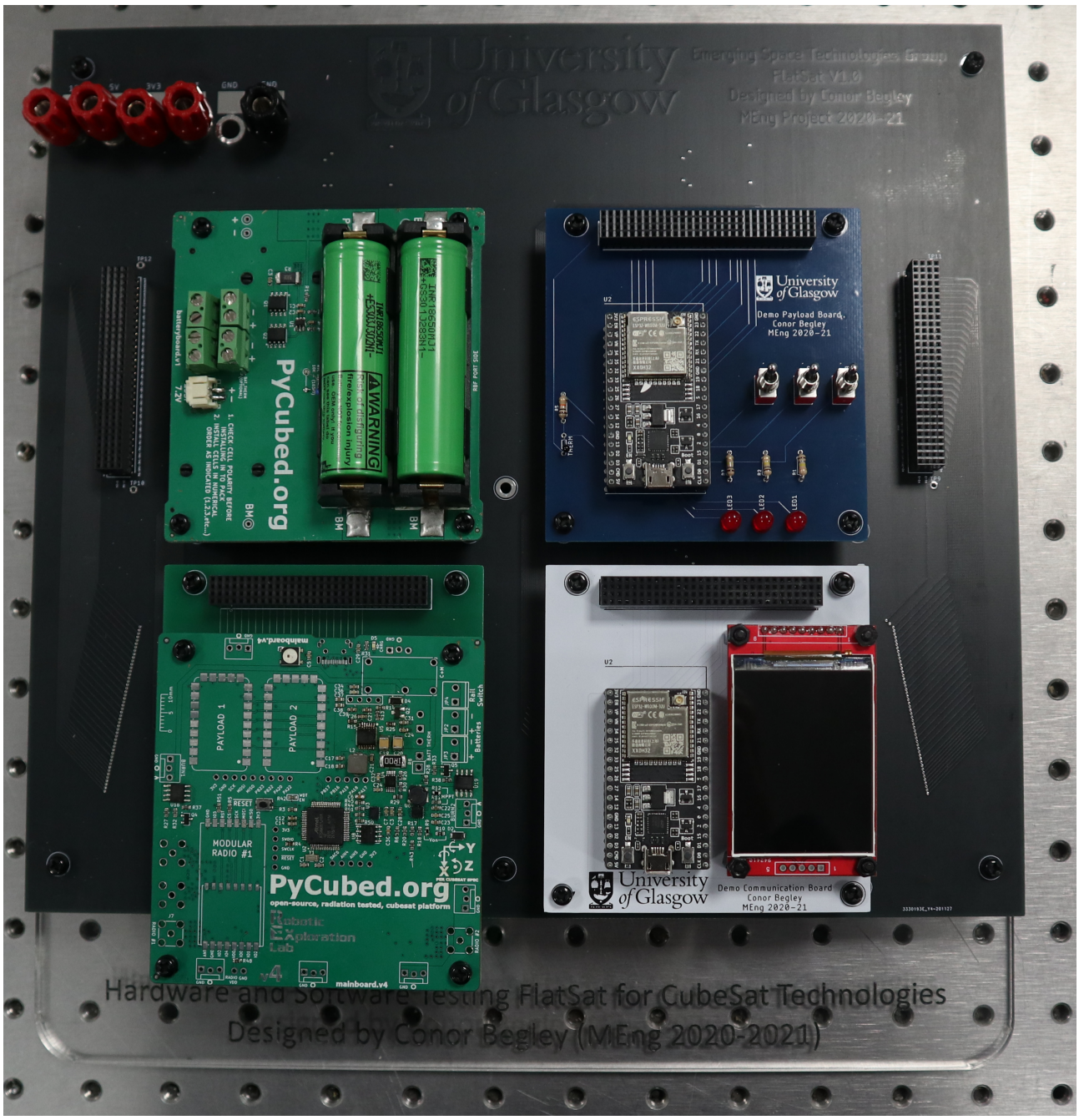


Figure 13: Example FlatSat Setup
Battery Board(*Top Right*), Payload Board (*Top Left*), PyCubed Mainboard (*Bottom Left*), Communications Board (*Bottom Right*)

3.4 Software Implementation

The following sections aim to highlight the main points relating to how the software was designed and coded. Most lines of code have not been shown here, in order to keep the explanations succinct. However, the code base can be seen in appendix A and on Github (`https://github.com/C-Begley/masters_project`).

To give a brief overview of the system, the user will provide a series of definitions in files. These files represent tasks, devices and interrupt handlers. A scheduler will read all these files and process them. Based on this input, the scheduler will produce a task timeline and will run these tasks repeatedly until the system is stopped. A flow chart for this can be seen in fig. 14 and is explained in further detail in the sections below.

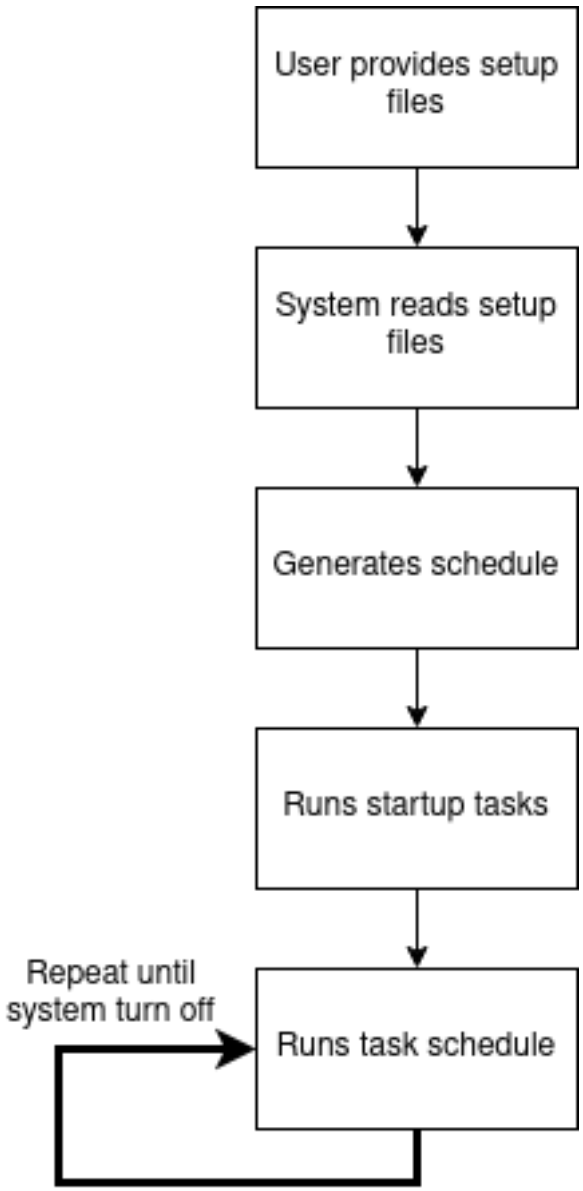


Figure 14: Software Flow Chart for System Overview

3.4.1 File Structure

The code base has three main files the user should not need to change. These files detail the classes used in the back-end of the system to handle scheduling and communication protocols. The only time these would be changed is if the user is expanding on the functionality of the system or changing the fundamentals of the scheduling.

- **mono_sch.py:** relates to RMS and is detailed in section 3.4.2.

- **com.py:** relates to board communication and is detailed in section 3.4.3.
- **scheduler.py:** relates to handling all task scheduling and is detailed in section 3.4.5.

These are three files which the user may edit. This will generally be done by a more experienced user, looking to add custom functions or interrupt handlers. This may also be done if the user wants to change how the scheduler runs:

- **custom_functions.py:** relates to any custom task code written by the user.
- **interrupts.py:** relates to custom interrupt handlers written by the user.
- **main.py:** relates to running of entire system and is detailed in section 3.4.6.

3.4.2 Rate Monotonic Scheduling Implementation

The RMS is handled by helper functions to the main program. There are two main helper functions and they can be found in the <mono_sch.py> file. Before understanding what these functions do, concepts from RMS must be understood. This program implements RMS using the empty slots method. This can be considered like a timeline, where each task takes up a set number of slots on the timeline. Each task must be run before it's deadline and when it is finished running, the deadline countdown resets. The first step in identifying this timeline is to find the lowest common multiple of the deadlines of all the tasks to be run. At this point in the timeline, the timeline pattern will repeat itself. To ensure all tasks can be run within this timeline and by each task's deadline, the total task utilisation must be found [55].

For a given task, the task utilisation can be defined as

$$U_i = \frac{C_i}{T_i}$$

where U_i = Task Utilisation, C_i = Task Duration and T_i = Task Period (*deadline*)

For tasks to be possible to be scheduled with RMS, the total tasks utilisation must be less than 1.

$$U_t = \sum_{i=0}^{n} \frac{C_i}{T_i} \qquad U_t < 1$$

where U_t = Total Task Utilisation, C_i = Task Duration, T_i = Task Period (*deadline*)

To model the real world and the fact that a task can not be immediately run after the previous one, an upper bound limit is introduced. This is due to considerations like the processor needing to find and load the next task to run. For this project, the Upper Bound Limit Theorem was used, which is the standard limit used in RMS [56].

$$U_b = n(2^{\frac{1}{2}} - 1), \qquad U_b > U_t$$

where U_b = Upper Bound Limit, n = Number of Tasks, U_t = Total Task Utilisation

Based on this theory, the two helper functions where created. Both helper functions need a dictionary of tasks to be run. The dictionary should map the task name to another dictionary with two fields, period and duration i.e.

```
{"<task_name1>" : {"duration":<int> , "period":,<int> },
 "<task_name2>" : {"duration":<int> , "period":,<int> },
 ...
 "<task_nameN>" : {"duration":<int> , "period":,<int> },
}
```

Any extra fields are ignored by the program and do not cause errors. This allows for the task file format discussed in section 3.5 to be used. The first function possible_schedule() returns a Boolean to indicate if it is possible to schedule the task or not. It is no more than an evaluation of the above mathematical expressions, so the function is not discussed here. The second helper function, schedule_order() attempts to schedule the tasks. For separation of concerns, this function does not the use possible_schedule() and so care should be taken when using it. The main part of this code is in in appendix A.1 and a flow diagram can be seen in fig. 15.

In schedule_order(), all tasks are initially placed in the unscheduled list, the scheduled list is empty and time starts at zero. The period of the timeline is found using the lowest common multiple. A loop runs until this period is reached. The unscheduled tasks are kept sorted by the next approaching deadline, so that the first task in this list is the next task to be run. This task is then added to the task_pattern(*timeline*), removed from the unscheduled list and added to the scheduled list. On each loop round the scheduled task lists are checked. If the task deadline has been surpassed, the task is re-added to the unscheduled tasks, and its deadline updated. On each loop, if a task is scheduled, the time is updated by the period of the task. Otherwise, the time is updated by a incremental step value. Once this loop terminates, a complete task pattern is produced as an ordered list of tuples, of the task name and time it should be run at. A simple example can be seen below.

```
task 1 - duration = 2,000, period = 5,000
task 2 - duration = 1,000, period = 10,0000
task 3 - duration = 1,000, period = 20,0000

task_pattern = [(task1, 0), (task2, 2,000), (task3, 3,000) (task1,5000),
(task1, 10000), (task2,12,000), (task1,15,000)]
```

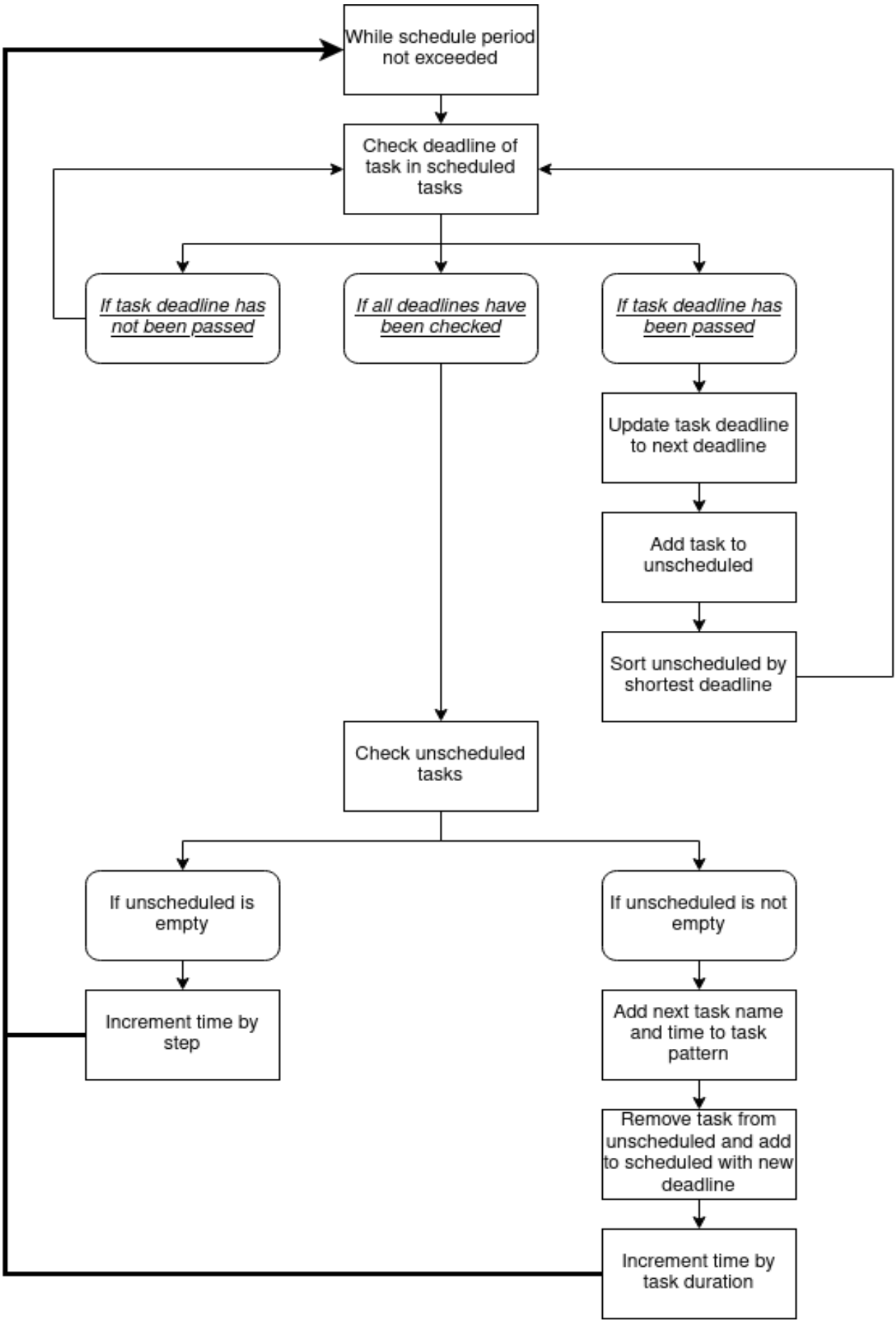


Figure 15: Software Flow Chart for RMS Task Scheduling

3.4.3 Inter-Device Communication Implementation

To make using the Circuit Python communication methods both easier for the main scheduler and for future users, communication helper functions were made in the <com.py> file. This is a library to use the UART, SPI and I^2C communication protocols of the mainboard. It can also be used for communication methods outside of the scheduler, as a separate library, for use in custom functions or otherwise. Each helper function has a series of steps, dependant on what the user wants to do. Each helper function takes generic parameters to indicate if data should be written, read etc. In addition to these, the helper function takes communication specific parameters for baud rate, timeout, parity etc. The majority of the parameters have a default value, so that only the ones needed have to be set as arguments. The code for each function in <com.py> can be seen in appendix A.2 and is also further explained in section 3.5.2. A generic overview can be seen in the fig. 16.

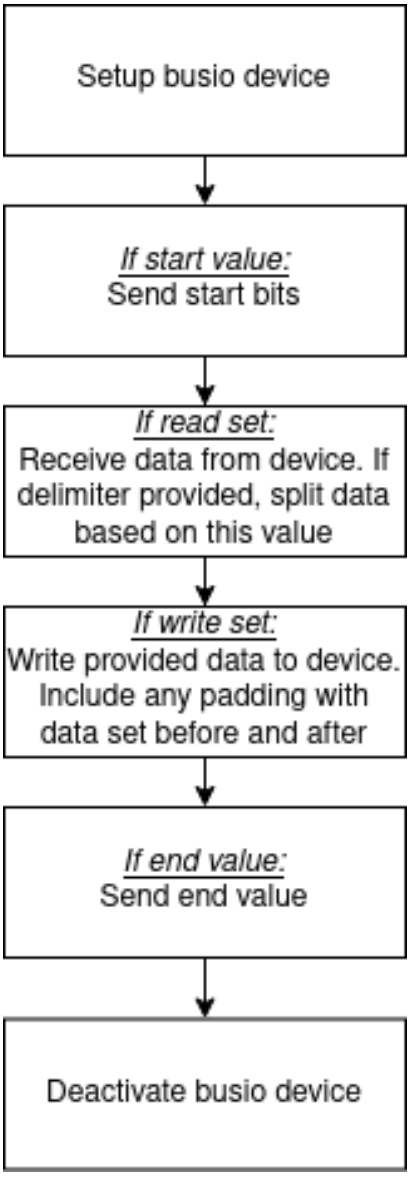


Figure 16: Software Flow Chart for Generic Communication Task

These helper functions allow for reading and writing to be done in the same function call. When reading, the received data can be split into a list based on a delimiter e.g '\n', '\t' , ';' etc. Writing can add a repeating symbol before or after the data to be sent as padding e.g a data padding of 0 for three bits before the data 101 looks like 000101. Reading is done before writing to prevent any data hazards from overwriting data that is supposed to also be read. An exception to this is with SPI. A separate function is used if data is to be written and read. SPI discards incoming data if a write action is performed or writes junk out if a read action is performed. To prevent data being lost, the SPI helper function handles SPI reads and writes simultaneously.

Some functions in <com.py> are included that do not directly relate to communication functions.The first of these is called add_default_dict() which takes two dictionaries, the second of which represents all the default values a dictionary should have. The function adds any key value pairs to the first dictionary that are present in the second dictionary but not present in the first. If the first dictionary has a key, its value left alone. However if a key is not present in the first dictionary, it is added to this dictionary and set to the value in the default dictionary. Nested dictionary values are handled accordingly. A flow diagram can be seen in fig. 17. This allows for default settings to be set for tasks. This is explained further in the setup function in section 3.4.5.

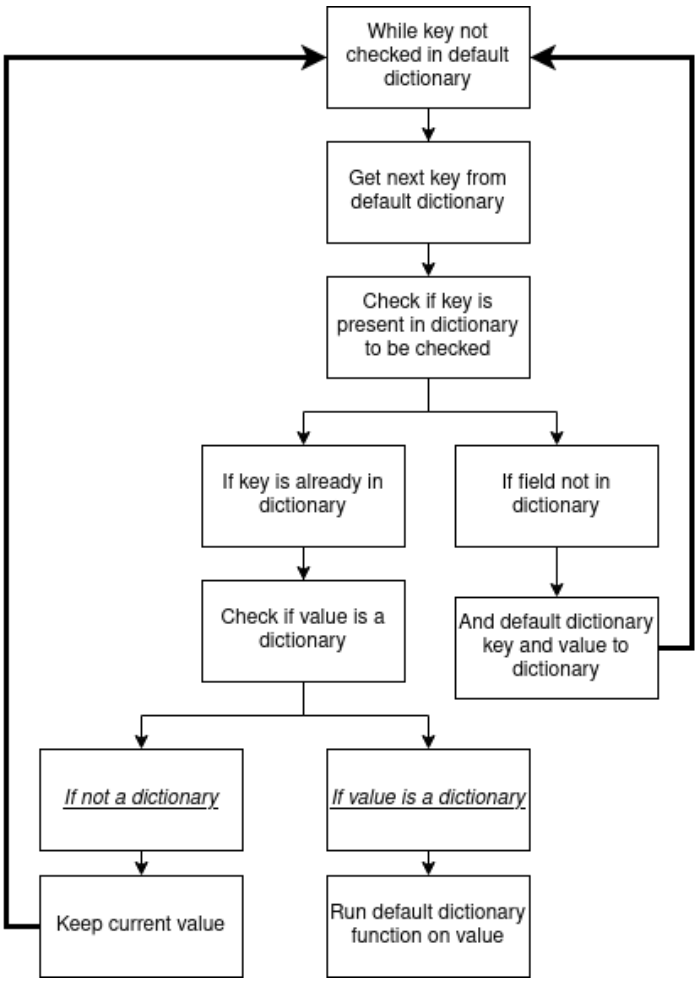


Figure 17: Software Flow Chart for Default Dictionary Function

The <com.py> library also contains the estimate_duration() function which can estimate the time for a communication task to run. This is provided to save the user from working out the task duration. However, it is not as accurate as figuring out the average time for the task by running it over a large number of iterations. The time to read and write to the buffer and file was estimated from running simulations on a range of values for ten thousand iterations. The same was done for the creation and tear down of busio objects for I^2C, UART and SPI. The time to send or receive data is estimated using the respective baud rate or frequency. A margin value can be added to the final estimated duration for safety. The margin value is a float, which represents a percentage of the estimated duration.

However, the function does not estimate the duration for the run-time of custom functions. If there is a custom function, this time must be worked out or estimated by the user. However, it can be provided using the 'time' field in the custom_function fields of a task (*see section 3.5.1*). The alternative is to set the scheduler to estimate the duration of the task by running it. This is done in the setup of the scheduler and explained in section 3.4.5.

3.4.4 User Input

As discussed in section 3.2.1, the user input is a series of files. However, the exact format was not decided on before beginning code development. This was done in order to leave room for experimentation and to simplify system requirements. Initially, free form text files were going to be used, alongside complex text recognition to parse it. However, the system design became very complex, very fast with this approach. While every user may not have coding experience, trying to build an entirely English based input system was unreasonable for the scope the project. Instead, it was decided a file system would be used : JavaScript Object Notation (JSON). There were other options like CSV(*comma-separated values*), XML(*Extensible Markup Language*) etc. However, Circuit Python just provides native support for JSON and CSV. For what was envisioned with hierarchical data structures and default entries, CSV was not an option with its table like structures, so JSON was chosen. With JSON, it does introduce a learning curve that may not be present with CSV as the user needs to understand JSON syntax like lists and dictionaries. Examples of these file templates can be seen in section 3.5.

3.4.5 Running Tasks

There are two parts to the main program in <main_scheduler.py>. The first is the setup function and the second are functions responsible for running tasks and the schedule. Both are contained within a class object which maintains the buffer, connected devices and task schedule. The intention is to run all of these tasks together as in section 3.4.6. However, each function can be run individually as well. The class fields can be retrieved and set to allow a more experience user use the class as they need. The code for this class can be seen in appendix A.3.

The first function, setup(), is used to process the user generated input and prepare the scheduler to run tasks. A flow chart for this function can be seen in fig. 18. It identifies any files on the mainboard and sorts them into groups of devices, interrupt handlers and tasks. Each type of file is read in, alongside

the default settings for each one. Then using the add_default_dict() function from section 3.4.3, the full profile is generated. Each of the default settings files have default names, but can be changed to suit the user's need in the function call. The tasks are then checked to see if they can be scheduled. If they can not be scheduled the function exits. Otherwise the task schedule timeline is generated.

During this setup phase there is the option to provide a task schedule timeline, so that the program does not have to generate one. There is also the option to increase the step size when generating the the task schedule timeline to reduce the time taken. As the schedule generates the timeline with a nanosecond resolution, this is often needed to save time for tasks with long periods. The task timeline can be retrieved from the task_pattern field for future use.

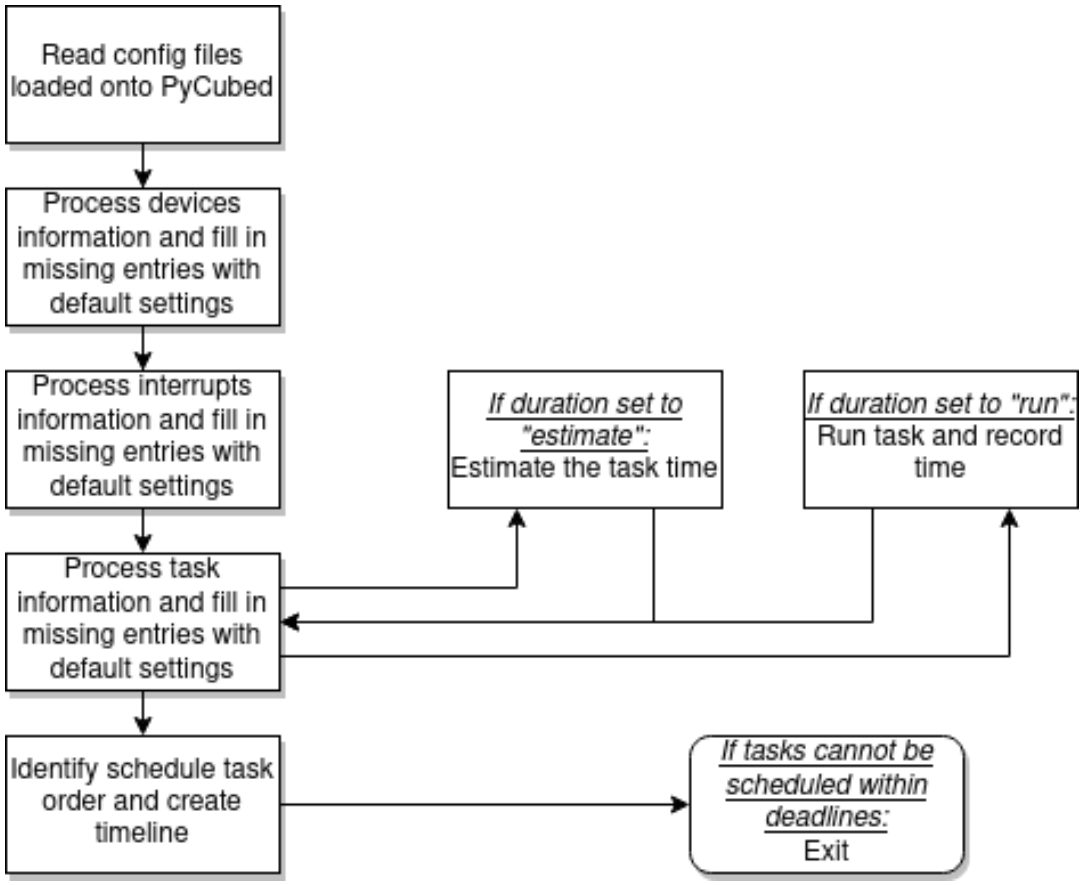


Figure 18: Software Flow Chart for Setup Function

The run_task() function relies on a series of helper functions to handle communication, which in turn call the relevant communication function from <com.py>. These helper functions just call the <com.py> function and handle any pass through to other devices. While this could have been done in the run_task() function, it would have bloated the already large function. There may also be the chance to develop a curried function which handles these helper functions in a later version, helping to reduce code repetition and maintain separation of concerns better. The flow chart for this function can be seen in fig. 19.

The first thing run_task() checks for is if the task is an interrupt handler or not. As explained in section 3.2.2, the interrupt handler is a custom function to poll incoming signals. As Circuit Python does

not handle interrupts, any interrupt handlers should be set to a short duration. If the task is a normal task, a series of checks are made to see what fields are set. If a field is set, the corresponding activity is carried out. The first step handles any pins that should be set active high or low. After that, if data is to be loaded from a buffer or file this is done. It should be noted the file takes precedence over the buffer, if both are attempted to be used to load data. Next a custom function can be run before a task. Then the communication part of the task is run. After any communication is carried out, the custom function can be run again, followed by data being stored to a buffer or file.

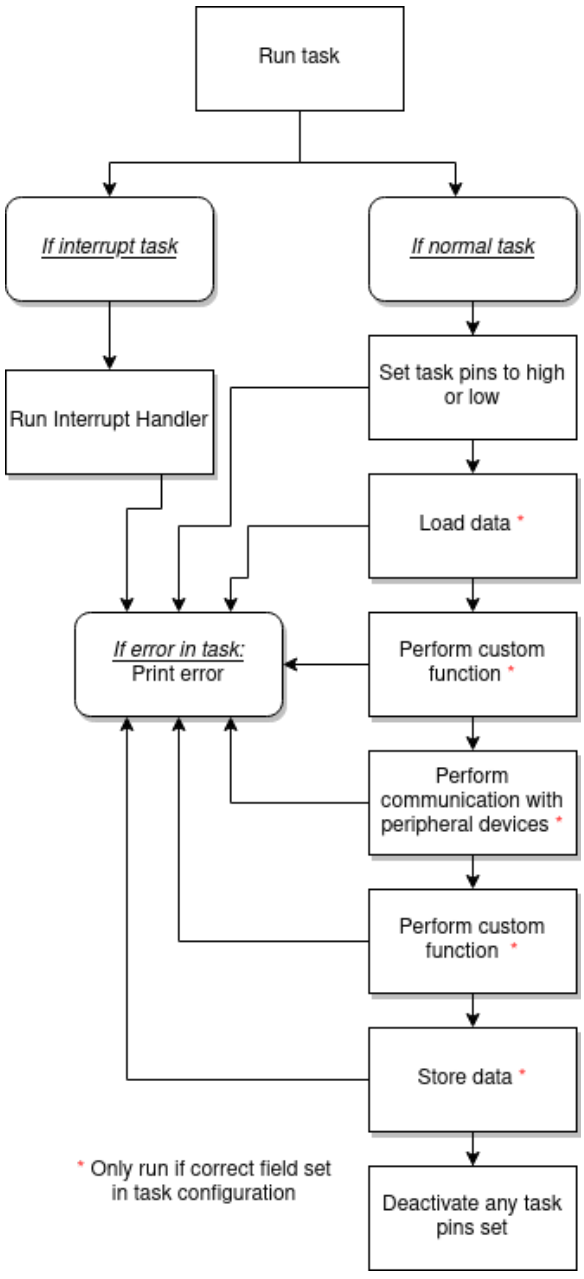


Figure 19: Software Flow Chart for Run Task Function

The run_schedule() is responsible for selecting the next tasks to run and maintaining the task timeline. To begin, a task with an infinite deadline is added to the pattern as a marker for the end of the timeline.

The scheduler runs at nanosecond level accuracy. The loop checks each time round if the timestamp of the next task in the timeline is reached. If it is reached, run_task() is run for that task. When the timeline ends, the timeline resets to the start and the loop runs again. The flow chart for this function can be seen in fig. 20.

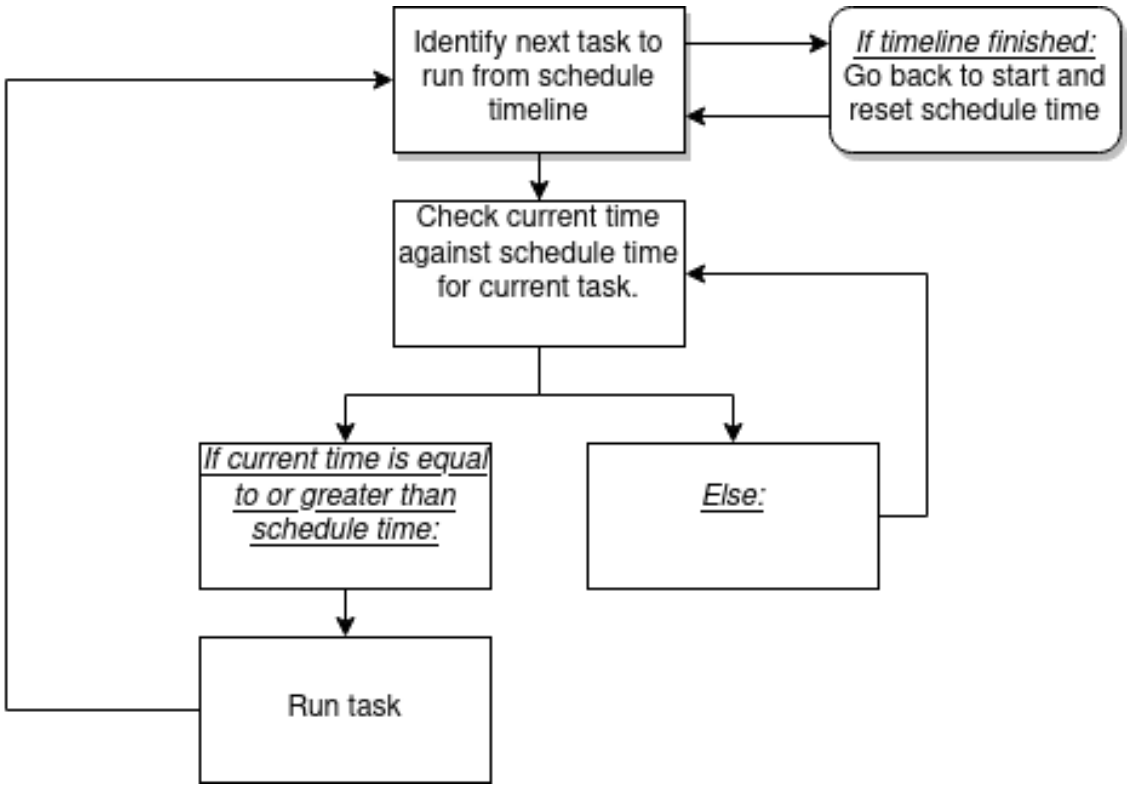


Figure 20: Software Flow Chart for Run Schedule Function

Alongside run_schedule(), the option for startup tasks is provided in this class. The function run_schedule() does not terminate and repeats all tasks in an endless loop. While this is ideal behavior for a scheduler, it is not perfect for a CubeSat system. Certain devices have to be configured when starting up. Should the user want to run a one time function like this for initialising a device, this is not possible with run_schedule(). Instead, run_startup() can be used. The function run_startup() essentially calls a modified version of setup and runs all tasks once with run_task(). An additional file, with a startup task name on each line, has to be provided to indicate running order.

3.4.6 Main Code

In order to run any tasks, the schedule class needs to be created and the setup and run functions called. This needs a basic level of programming knowledge. To further remove this requirement and increase accessibility for inexperienced users, a main file was created. This handles the creation of the scheduler, setup, startup tasks, and calling the run schedule function. This main file also assumes a default file structure, which should help with the organisation of tasks. The code for this can be seen in appendix A.4 and a flow of an example running system can be seen in fig. 14.

- **../** : This contains <custom_functions.py>, <interrupt_functions.py>, <main.py> and library files
- **/devices/** : This contains the default devices file and any device entries
- **/startup/** : This contains the default file for start up tasks and any start up task entries
- **/tasks/** : This contains the default task file and any task entries
- **/interrupts** : This contains the default interrupt file and interrupt handler entries
- **/files/** : This is where any data files should be stored and accessed
- **/testing** : This contains the unit tests and mock libraries

3.5 Software Input File Implementations

As discussed in section 3.2.1 and section 3.4.5, the user interacts with the system through a series of files. Each task and interrupt handler has a separate file and the devices have just one file. In the following the sections, the file templates that act as file default settings are discussed, followed by example entries for each type of use case. In addition to explaining the file structure, these sections are intended to serve as reference for any user hoping to use the system.

3.5.1 Default Task File Template

This is the file structure for any task relating to reading, writing, storing or loading of data and custom functions:

```
{"duration": 0,
 "period":   0,
 "connection" :{
                "device_name":"",
                "type": "",
                "send" : false,
                "data" : "",
                "receive": false,
                "data_delimiter" : ""
                "data_size" : null,
                "pass_through" : false,
                "pass_location" : ""
 },
 "connection_settings" : {
                          "front_data_padding" : 0,
                          "back_data_padding" : 0,
                          "start": true,
```

```
                        "start_value": 0,
                        "end": false,
                        "end_value": 0,
 },
 "load_settings" : {
                    "load_from_buf": false,
                    "load_address_start" : 0,
                    "load_address_end" : 0,
                    "load_from_file": false,
                    "load_file" : "",
                    "line_delimiter": "",
                    "load_line" : false,
                    "start_line": 0,
                    "end_line":0,
                    "disjoint": false,
                    "lines":[ ]
 },
 "store_settings":{
                   "store_to_buf": false,
                   "store_address" : 0 ,
                   "store_to_file": false,
                   "store_file" : "",
                   "file_append" : true
 },
 "pins":{
         "pin_high" : [ ],
         "pin_low"  : [ ]
 },
 "custom_function": {
                 "custom" : false,
                 "before" : false,
                 "after" : false,
                 "custom_function_name": "",
                 "arguments" : [ ],
                 "time" : 10000
  }
}
```

- **duration:** This describes how long a task takes to run in nanoseconds. Using the value "run" the system can run the function and get the period from that. There is also the option of "estimate" where the system estimates the duration using average values for the communication features

used.

- **period:** This describes how often the task has to run in nanoseconds. It can also be considered the deadline of a task.
- **connection:** This describes how any data is to be sent out or received by the mainboard.
  - **device_name:** This describes the device to send data to or to write to. The name must be defined in the devices file also.
  - **type:** This describes the type of connection to use. It is currently implemented for SPI, I^2C and UART.
  - **send:** This indicates if data is to be sent to the device listed, set to true or false.
  - **data:** Data to be sent if "send" is set to true. This can also be left empty with data loaded from buffer or a file (*see load_settings*).
  - **receive:** Indicates if data is to be received from the device listed, set to true or false.
  - **data_delimiter:** If the data is multi-lined,it describes the character(s) the data should be split on.
  - **data_size:** Describes the size of the data to be received in bytes, if "receive" is set to true.
  - **pass_through:** For data that is received, there is the option to send it on to another location, set true or false.
  - **pass_location:** Names the location of the device that data is to be passed through to.
- **connection_settings:** This describes any formatting of the data to be sent or received.
  - **front_data_padding:** This is a number of "0"s to be added at the start of a signal, useful if data is stripped when processed.
  - **back_data_padding:** This is a number of "0"s added at the end of a signal.
  - **start:** This indicates if there is a start value to be sent before the data, set to true or false.
  - **start_value:** This is the byte string that will be sent, if start is set to true.
  - **end:** This indicates if there is a end value to be sent after the data, set to true or false.
  - **end_value:** This is the byte string that will be sent, if end is set to true.
- **load_settings:** These are the settings that need to be configured if data is to be retrieved for sending or use in a custom function. The program favours loading from a file over a buffer.
  - **load_from_buf:** This indicates if the data to be sent should be loaded from the scheduler buffer, set to true or false.
  - **load_address_start:** This is integer value for the starting address location of the data to be loaded from the scheduler's buffer, if load_from_buf is set to true.

  - **load_address_end:** This is integer value for the starting address location of the data to be loaded from the scheduler's buffer, if load_from_buf is set to true.
  - **load_from_file:** This indicates if data should be loaded from a file, set to true or false.
  - **load_file:** This is the name of the file data is to be loaded from, if load_from_file is true.
  - **line_delimiter:** These are characters to indicate what characters are used to show how lines are terminated in a file.
  - **load_line:** This indicates if data is to be loaded from a series of lines in the file, if set to true, otherwise the entire file is loaded.
  - **start_line:** If load_line is true, this is the starting line number.
  - **end_line:** If load_line is true, this is the ending line number.
  - **disjoint:** If this is set to true, lines are loaded based on the line numbers in the lines field.
  - **lines:** This is a list of all the line numbers that should be loaded if disjoint is true.

- **store_settings:** These are the settings that need to be configured if data received is to be stored in a buffer or file. The program can store to both buffer and file simultaneously.
  - **store_to_buffer:** This indicates if the data received should be stored to the scheduler's buffer, set to true or false.
  - **store_address:** This is the address location where the buffer should start storing the data.
  - **store_to_file:** This indicates if the received data should be store to a file, set to true or false.
  - **store_file:** This is the name of the file for the received data to be stored, if store_to_file is set to true. Must use either the absolute path or the relative path from wherever the main programming running is saved.
  - **file_append:** If set to true, the task will add the data as a new line to the existing data. If set to false it will overwrite the entire file.

- **pins:** These are the settings for configuring output pins during a task.
  - **pin_high:** These are the list of pins that are to be set high for the duration of the task.
  - **pin_low:** These are the list of pins that are to be set low for the duration of the task.

- **custom_function:** These are the options that allow the user to run functions in addition to sending or receiving data.
  - **custom:** This is set to true to show if a custom function is to be run.
  - **before:** If this is set to true, that custom function will run before the sending or receiving data.
  - **after:** If this is set to true, the custom function will run after sending or receiving data.

- **custom_function_name:** This is the name of the function to be run, as defined in the function dictionary in the <custom_functions.py>.
- **arguments:** This is a list of any additional arguments that should be passed to the function.
- **time:** This is an estimate of how long the custom function runs for. This is only used if the duration for the entire task is not provided and the communication time has to be estimated.

3.5.2 Default Device File Template

This file defines the properties peripheral devices have

```
{"I2C":{"address":32, "SDA":"D12", "SCL":"D13", "baud":100000, "timeout":255},

 "UART":{"RX":"D0", "TX":"D1", "baud":9600, "bits":8, "parity":null,
         "stop":1, "timeout":1, "receiver_buffer_size":64}

  "SPI" : {"CLK":"D52", "MOSI":null, "MISO":null, "slave":null, "write_value" : 0,
           "baud":100000, "polarity":0, "phase":0, "bits":8}
}
```

- **I²C:** This defines the the properties of an I²C connection with a device. As a I²C bus is usually common to all devices in a system/subsystem, often the only setting that needs to be changed from the default settings is the address of a device.
  - **address:** This is the unique address (in decimal) to communicate with the device. For I²C it must have an address in the range 8 to 119 (I²C has reserved addresses in 0-7 and 119-127).
  - **SDA:** This defines the SDA pin on the mainboard used to communicate with the device(s).
  - **SCL:** This defines the SCL pin on the mainboard used to communicate with the device(s).
  - **frequency:** This refers to the clocking frequency of the SCL line in Hertz.
- **UART:** This defines the properties of any UART connection with the device. UART can use the same lines for multiple devices. However, this may cause noise or frame collisions on data being received/sent if multiple devices attempt this at the same time. With RMS, there should not be an issue, unless the subsystem cards route data to each other outside of using the mainboard.
  - **RX:** This is the UART RX on the mainboard, it should be connected to the TX of the device.
  - **TX:** This is the UART TX on the mainboard, it should be connected to the RX line on the device.
  - **baud:** This is the baud rate of the UART connection, measured in symbols per second.

- **bits:** This is the number of bits in a UART frame.
- **parity:** Null means no parity, while the value odd is used for odd parity and even is used for even parity.
- **stop:** This is the number of stop bits, either 1 or 2.
- **timeout:** This is the time in seconds the mainboard should wait while trying to read UART data.

- **SPI:** This defines the properties of an SPI connection with a device. SPI tends to use the same Master In Slave Out (MISO), Master Out Slave In (MOSI), and Clock (CLK). Often only the Slave Select, which indicates which peripheral is in use, needs to be changed.
  - **CLK:** This is the clock pin, used to synchronise data transfer between devices.
  - **MOSI:** This is the pin used as the data line from the mainboard to the peripheral (MOSI).
  - **MISO:** This is the pin used as the data line from the peripheral to the mainboard (MISO).
  - **slave:** This is the slave select pin used to tell a peripheral communication with that device has begun.
  - **write_value:** This is the value that is written to the slave when data is being read from it. With SPI for every byte of data sent, a byte of data is received. When a read from a slave is made, data must be written to the slave and when data is written to a slave, data is also received from the slave. However this extra data being written or read is often ignored.
  - **baud:** This is the baud rate of the SPI connection, measured in symbols per second.
  - **polarity:** This is the base level of the CLK i.e if a tick pulls to a logic level 1, the baseline is 0 and if the tick pulls to 0, the baseline is 1.
  - **phase:** Whether the data is clocked on the rising edge or falling edge of a CLK tick. This is set to 0 or 1 respectively.
  - **bits:** The number of bits per word of data sent.

3.5.3 Default Interrupt File Template

This template describes the base settings for any interrupt handler. An additional function must be defined in <interrupts_functions.py> alongside this file entry.

```
{"interrupt_function": "",
  "period":0,
  "duration":0,
  "pins":[ ],
  "arguments":[ ]
}
```

- **interrupt_function:** This refers to the name of the interrupt, as mapped using the dictionary in the interrupt.py file.

- **period:** This describes how often the task has to run, in nanoseconds. It can also be considered the deadline of the interrupt handler.
- **duration:** This describes how long a task takes to run in nanoseconds.
- **pins:** This is a list of all the pins the interrupt will need access to.
- **arguments:** These are any values that should be passed to the interrupt handler function.

3.6 Simple Example Files

As described in section 3.4.5, the scheduler reads in every task file and where data is missing, it inserts the default data settings. This allows the user to only have to define the unique behavior of tasks, devices, interrupts etc. The purpose of these examples is to show how only a few fields need to be changed to define an entry. The following examples use the example default files shown above as the default settings.

3.6.1 Example Devices

The following setup is used as an example. A peripheral device for communication uses I^2C, with an address of 33 and a baud of 5000. The SDA is connected to the D12 of the mainboard and SCL pin is connected to the D13 pin of the mainboard. There is also a UART connection with RX (*TX on mainboard*) going to pin D16 and TX(*RX on mainboard*) connected to D15. The baud is 9600 and the buffer size is 128 bits. It can be assumed the rest of the settings are the default standard. The payload uses all default settings.

As the communication I^2C uses the default pins, these do not need to be redefined. Only the address and baud are changed by providing the entries in the file definitions. For the communication UART new pins are used and so these entries are redefined. Similarly the buffer size is redefined to a bigger value. As SPI is not used, no entry is provided and the default will be used. However, as the user is not expected to be writing tasks for the communication device using SPI, this should not present an issue. As the payload uses all the default settings, an empty entry is provided.

```
{"Communication":{
    "I2C":{"address":33, "baud":50000},
   "UART":{"RX":"D15", "TX":"D16", "receiver_buffer_size":128} }
 "Payload": {}
}
```

3.6.2 Example Tasks

- **Write Task**
  For this example, data is being written to a communication board. The communication board sends data out every 6 milliseconds. Sending 20 bytes of data to the communication board takes

1 millisecond. It uses I^2C and needs to be padded by 1 byte. The data size value provided does not include this padding value. The data to be sent should be loaded from the buffer, starting at the address 12.

In the file, the duration and period are redefined to suit this task, in nanoseconds. The default settings do not send, read, load or write data. Therefore the send value must be set to true in the connection fields. Similarly the data size must also be given. The front data padding in the connection_settings field is set to 1 as per the requirements. To get the data to be sent from the mainboard buffer, the load_settings are changed, setting the load from the buffer value to be true and giving the address start and end *(12 and 32 (12 + 20 bytes))* values.

```
{ "write_coms":{
                "duration": 1000,
                "period":   6000,
                "connection" :{
                              "device_name":"Communications",
                              "type": "I2C",
                              "send" : true,
                              "data_size" 20
                },
                "connection_settings" : {"front_data_padding" : 1},
                "load_settings" : {
                                   "load_from_buf": true,
                                   "load_address_start" : 12,
                                   "load_address_end" : 32,
                }
}}
```

- **Read Task:**
  For this example, data is being received from a sensor on a payload. The mainboard has to request the data via I2C. The sensor data is 20 bytes and be should be written to the buffer, starting at the address 12.

  In the file, the duration and period are redefined to suit this task. The duration is set to null, which will let the mainboard give an estimate for the duration. The read value must be set to true in the connection fields. Similarly the data size must also be given. To get the data received to be stored to the mainboard buffer, the store_settings are changed, setting the store_to_buf value to be true and giving the starting address value, 12.

```
{"read_sensor":{
                "duration": null,
                "period":   6000,
```

```
                "connection" :{
                                "device_name":"Payload",
                                "type": "I2C",
                                "receive": true,
                                "data_size" : 20,
                },
                "store_settings":{
                                  "store_to_buf": true,
                                  "store_address" : 12 ,
                }
}}
```

- **A Combination of Tasks:**
  For the examples above, these two tasks could be combined and run using the pass-through function, reducing the need to store in a buffer.

```
{"read_payload_write_coms":{
                "duration": 1000,
                "period":   6000,
                "connection" :{
                                "device_name":"Payload",
                                "type": "I2C",
                                "send" : true,
                                "data_size" 20
                                pass_through:" true
                                pass_location: "Communications"
                },
                "connection_settings" : {"front_data_padding" : 1}
}}
```

3.6.3 Example Custom Function

For the custom function example, data previously read from a sensor into the mainboard buffer must be manipulated before being sent to the communication board. While this reduces the separation of concerns, where the communication board should perhaps be performing the data manipulation, it serves to show that for peripherals with limited resources, computational tasks can be offloaded to the mainboard. As lists in Circuit Python are passed by reference, the buffer can be edited in the custom function without needing to return the data buffer. However, to save time, the function is expected to return the data.

The func_dict in the custom_function settings is updated to include an entry for the custom function name. The custom and before fields are set to true, so the custom function runs before the data is

sent to the communication payload. The name “process_sensor” is used in the custom function name dictionary. The arguments are set to 12 and 20 for the start buffer value and length. The period is set to 4000 nanoseconds and the duration set to “estimate”, so that the mainboard will estimate how long the task is. The time value in custom_function settings must be be included for this estimate, which in this case is 300 nanoseconds.

```
    {"write_screen_data":{
                "duration": "estimate",
                "period":   4000,
                "connection" :{
                              "device_name":"Communications",
                              "type": "UART",
                              "send" : true,
                              "data_size" 20
                },
                "custom_function": {
                                "custom" : true,
                                "before" : true,
                                "custom_function_name": "process_sensor",
                                "arguments" : [12, 20],
                                "time": 300
                                }
}}
```

3.6.4 Example Interrupt

Like custom functions, the interrupts have to be defined in a separate file, <interrupts_functions.py>. In the case of this interrupt, all the parameters were set. Generally the default interrupt file will probably not be used, as there will be different pins to poll for incoming signals, a different function and different duration and period. The example function is a simple function which takes a start and end address and replaces all the values in a buffer with a provided value. The custom function name is changed to the string in the dictionary in <interrupt_functions.py file>. The pin that is checked is D17 and the arguments the handler would use are included. In this case the values between buffer index 13 and 34 would be replaced to the value 0.

```
{ "interrupt_function": "clear_buf",
  "period":10000,
  "duration":5000,
  "pins":[D17],
  "arguments":[12,34,0]
}
```

3.7 Testing

As discussed in section 3.2.5, a complete test suite could not be created. This was in part due to late construction of physical systems but also due to the nature of the style of software created. The tests could have been run on a main computer using the full Python distribution. This would have allowed for mocking of each library and full access to testing libraries available to Python. However, these tests could not be run using Circuit Python and there would not be a guarantee these tests would catch all the issues in Circuit Python. Additionally, using a computer and Python would not allow for on-board testing. For CubeSats, on-board testing allows for checking of system data for errors that maybe introduced due to radiation errors. While discussed in section 2.3.2, radiation is not usually an issue, this is still an added benefit for future considerations. In this project, the tests are run on before creating the initialising the scheduler in <main.py>.

<mono_sch.py> has very a series of test cases. The tests are functional tests ensuring that series of tasks that mathematically should be able to be scheduled with RMS are scheduled and another set that are not mathematically possible to be scheduled are not. There are also tests checking handling of empty dictionaries and non-dictionary objects passed as arguments. The schedule function crashes with incorrectly formatted JSON data. To avoid over-complicating and bloating the code with error handling for all of JSON formatting issues, the burden of incorrectly formatted JSON data currently falls on the user to fix. This can be considered an acceptable solution, as trying to handle or correct these errors may result in the errors being corrected to a different behaviour than what the user intended.

To test the <main_scheduler.py> class, a mock <com.py> library was created. This library returns either a list of ‘a’ characters for SPI reads, a list of ‘b’ characters for UART reads and a list of ‘c’ characters of I^2C read. This mock library is not a standard mock library, as mocking is not available in Circuit Python. Instead, to use this library, a testing function was added to <main_scheduler.py>. This function replaces the global pointer to the <coms.py> import. This then allows for the communication read methods to be mocked or more accurately temporarily replaced. The write functions could not be mocked, as outgoing signals would have to be verified to be the right pattern. Using this basic mocking, test cases then check that each type of communication task could be run, alongside storing to buffers and files. Custom functions and interrupts were also tested using a similar mocking method. The main loop of the scheduler could not be checked as this ran forever, so instead only run_task() was used in test cases. As Circuit Python does not have a proper test suite or timeout function, testing run_schedule(), which runs forever, was tested by ad hoc running of code.

The communication helper functions in <coms.py> could not be tested using mocking, as actual data must be read and written to check if they are working. Instead once the physical breadboard system in section 3.3 was constructed, a series of physical tests were run using the ESP32 on the demo boards. These consisted of sending data between the demo boards and the mainboard. While this helped catch bugs, this solution is far from ideal. It does not support the open source testing ideals, as the system must be reconstructed for someone to change any code and test it.

4 Results

4.1 Demonstration Project

As the PCBs were not completed by the time of the report deadline, the demonstration program was run entirely on the breadboard designs. The example <main.py> in section 3.4.6 was used on the Circuit Python mainboard. Circuit Python, after booting up, automatically runs any files called <main.py>. This program is generic and can be used by other users without any change, with the exception of the accompanying configuration files. A flow diagram of the system is included in fig. 21 and all of the accompanying code can be seen in appendix C.1. The circuit and setup used matches the breadboard FlatSat in fig. 11.

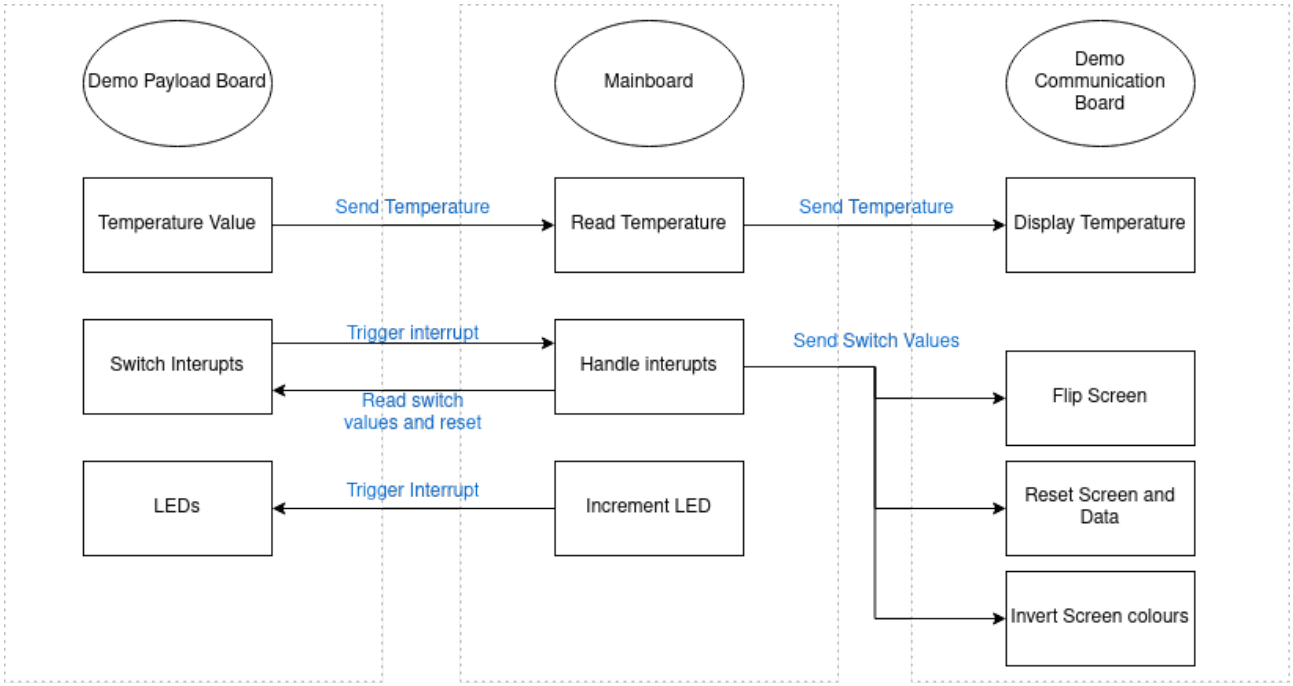


Figure 21: Demonstration System Flow Diagram

4.1.1 Demonstration Payload Board

A payload class was created to represent this demonstration board. In addition to the benefits of object encapsulation, this will allow future users to modify the board on a class level or perhaps create a subclass from this. This should help in future work. The payload class has three main features:

- **Temperature Values:** This is the primary purpose of the payload. It acts as a very simple sensor and shows how data can be sent from a payload to a communication board via the mainboard. To calculate the temperature, a thermistor library and class was created for Circuit Python, <therm.py> and can be seen in appendix C.1.3. In the case of this demonstration board a 10k thermistor is put with a 10 kOhm resistor in a voltage divider configuration [57]. This is used to find the value of the thermistor at that temperature. The formula to estimate the temperature is a simplified Steinhart-Hart equation [58].

$$R_{therm} = R_{ref} * (\frac{V_s}{V_o ut})$$

Voltage Divider equation

R_{therm} = Thermistor Resistance, R_{ref} = Reference Resistor, V_s = Source Voltage, V_{out} = Output Voltage (*at midpoint in circuit*)

$$\frac{1}{T} = \frac{1}{T_o} + \frac{1}{\beta} * ln(\frac{R}{R_o})$$

Steinhart - Hart equation

T = Measured Temperature, T_o, β = Thermal Constant of Thermistor, R = Thermistor Resistance at T, R_o = Reference Temperature of Resistor

- **Interrupt Driven LED Counter:** This is a feature to show how the demonstration board can handle interrupts. A pin on the ESP32 is set to a hardware interrupt and anytime the incoming signal is high, the interrupt triggers. In the case of this payload, the LEDs display a number in binary and the interrupt triggers an increment of this binary number. The count rolls over when 7 is reached, which when all LEDs are on.
- **Control Switches for Communication Board:** This is a feature to show how commands to other boards can be sent using the mainboard. It also shows how the interrupt handler works on the mainboard. The switches trigger on a rising or falling edge. This allows them to be flipped in either direction to trigger the function. When flipped, they set a specific location in the payload's I^2C memory space to be the value of 1. The mainboard regularly reads these locations. After the addresses have been read, the mainboard generates an interrupt in order to clear the switch values currently stored to 0.

4.1.2 Demonstration Communication board

The demonstration communication board focuses on using the LCD screen included in the board's design to present data that would be sent out to Earth from the CubeSat. A class called Coms was made to handle this. Loborris firmware provides drivers for running screens that use the ILI9341 chip, which is used in the chosen screen. A plotting function was written to give a more complex display than just text values. The temperature values come via I^2C from the mainboard, after being read from the payload demonstration board. The communication board watches for this value and when it is received, the plot on the screen is updated. A record of the current value and highest and lowest values are also maintained.

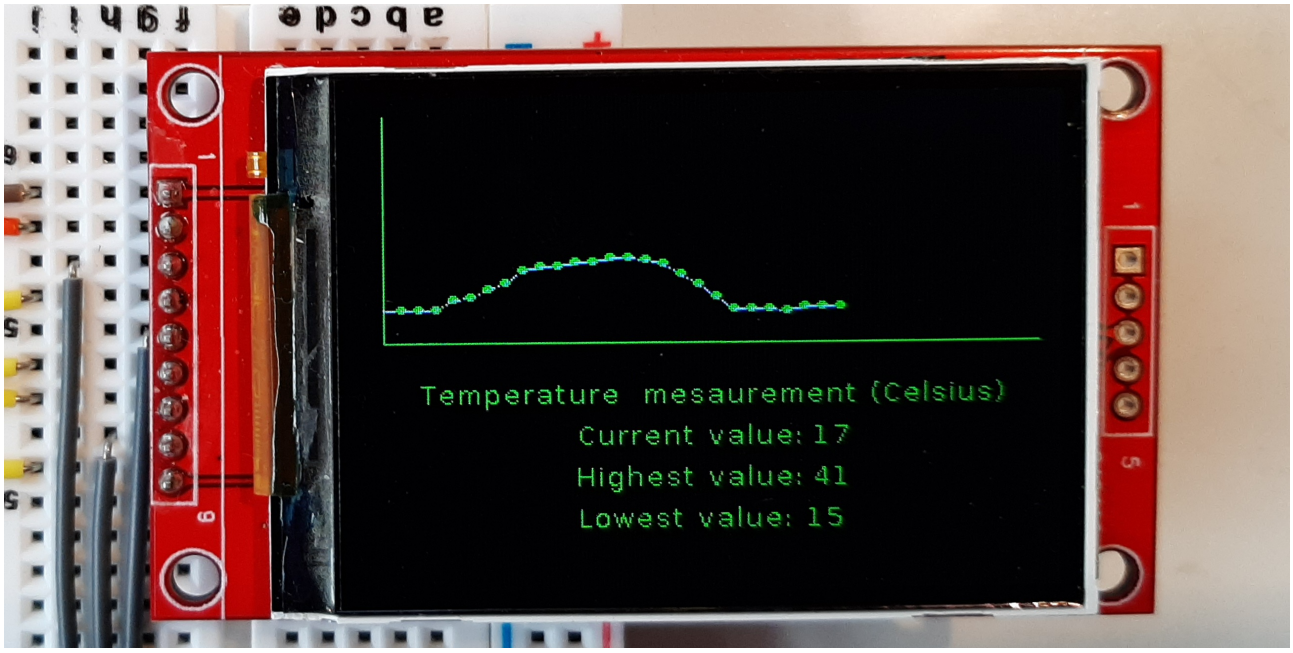


Figure 22: Image of Plot Displayed on Demonstration Communication Board

Alongside this, the communication board watches the incoming UART data for a series of values. These are the signals transferred from the switches on the demonstration payload board, by an interrupt handler on the mainboard through UART to the communication board. Each value performs a different function and only performs this when the value is 1. The first value flips the screen 180 degrees from its current orientation. The second value completely resets the plot and stored values on the communication boards. The last value inverts the screen's colour scheme. UART was used instead of continuing with I^2C, as it demonstrates the use of another communication protocol on the mainboard and demonstration boards.

4.1.3 Mainboard Tasks

In order to demonstrate the scheduling ability of the system, the mainboard was provided with a series of tasks to interact with the two demonstration boards. All these task definitions can be seen in appendix C.2.1.

- **Reading Temperature:** The mainboard reads the thermistor temperature from the demonstration payload via I^2C. The same task then uses the pass through feature to pass the data onto communication without needing to store the value.
- **LED Increment:** This is a simple task, which sets a pin high. The pin going high triggers an interrupt on the connected demonstration board which handles LED increments.
- **Interrupt Handler:** The mainboard has an interrupt handler task which is scheduled more regularly than the other tasks. This handler polls the incoming interrupt pin from the demonstration payload board. If it is set, the mainboard reads the data state of the three switches and passes them on via UART to the communication board.

4.2 Other Example Uses

While the demonstration project was the primary test case for this project, some other work was carried out to demonstrate other potential use cases. The system was built with the demonstration boards in mind. By adding newer components after everything had been coded and built, it briefly demonstrates the system is a viable solution for other types of CubeSat components that the system was not explicitly designed for.

4.2.1 SPI Device

As the demonstration project did not include SPI due to the limitations of ESP32, the SPI capabilities were not a main focus of design. However, following the completion of the system, a demonstration with an SPI device and the mainboard was carried out. The device was a thermocouple [59] and it relayed the current temperature of the system, via SPI. The task configuration for this just required reading data from MOSI lines and providing slave select lines to set up device configurations. In order to use the thermocouple, the startup function in <main_sch.py> was used. This was needed, as a series of bits had to be configured on the thermocouple when turning on, before data can be received. The data returned was then edited using a custom function.

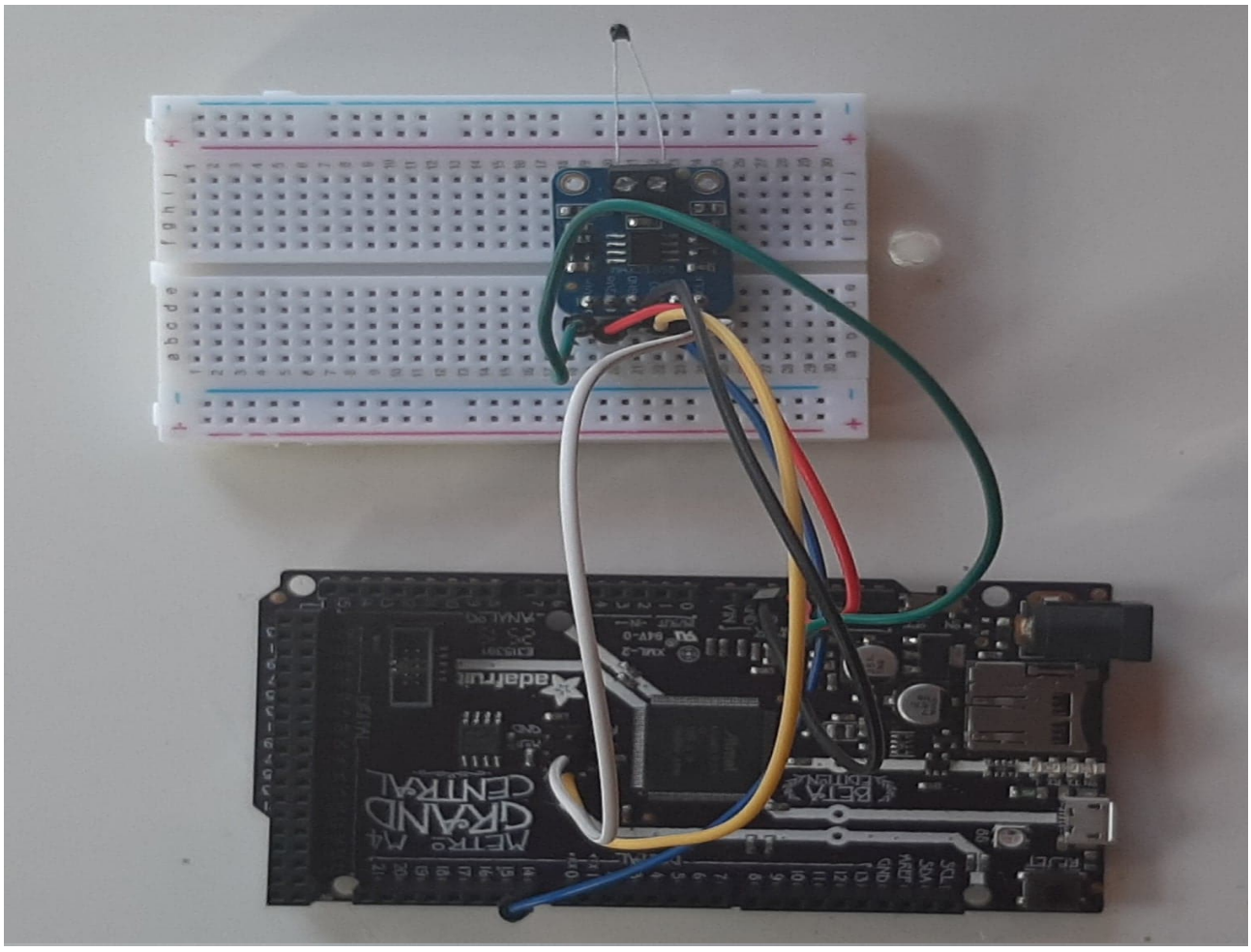


Figure 23: Image of SPI Thermocouple

4.2.2 CAN Communication

The PyCubed mainboard does not natively support CAN. As mentioned in section 2.2.5, CAN is currently being considered as a viable standard for CubeSat communication. To try and provide the possibility of using CAN in the future for these test boards, a quick prototype was developed. The prototype uses an external CAN transceiver. This transceiver is available as individual IC components, but a breakout board was used alongside the already existing breadboards for ease of development.

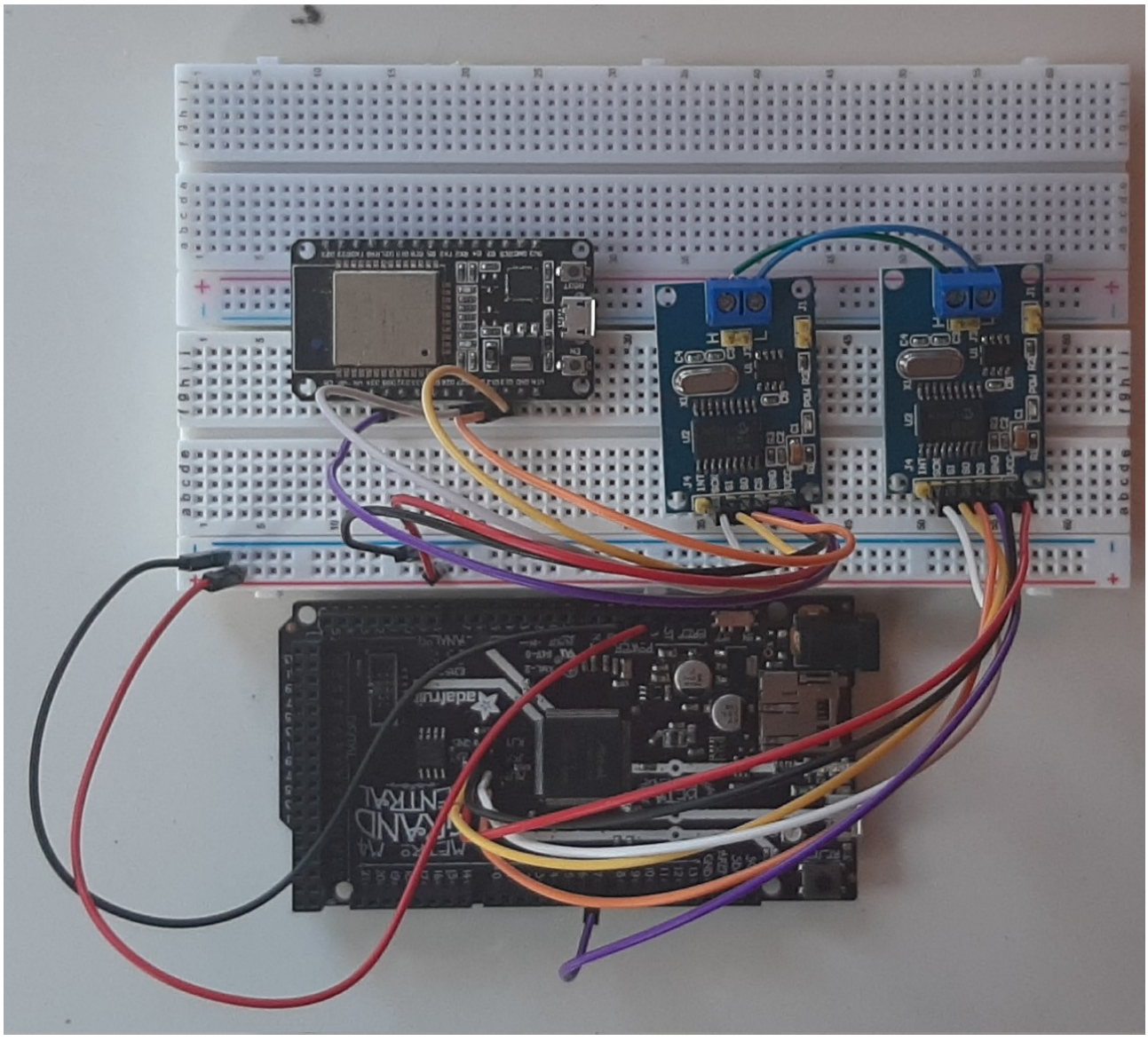


Figure 24: Image of Prototype CAN Setup

This CAN transceiver uses SPI to communicate with peripheral devices [60]. While this means CAN is not a built in feature of the system and technically uses SPI, it at least provides the basis for further development of devices that may use CAN, without the need to redesign the current system. The ESP32s can also communicate with the CAN bus using SPI, allowing for the demonstration boards to also have access to CAN. The CAN buses do require initial configuration, which is a series of data words sent via SPI.

5 Further Discussion

5.1 Goals Achieved:

The project had three main goals when it started out:

- **Development of Test Bed:** This was the primary goal of the project and the one that has been most successfully achieved. A FlatSat for testing PC104 cards was developed, including test points, connections to other FlatSats, and power supplies. This should be compatible with a range of projects and eliminate ad hoc wiring of connections. Alongside that, a space ready mainboard and other hardware was assembled for use in testing other components. To complement this, software was written allowing the users to schedule tasks and emulate CubeSat behaviour for use in testing and verification of designs for CubeSat components.

- **Improvement of Open Source Resources:** PyCubed is an open source platform that formed the basis of this project. While the PyCubed firmware and designs were not changed, new software suitable for running on the PyCubed designs was developed. Alongside this, an open source FlatSat design and two open source boards that serve as emulations for CubeSat communication and Payload boards were created. All of this is hosted on a GitHub. When the PyCubed board is fully assembled and the software has been run on a PyCubed board (*instead of the Adafruit breakout board*), a push request of all the designs and code developed during this project will be made to the PyCubed GitHub adaptions directory.

- **Development of CubeSat Ecosystem:** This goal was not really achieved by this project. Due to limited interaction with members of the ISET department and unavailability of suitable projects in the department, it was not really possible to demonstrate the use of this system as a CubeSat platform. The system has been shown to work with the demonstration payload and communication setup. This serves as a proof of concept for further development.

5.2 Future Work

This project is also quite open ended, so there was always going to be scope and room for more work. While the project was successful, there are a few areas that could be enhanced, improved and developed:

- **Completed System:** As explained in the COVID-19 forward and section 3.3, the software was never run on the PyCubed mainboard as the board construction was not finished. While the same processor and similar firmware was used on a breakout board for prototyping, this does not fully validate the system use case. Given more time or better circumstances, this board would have been assembled and software verified to work. Due to the similarities of the prototypes there should not be any issues, but this cannot be guaranteed until the code is run on the board. This is hopefully a relatively simple next stage of this project and should be completed before any other future work is carried out.

- **Improvement on Scheduling:** There are issues with the implementation of scheduling in this system, mainly a lack of priorities and interrupts. The lack of priorities as discussed in section 3.2.2 was done to reduce the complexity and reasoning for the user. However, in a real implementation of CubeSats not all tasks are considered equal. When there is timing conflict, priority is given to mission critical tasks, that ensure the CubeSat can continue its orbit. If an important task of something relating to power or altitude control is generated, this tasks logically needs to take precedence over something like processing of sensor data or transmitting signals down to Earth.

  Every task is assumed to have a hard deadline. A hard deadline suggests every task must be completed by their deadline. This allows for tasks to simply be considered on a linear timeline, and creates no issues about checking which task to run next. However, certain tasks like reading sensor data can be pushed out or missed every so often with no detrimental effect. This is a soft deadline, where the value of the task decreases with a delay but the system can still function adequately. Having soft and hard deadlines complements being able to schedule with priorities.

  Finally, once the scheduled timeline is created, it can not be edited. This means tasks cannot be removed or generate dependant on the system status. Very seldom run tasks must be included in the original timeline, which can lead to a very large time period until the timeline repeats itself. This is the case for triggering solar panels to begin charging on each orbit among other tasks. Should the data received from sensors prompt new tasks to be created and uploaded from Earth, this cannot be added to the system without a complete reboot and rescheduling. With more complex operating systems, scheduling new and user tasks like this is possible.

  While these have no real effect on the testing of components, they do affect the viability of using this designed ecosystem in space. If an entire system was developed using the system design, changes would have to be made. However, adding all of these changes will increase the burden of knowledge needed by the user. A heavily modified RMS scheduler would be needed if not a new algorithm altogether. There may be a need to change the system language. Changing the language would further alienate new users to the system. It may be that to keep the system easy to use for testing and new users, it is not possible to make it a complete suite viable for complex CubeSat systems. A suitable alternative maybe developing the test system to run all the user designed sub-systems and leaving mission critical hardware like ADCS and EPS under a different system.

- **Interrupts:** The lack of interrupts follows on from the scheduling issues. The system currently uses pseudo interrupt handlers, which poll certain incoming signals regularly. However, this is not very elegant compared to hardware triggered interrupts. It has a poor response time and greater CPU usage in comparison to proper hardware interrupt handlers. But by using RMS and scheduled interrupt handlers, it is simpler and easier to visualise on a timeline than interrupts which can appear and disrupt the current task. In addition to this, hardware interrupts would require a context switching mechanism and more memory management. Again, like with scheduling, this falls into the trade off of a simple to use system and a complex complete system.

- **CAN:** As discussed in section section 3.1.2 CAN is not included in the PyCubed mainboard. It would be been beneficial to have CAN integrated directly into the system. Currently, it requires a peripheral device which uses SPI. At a minimum, a CAN PC104 board could be created, with a whole series of SPI based devices like in section 4.2.2. Any device could be connected to one of these CAN devices on the board and use them for CAN communication. A better solution would be the development of CAN libraries for both the PyCubed and the ESP32 devices, alongside the integration of a CAN transceiver to each device's design. This would appear to the user just as a communication option in the task entry and remove the need for the knowledge required to set up the CAN device.

- **More Task Options:** While comprehensive, a complete suite of task options was not generated. These can be considered edge cases of the tasks, but still common enough that they maybe used:

  - **Different custom functions before and after a task:** Currently only one function is available for either before, after or both. This is overcome by including a flag in the custom function code to see if the function is before or after, however this is a hacked solution and requires more coding knowledge that should be expected of the user.
  - **Separate communication protocols for pass-through:** Currently the pass-through device protocol is not defined and assumes to use the main communication's protocol. For demonstration boards that have the same hardware, this is not an issue. However there could be cases when ports of a communication type on the mainboard or demonstration board are used, requiring a different protocol to be used. This is alongside the more common issue of certain peripherals not having a certain communicate hardware available.
  - **Option to send out the same message to multiple devices in the same task:** Currently system tasks allow for only one device to be used for reading, writing or pass through. For tasks were data has to be collected from multiple devices or were data messages need to be sent to multiple devices, copies of tasks need to be made, with only the device name changed. It should be possible to use a list structure or similar to handle multi-device writing and reading in the configuration file and then run communication functions for each device.
  - **Protected memory:** Currently all tasks can access all files and all of the buffer space. This can lead to dangers of overwriting data or files, if strict data management is not maintained. This is especially the case with the buffer, where the end buffer size may be dynamic, allowing the user to overwrite further than they expected. To avoid buffer overwriting, it should be possible to assign each task a set block of the buffer, to which they are only allowed to access. This could also be done with filenames or similar.
  - **Context switch:** As a follow-on to protected memory, it could be possible to give all tasks virtual memory instead. This would compliment protected memory and reduce the user's need to manage address locations. The address management would be handled behind the scenes by the scheduler class without any considerations on the user's side. A simple

version would add an offset to each tasks and set memory size. More complex virtual memory, like dynamic allocation and random access, could also be possible. However, this may complicate the internal code of the system for little additional gain.

- **Testing:** Full mock tests would be a beneficial addition to the current test suite. As explained in section 3.7, some tests were done to validate <main_sch.py> and <mono_sch.py>. An alternate mock solution was created, but is not fully comprehensive. All of the communication protocols could not be tested without other physical components. This is not a great test practice, as they become reliant on test devices and only show the tests work with those devices. It also diminishes the benefit of open source testing, as any user wanting to change the software, has to replicate the system in order to ensure the old tests pass and no new bugs are introduced. To fully demonstrate the helper functions and mainboard communication perform correctly, some form of proper mocking would have to be carried out.

- **Integration with University Projects:** This was an initial aim of the project, however with COVID-19, this was no a viable option at the time. The system is supposed to be a test environment for future and current projects by the ISET. This would provide the opportunity to get direct feedback on the systems functionally and usability, which could be used as basis for future iterations improvements.

- **Development of system on other boards and devices:** While other current open source devices proved not to be viable, it would be beneficial to develop a system agnostic software. If OreSat becomes viable again in the future, this could be an option or even just using a private sector board. While this would reduce the value as an open source project, it may increase the overall usability of the system. It would allow the user to make hardware decisions without concern for compatibility with software. This would possibly require the project to be redefined at a lower language level, which reduces the customisation an inexperience user could do when compared to the simplicity of Python.

- **More Adaptor cards:** Currently using existing boards require basic PCB knowledge to create adaptor plates for any other third party board. There are at least three generic adaptor cards that could be very useful in the future. The first would be a breadboard style adaptor, allowing the user to wire a prototype to certain connections. The other two boards relate to PC/104 connections. The first would be an adaptor from the standard PC/104 connector to a PC/104 plus connector. The second would be an adaptor from 4 rows of 26 pins to the PC/104 Plus connector. The 4x26 pins are a common alternative modification of the PC/104 Plus.

- **Interactive User Interface:** This is a tertiary final step of the project, which may help with use of the system by inexperienced uses. A program with a series of text fields and drop down menus could help eliminate the learning curve completely. It would remove the need for the new user to understand JSON and perhaps provide features like auto-generating default settings.

6 Conclusion

Overall, this project can be regarded as a success. While not everything was achieved, a large percentage of this project was. The project ultimately sought to provide a solution to help with the development of future CubeSats by the University of Glasgow ISET department. This project reduces the design overheard involved in rapidly prototyping a system through multiple avenues: a FlatSat, a mainboard and battery board suitable for space, a software scheduler and emulator, and demonstration boards.

A FlatSat was designed to simplify construction of prototype boards. It allows for multiple cards to be placed beside each other as opposed to stacked. There are connections for power and additional connections on the side, either for diagnostics or connection to more FlatSats. This FlatSat is wired with connections to all PC/104 Plus connectors included, allowing full access to all signals and reducing the need for ad hoc prototype connections and boards for testing.

A hardware mainboard was built. This mainboard has been used in previous successful space missions and is completely open source. The mainboard is from the PyCubed project and runs Circuit Python. The simplicity of the Python language allows for rapid coding and software development, for both experienced and inexperienced designers alike. An adaptor plate was made to allow this board to work with the FlatSat. The PyCubed battery board was also manufactured.

Software is provided for the mainboard, allowing tasks to be scheduled with limited programming and system knowledge. Communication, interrupt handlers and custom functions can all be run, with a few lines of text in configuration files. Each configuration can be saved for re-running on the board at a later time. For each test and iteration, no code needs to be changed, just the configuration files. However, the option to change the code and add more functionality is also available if needed.

Alongside the hardware mainboard, two demonstration boards were constructed. One board represented a payload board with sensors and input and the other represents a communication board with a screen output. These boards serve to demonstrate the hardware and software capabilities of the system. Both boards can also be reprogrammed to perform other functions or emulations of other CubeSat subsystems. They could also be used a tutorials to introduce users to the system and serve as a starting point for future projects.

While this did not become the full CubeSat ecosystem that was initially envisioned, it is a strong prototype. Further work has to be done to bring it to a complete system. But the final system certainly serves as a proof of concept, if not the basis of a viable system.

To conclude, a software and hardware system was constructed to emulate the main computer of a CubeSat. Programming the system is done with JSON based text files which describe the tasks to be carried out. A physical test bed was made, to help further simplify testing and two hardware boards were provided to help emulate other components of a CubeSat. There is room for improvement, but hopefully this is an open source system that in its current state can be used to help design, test, and improve projects by the University of Glasgow and further a field.

References

[1] A. Johnstone, *Cubesat design specification*, 14th ed., The CubeSat Program, 2020.

[2] *Cubesat101 basic concepts and processes for first-time cubesat developers*, NASA, 2017.

[3] A. Scholz and J.-N. Juang, "Toward open source cubesat design," *Acta Astronautica*, vol. 115, pp. 384–392, 2015.

[4] A. Toorian, K. Diaz, and S. Lee, "The cubesat approach to space access," in *2008 IEEE Aerospace Conference*, 2008, pp. 1–14.

[5] *Nanosats database*, Last Accessed 23/9/2020. [Online]. Available: `https://www.nanosats.eu/`.

[6] A. Klesh *et al.*, "Marco: Early operations of the first cubesats to mars," in *Small Satellite Conference*, Jul. 2018, SSC18-WKIX–04.

[7] *About clyde space*, Last Accessed 8/9/2020. [Online]. Available: `https://www.aac-clyde.space/about-us`.

[8] S. Waydo, D. Henry, and M. Campbell, "Cubesat design for leo-based earth science missions," in *Proceedings, IEEE Aerospace Conference*, vol. 1, 2002, pp. 1–1.

[9] F. Davoli *et al.*, "Small satellites and cubesats: Survey of structures, architectures, and protocols," *International Journal of Satellite Communications and Networking*, vol. 37, no. 4, 2018. DOI: `https://doi.org/10.1002/sat.1277`.

[10] X. Xia *et al.*, "Nanosats/cubesats adcs survey," in *2017 29th Chinese Control And Decision Conference (CCDC)*, 2017, pp. 5151–5158.

[11] A. PoghosyanAless and A. Golkar, "Cubesat evolution: Analyzing cubesat capabilities for conducting science missions," *Progress in Aerospace Sciences*, vol. 88, pp. 59–83, 2017. DOI: `https://doi.org/10.1016/j.paerosci.2016.11.002`.

[12] C. Nieto-Peroy and M. Reza Emami, "Cubesat mission: From design to operation," *Applied Sciences*, vol. 9, p. 3110, 2019.

[13] B. Klofas and K. Leveque, "A survey of cubesat communication systems: 2009–2012," in *10th Annual CubeSat Developers' Workshop*, Apr. 2013.

[14] B. Klofas, "Cubesat communications update," 2014, [Online]. Available: `https://www.klofas.com/papers/ESG-P-14-165-CubeSat_comm_update.pdf`.

[15] J. Bouwmeester, M. Langer, and E. Gill, "Survey on the implementation and reliability of cubesat electrical bus interfaces," *CEAS Space Journal*, vol. 9, pp. 163–173, 2017. DOI: `10.1007/s12567-016-0138-0`.

[16] *Unisec presentation on standardisation of cubesats*, Last Accessed 9/10/2020. [Online]. Available: `http://www.unisec-global.org/pdf/uniglo3/day3_0910-0920.pdf`.

[17] A. Scholz *et al.*, "Open source implementation of ecss can bus protocol for cubesats," *Advances in Space Research*, vol. 62, no. 12, pp. 3438–3448, 2018. DOI: `10.1016/j.asr.2017.10.015`.

[18] *History of can*, Last Accessed 14/10/2020. [Online]. Available: https://www.can-cia.org/can-knowledge/can/can-history/.

[19] "Space radiation effects on electronic components in low-earth orbit," NASA, , UT, Tech. Rep. PRACTICE NO. PD-ED-1258, Apr. 1996.

[20] R. Kingsbury *et al.*, "Tid tolerance of popular cubesat components," in *2013 IEEE Radiation Effects Data Workshop (REDW)*, 2013, pp. 1–4.

[21] J. Straub *et al.*, "Openorbiter: A low-cost, educational prototype cubesat mission architecture," *Machines*, vol. 1, pp. 1–32, Mar. 2013. DOI: 10.3390/mach1010001.

[22] J. Straub, "An update on the openorbiter i mission," in *2017 IEEE Aerospace Conference*, 2017, pp. 1–6.

[23] D. Geeroms *et al.*, "Ardusat, an arduino-based cubesat providing students with the opportunity to create their own satellite experiment and collect real-world space data," in *22nd ESA Symposium on European Rocket and Balloon Programmes and Related Research*, ser. ESA Special Publication, vol. 730, Sep. 2015, p. 643.

[24] N. Chrona, "Msc thesis on upsat," Last Accessed : 23/9/20 Available here: https://nchronas.github.io/upsat_msc_thesis, M.S. thesis, University of Patras, Dec. 2017.

[25] *Upsat website*, Last Accessed 28/09/2020. [Online]. Available: https://upsat.gr/.

[26] *Equisat website*, Last Accessed 20/09/2020. [Online]. Available: https://brownspace.org/EQUiSat-resources/.

[27] M. Holliday *et al.*, "Pycubed: An open-source, radiation-tested cubesat platform programmable entirely in python," in *Small Satellite Conference*, Jul. 2019, SSC19-WIII–04.

[28] A. Greenberg *et al.*, *Nasa csli application for oresat: Oregon's first nanosatellite*, Nov. 2016.

[29] *Oresat website*, Last Accessed 10/10/2020. [Online]. Available: https://www.oresat.org/.

[30] *Pycubed website*, Last Accessed 9/12/2020. [Online]. Available: https://www.notion.so/PyCubed-4cbfac7e9b684852a2ab2193bd485c4d.

[31] *Pycubed github*, Last Accessed 9/12/2020. [Online]. Available: https://github.com/pycubed.

[32] *Oresat website*, Last Accessed 9/12/2020. [Online]. Available: https://www.oresat.org/.

[33] *Upsat github*, Last Accessed 12/10/2020. [Online]. Available: https://github.com/oresat.

[34] *Upsat website*, Last Accessed 23/09/2020. [Online]. Available: https://upsat.gr/.

[35] *Pycubed github*, Last Accessed 23/09/2020. [Online]. Available: https://gitlab.com/librespacefoundation/upsat.

[36] *Link to equisat google drive*, Last Accessed 23/09/2020. [Online]. Available: https://brownspace.org/EQUiSat-resources/.

[37] *Equisat github*, Last Accessed 20/09/2020. [Online]. Available: https://github.com/BrownSpaceEngineering.

[38] *Comparison table of m cortex processors - arm resources*, Last Accessed 04/10/2020. [Online]. Available: https://developer.arm.com/-/media/Arm%5C%20Developer%5C%20Community/PDF/Cortex-A%5C%20R%5C%20M%5C%20datasheets/Arm%5C%20Cortex-M%5C%20Comparison%5C%20Table_v3.pdf?revision=077cd073-b4ff-4193-8a26-5e854aa6e431.

[39] *Circuit python documentation*, Last Accessed 23/11/2020. [Online]. Available: https://circuitpython.readthedocs.io/en/6.0.x/README.html.

[40] T. B. Chandra, P. Verma, and A. K. Dwivedi, "Operating systems for internet of things: A comparative study," in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, ser. ICTCS '16, Association for Computing Machinery, 2016. DOI: 10.1145/2905055.2905105.

[41] *Equisat power bugdet*, Last Accessed 9/12/2020. [Online]. Available: https://brownspace.org/EQUiSat-resources/#1458886750869-b1017fa5-b97b.

[42] *Pc/104 plus specification*, 2.6, Available here https://pc104.org/wp-content/uploads/2015/02/PC104_Spec_v2_6.pdf Last accessed: 8/12/2020, PC/104 Embedded Consortium, 2008.

[43] *Pc/104 plus specification*, 2.3, https://pc104.org/wp-content/uploads/2015/02/PC104_Plus_v2_32.pdf Last accessed: 8/12/2020, PC/104 Embedded Consortium, 2008.

[44] *Isis 16u cubesat structure*, Last Accessed 15/10/2020. [Online]. Available: https://www.isispace.nl/product/16-unit-cubesat-structure/.

[45] *Ipc-2221a generic standard on printed board design*, http://www-eng.lbl.gov/~shuman/NEXT/CURRENT_DESIGN/TP/MATERIALS/IPC-2221A(L).pdf Last accessed: 8/12/2020, Institute of Printed Circuit Boards, 2003.

[46] *Lcd module specification*, http://www.lcdwiki.com/res/MSP2202/QDTFT2201_specification_v1.1.pdf Last accessed: 8/12/2020, QDTech, 2020.

[47] W. Vanderbauwhede and J. Singer, "Process scheduling," in *Operating Systems Foundations*. ARM Education Media, 2019, pp. 94–101.

[48] *Mbed website*, Last Accessed 15/10/2020. [Online]. Available: https://os.mbed.com/mbed-os/.

[49] *Ardunio website*, Last Accessed 15/10/2020. [Online]. Available: https://www.arduino.cc/en/Guide/Introduction.

[50] *Micro-python website*, Last Accessed 25/11/2020. [Online]. Available: https://micropython.org/.

[51] E. Foster, "A comparative analysis of the c++, java, and python languages," Dec. 2014.

[52] *Loboris esp32 wiki*, Last Accessed 5/12/2020. [Online]. Available: https://github.com/loboris/MicroPython_ESP32_psRAM_LoBo/wiki.

[53] *Esp32 wroom 32d/esp32 wroom 32u datasheet*, https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf Last accessed: 8/11/2020, Esperiff Systems, 2020.

[54] K. Rembor, *Introducing the adafruit grand central m4 express*, Adafruit, 2020.

[55] R. Cayssials, J. Orozco, J. Santos, and R. Santos, "Rate monotonic scheduling of real-time control systems with the minimum number of priority levels," Jan. 1999, pp. 54–59. DOI: 10.1109/EMRTS.1999.777450.

[56] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973. DOI: 10.1145/321738.321743.

[57] J. D. Irwin and R. M. Nelms, "Voltage division," in *Engineering Circuit Analysis*. New York, NY: Wiley, 2015, pp. 38–40.

[58] J. S. Steinhart and S. R. Hart, "Calibration curves for thermistors," *Deep Sea Research and Oceanographic Abstracts*, vol. 15, no. 4, pp. 497–503, 1968, ISSN: 0011-7471. DOI: https://doi.org/10.1016/0011-7471(68)90057-0. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0011747168900570.

[59] *Max31855cold-junction compensated thermocouple-to-digital converter*, https://cdn-shop.adafruit.com/datasheets/MAX31855.pdf Last accessed: 8/12/2020, Maxim, 2020.

[60] *Mcp2515 stand-alone can controller with spi interface*, https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf Last accessed: 27/12/2020, MicroChip, 2020.

A Main project code

A.1 mono_sch.py

```
import json
import time

def lcm_2(x, y):
   if x > y:
       greater = x
   else:
       greater = y

   while(True):
       if((greater % x == 0) and (greater % y == 0)):
           lcm = greater
           break
       greater += 1

   return lcm


def lcm_arr(nums):
    lcm = nums[0]
    for num in nums[1:]:
        lcm = lcm_2(lcm, num)
    return lcm

#Schedule Tasks under monotonic task assumptions
def possible_schedule(tasks):
    if tasks == {} or type(tasks) != dict:
        return False
    timing = 0
    task_order = []
    n = len(tasks)
    u_b = n*(2**(1/n) - 1)
    u_t = 0
    for task in tasks:
        u_t += tasks[task]["duration"]/tasks[task]["period"]
        if (u_t > u_b) or (1 <= u_t):
            return False
    return True

def schedule_order(tasks, step=1):
    periods = sorted([tasks[task]["period"] for task in tasks])
    sch_period = lcm_arr(periods)
    mock_time = 0
    task_pattern = []
    next_deadline = {}
    scheduled = []
    unscheduled = []

    task_list = list(tasks.keys())
    task_list.sort()

    for task in task_list:
        unscheduled.append(task)
        next_deadline[task] = tasks[task]["period"]
        unscheduled.sort(key=lambda kv: next_deadline[kv])
    while mock_time < sch_period:
            for task in scheduled:
                if mock_time >= next_deadline[task]:
                    unscheduled.append(task)
                    next_deadline[task] += tasks[task]["period"]
                    unscheduled.sort(key=lambda kv: next_deadline[kv])
                    scheduled.remove(task)
            if len(unscheduled) != 0:
                next_t= unscheduled[0]
                del unscheduled[0]
                task_pattern.append((next_t, mock_time))
                mock_time += tasks[next_t]["duration"]
                scheduled.append(next_t)
            else:
                mock_time += step
    return (task_pattern, sch_period)
```

A.2 com.py

```
import board
import busio
from pins import pins
import digitalio

uart_parity = {"odd": busio.UART.Parity.ODD, "even": busio.UART.Parity.EVEN, None:None}

#Warning does not handle deep copies of dictionaries. Works for current file templates, at most there is two levels of dictionaries
def add_default_dict(data,default):
    for key in default:
        if type(default[key]) == dict:
            if key not in data:
                data[key] = default[key].copy()
            else:
                if type(data[key]) == dict:
                    data[key] = add_default_dict(data[key], default[key])
        else:
            if key not in data:
                data[key] = default[key]
    return data

def calculate_duration(task, devices, margin = 0.2):
    if task["duration"] != None:
        return task
    else:
        send_data_size = len(task["connection"]["data"])
        receive_data_size = task["connection"]["data_size"]
        duration = 0
        if task["connection"] == "I2C":
            baud = devices[task["connection"]["device"]]["frequency"]
            duration += 12700000
        elif task["connection"] == "SPI":
            baud = devices[task["connection"]["device"]]["baud"]
            duration += 113000
        elif tasks["connection"] == "UART":
            baud = devices[task["connection"]["device"]]["baud"]
            duration += 100000

        if task["load_settings"]["load_from_file"]:
            send_data_size = task["load_settings"]["file_data_size"]
            duration += (send_data_size * 400) + 550000

        if task["load_settings"]["load_from_buffer"]:
            send_data_size = (task["load_settings"]["load_address_end"]-task["load_settings"]["load_address_start"])
            duration += (send_data_size * 11100) + 49000

        if task["connection"]["receive"]:
            duration += baud * receive_data_size

        if task["connection"]["send"]:
            duration += baud * send_data_size

        if task["connection"]["pass_through"]:
            baud = devices[task["connection"]["pass_location"]]["baud"]
            duration += baud * receive_data_size

        if task["store_settings"]["store_to_file"]:
            if task["store_settings"]["append"]:
                duration += (receive_data_size * 29000) + 122000000
            else:
                duration += (receive_data_size * 10000) + 1093200000

        if task["store_settings"]["store_to_buf"]:
            duration += (receive_data_size* 11100) + 49000

        return duration *(1+ margin)

def Com_SPI(data = None, data_size=None, CLK = board.SCK, MOSI=None, MISO= None, slave = None, write_value = 0,  baud=100000, polarity
    =0, phase=0, bits=8,
            front_data_padding = 0, back_data_padding = 0, start= False, start_value = 0, end = False, end_value = 0, delimiter=""):

    spi = busio.SPI(CLK, MISO=MISO, MOSI=MOSI)
    received = None

    while not spi.try_lock():
        pass

    p = None
    if slave != None:
        p = digitalio.DigitalInOut(pins[slave])
```

```
        p.direction = digitalio.Direction.OUTPUT
        p.value = True

    if start:
        spi.write(bytes([int(str(start_value),2)]))

    if data_size != None and data:
        send = bytearray(front_data_padding) + bytearray(data) + bytearray(back_data_padding)
        received = bytearray(data_size)
        spi.write_readinto(received, send)
        if delimiter != "":
            received = str(res)[2:-1].split(delimiter)

    elif data_size != None:
        received = bytearray(data_size)
        received = spi.readinto(received, write_value=write_value)
        if delimiter != "":
            received = str(res)[2:-1].split(delimiter)

    elif data:
        send = bytearray(front_data_padding) + bytearray(data) + bytearray(back_data_padding)
        spi.write(send)

    if end:
        spi.write(bytes([int(str(end_value),2)]))

    if slave != None:
        p.deinit()

    spi.unlock()
    if slave != None:
        p.deinit()
    spi.deinit()
    return received

def Com_UART(data = None, data_size=None, baud=9600, bits = 8, parity=None, stop=1, timeout=1, receiver_buffer_size=64, TX = board.TX,
      RX = board.RX, front_data_padding = 0, back_data_padding = 0,  start= False, start_value = 0, end = False, end_value = 0,
      delimiter=""):

    uart = busio.UART(TX,RX, baudrate=baud, bits=bits, parity=uart_parity[parity], stop=stop, timeout=timeout, receiver_buffer_size=
      receiver_buffer_size)
    received = None

    if start:
        uart.write(bytes([int(str(start_value),2)]))

    if data_size != None:
        received = uart.read()
        if delimiter != "":
            received = str(res)[2:-1].split(delimiter)

    if data:
        send = bytearray(front_data_padding) + bytearray(data) + bytearray(back_data_padding)
        uart.write(send)

    if end:
        uart.write(bytes([int(str(end_value),2)]))

    uart.deinit()
    return received

#Reads data and then writes
def Com_I2C(addr, data = None, data_size = None, SDA = board.SDA, SCL = board.SCL, frequency=400000, front_data_padding = 0,
      back_data_padding = 0,  start= False, start_value = 0, end = False, end_value = 0, delimiter = ""):
    i2c = busio.I2C(SCL, SDA, frequency=frequency)
    received = None
    while not i2c.try_lock():
        pass

    if start:
        i2c.writeto(addr, bytes([int(str(start_value),2)]))
        #i2c.writeto(addr, bin(int(str(start_value),2)))

    if data_size != None:
        received = bytearray(data_size)
        i2c.readfrom_into(addr, received)
        if delimiter != "":
            received = str(res)[2:-1].split(delimiter)

    if data:
        send = bytearray(front_data_padding) + bytearray(data) + bytearray(back_data_padding)
        i2c.writeto(addr, send)

```

```
    if end:
        i2c.writeto(addr, bytes([int(str(end_value),2)]))

    i2c.unlock()
    i2c.deinit()
    return received
```

A.3 main_scheduler.py

```
import mono_sch as sch
import com as com
import time
import os
import sys
from pins import pins
import digitalio
import json
import custom_functions as cf
import interrupt_functions as intrf

class scheduler():
    def __init__(self, buffer_size = None, buffer=None, increment=1):
        if buffer:
            self.buffer = buffer
            self.buffer_size = len(buffer)
        elif buffer_size:
            self.buffer = bytearray(buffer_size)
            self.buffer_size = buffer_size
        else:
            self.buffer = bytearray(0)
            self.buffer_size = 0
        self.increment = increment
        self.task_pattern = None
        self.sch_period = None

    def run_SPI(self, task, data):
        res = None
        if task["connection"]["receive"] or task["connection"]["send"]:
            res = com.Com_SPI( data_size = task["connection"]["data_size"], data = data, delimiter = task["connection"]["
      data_delimiter"], start = task["connection_settings"]["start"], start_value = task["connection_settings"]["start_value"], end =
      task["connection_settings"]["end"], end_value = task["connection_settings"]["end_value"], front_data_padding = task["
      connection_settings"]["front_data_padding"],back_data_padding = task["connection_settings"]["back_data_padding"],baud = self.
      devices[task["connection"]["device_name"]]["SPI"]["baud"],bits = self.devices[task["connection"]["device_name"]]["SPI"]["bits"],
      polarity = self.devices[task["connection"]["device_name"]]["SPI"]["polarity"],phase = self.devices[task["connection"]["
      device_name"]]["SPI"]["phase"],write_value = self.devices[task["connection"]["device_name"]]["SPI"]["write_value"],CLK = pins[
      self.devices[task["connection"]["device_name"]]["SPI"]["CLK"]],MOSI = pins[self.devices[task["connection"]["device_name"]]["SPI"
      ]["MOSI"]],MISO = pins[self.devices[task["connection"]["device_name"]]["SPI"]["MISO"]],)

            if res != None:
                if task["connection"]["pass_through"]:
                    com.Com_SPI(data = res, delimiter = task["connection"]["data_delimiter"], start = task["connection_settings"]["
      start"], start_value = task["connection_settings"]["start_value"], end = task["connection_settings"]["end"], end_value = task["
      connection_settings"]["end_value"], front_data_padding = task["connection_settings"]["front_data_padding"], back_data_padding =
      task["connection_settings"]["back_data_padding"], baud = self.devices[task["connection"]["device_name"]]["SPI"]["baud"], bits =
      self.devices[task["connection"]["device_name"]]["SPI"]["bits"], polarity = self.devices[task["connection"]["device_name"]]["SPI"
      ]["polarity"], phase = self.devices[task["connection"]["device_name"]]["SPI"]["phase"], write_value = self.devices[task["
      connection"]["device_name"]]["SPI"]["write_value"], CLK = pins[self.devices[task["connection"]["device_name"]]["SPI"]["CLK"]],
      MOSI = pins[self.devices[task["connection"]["device_name"]]["SPI"]["MOSI"]], MISO = pins[self.devices[task["connection"]["
      device_name"]]["SPI"]["MISO"]],)
            return res

    def run_UART(self, task, data):
        res = None
        if task["connection"]["receive"] or task["connection"]["send"]:
            res = com.Com_UART( data_size = task["connection"]["data_size"], data = data, delimiter = task["connection"]["
      data_delimiter"], start = task["connection_settings"]["start"], start_value = task["connection_settings"]["start_value"], end =
      task["connection_settings"]["end"], end_value = task["connection_settings"]["end_value"], front_data_padding = task["
      connection_settings"]["front_data_padding"], back_data_padding = task["connection_settings"]["back_data_padding"], baud = self.
      devices[task["connection"]["device_name"]]["UART"]["baud"], bits = self.devices[task["connection"]["device_name"]]["UART"]["bits
      "], parity = self.devices[task["connection"]["device_name"]]["UART"]["parity"], stop = self.devices[task["connection"]["
      device_name"]]["UART"]["stop"], timeout = self.devices[task["connection"]["device_name"]]["UART"]["timeout"],
      receiver_buffer_size = self.devices[task["connection"]["device_name"]]["UART"]["receiver_buffer_size"], TX = pins[self.devices[
      task["connection"]["device_name"]]["UART"]["TX"]], RX = pins[self.devices[task["connection"]["device_name"]]["UART"]["RX"]])

            if res != None:
                if task["connection"]["pass_through"]:
                    res = com.Com_UART( data = res, delimiter = task["connection"]["data_delimiter"], front_data_padding = task["
      connection_settings"]["front_data_padding"], back_data_padding = task["connection_settings"]["back_data_padding"], start = task[
      "connection_settings"]["start"], start_value = task["connection_settings"]["start_value"], end = task["connection_settings"]["
```

```
end"], end_value = task["connection_settings"]["end_value"], baud = self.devices[task["connection"]["pass_location"]]["UART"]["
baud"], bits = self.devices[task["connection"]["pass_location"]]["UART"]["bits"], parity = self.devices[task["connection"]["
pass_location"]]["UART"]["parity"], stop = self.devices[task["connection"]["pass_location"]]["UART"]["stop"], timeout = self.
devices[task["connection"]["pass_location"]]["UART"]["timeout"], receiver_buffer_size = self.devices[task["connection"]["
pass_location"]]["UART"]["receiver_buffer_size"], TX = pins[self.devices[task["connection"]["pass_location"]]["UART"]["TX"]], RX
 = pins[self.devices[task["connection"]["pass_location"]]["UART"]["RX"]])
        return res

    def run_I2C(self, task, data):
        res = None

        if task["connection"]["receive"] or task["connection"]["send"]:
            res = com.Com_I2C( self.devices[task["connection"]["device_name"]]["I2C"]["address"], data_size = task["connection"]["
data_size"], data = data, delimiter = task["connection"]["data_delimiter"], start = task["connection_settings"]["start"],
start_value = task["connection_settings"]["start_value"], end = task["connection_settings"]["end"], end_value = task["
connection_settings"]["end_value"], front_data_padding = task["connection_settings"]["front_data_padding"], back_data_padding =
task["connection_settings"]["back_data_padding"], SDA= pins[self.devices[task["connection"]["device_name"]]["I2C"]["SDA"]], SCL=
 pins[self.devices[task["connection"]["device_name"]]["I2C"]["SCL"]], frequency = self.devices[task["connection"]["device_name"
]]["I2C"]["frequency"])

            if res != None:
                if task["connection"]["pass_through"]:
                com.Com_I2C(devices[task["connection"]["pass_location"]]["I2C"]["address"], data = res, delimiter = task["connection"
]["data_delimiter"], front_data_padding = task["connection_settings"]["front_data_padding"], back_data_padding = task["
connection_settings"]["back_data_padding"], start=task["connection_settings"]["start"], start_value = task["connection_settings"
]["start_value"], end=task["connection_settings"]["end"], end_value = task["connection_settings"]["end_value"], SDA= pins[
devices[task["connection"]["pass_location"]]["I2C"]["SDA"]], SCL= pins[devices[task["connection"]["pass_location"]]["I2C"]["SCL"
]], frequency = devices[task["connection"]["pass_location"]]["I2C"]["baud"])
        return res

    def handle_interrupt(self, task):
        intr_pins = []
        for pin in task["pins"]:
             p = digitalio.DigitalInOut(pins[pin])
             p.switch_to_input()
             intr_pins.append(p)
        intrf.func_dict[task["interrupt_function"]](self.buffer,self.devices, intr_pins,task["arguments"])

        for pin in intr_pins:
            pin.deinit()

    def setup(self,base_folder = "", tasks_folder = "tasks/", devices_folder ="devices/", interrupts_folder  = "interrupts/", schedule
 = None, devices = "devices.txt", default_task="default_task.txt", default_device="default_devices.txt", default_interrupt = "
default_interrupt.txt", step=1):

        interrupt_files = os.listdir(base_folder+interrupts_folder)
        task_files = os.listdir(base_folder+tasks_folder)

        if default_task in task_files:
                    task_files.remove(default_task)

        if default_interrupt in interrupt_files:
            interrupt_files.remove(default_interrupt)

        self.tasks = {}
        self.devices = {}

        with open(base_folder + devices_folder + devices) as f:
            self.devices = json.loads(f.read())

        with open(base_folder + devices_folder+default_device) as f:
            devices_default = json.loads(f.read())

            for device in self.devices:
                self.devices[device] = com.add_default_dict(self.devices[device], devices_default)

        with open(base_folder + interrupts_folder + default_interrupt) as f:
            interrupt_default = json.loads(f.read())
            for file in interrupt_files:
                with open(base_folder+interrupts_folder+file) as f:
                    interrupts = json.loads(f.read())
                    for interrupt in interrupts:
                        self.tasks[interrupt] = com.add_default_dict(interrupts[interrupt], interrupt_default)
                        self.tasks[interrupt]["task_type"] = "interrupt"

        with open(base_folder + tasks_folder+default_task) as f:
            tasks_default = json.loads(f.read())

            for file in task_files:
                with open(base_folder + tasks_folder+ file) as f:
                    tasks = json.loads(f.read())
                    for task in tasks:
                        if "duration" in tasks[task]:
```

```
                        if tasks[task]["duration"] == None:
                            tasks[task]["duration"] = com.calculate_duration(tasks[task], self.devices)

                    self.tasks[task] = com.add_default_dict(tasks[task], tasks_default)
                    self.tasks[task]["task_type"] = "task"

        if schedule == None:
            if not sch.possible_schedule(self.tasks):
                return False

            self.task_pattern, self.sch_period = sch.schedule_order(self.tasks, step=step)
        else:
            self.task_pattern, self.sch_period = schedule[0], schedule[1]
        return True

    def run_task(self,task_name):
        task = self.tasks[task_name]
        self.task_run(task)

    def task_run(self, task):
        active_pins = []

        if task["task_type"] == "interrupt":
            self.handle_interrupt(task)
        else:
            #Set pins high and low
            if "pins" in task:
                for pin in task["pins"]["pin_high"]:
                    p = digitalio.DigitalInOut(pins[pin])
                    p.direction = digitalio.Direction.OUTPUT
                    p.value = True
                    active_pins.append(p)

                for pin in task["pins"]["pin_low"]:
                    p = digitalio.DigitalInOut(pins[pin])
                    p.direction = digitalio.Direction.OUTPUT
                    p.value = False
                    active_pins.append(p)

            data = task["connection"]["data"]
            #Load data
            if task["load_settings"]["load_from_buf"]:
                data = self.buffer[task["load_settings"]["load_address_start"]:task["load_settings"]["load_address_end"]]

            if task["load_settings"]["load_from_file"]:
                i = 0
                lines = []
                with open(task["load_settings"]["load_file"]) as f:
                    if not task["load_settings"]["disjoint"]:
                        i = task["load_settings"]["start_line"]
                        while(i < task["load_settings"]["end_line"]):
                            lines.append(f.readline())
                            i +=1
                    else:
                        current_l = 0
                        l = task["load_settings"]["lines"][current_l]
                        while (i <= task["load_settings"]["lines"][-1]):
                            if l == i:
                                lines.append(f.readline().strip())
                                current_l += 1
                                if current_l >= len(task["load_settings"]["lines"]):
                                    break
                                l = task["load_settings"]["lines"][current_l]
                            else:
                                f.readline()
                            i += 1
                data = task["load_settings"]["line_delimiter"].join(lines)

            if task["custom_function"]["custom"] and task["custom_function"]["before"]:
                d = cf.func_dict[task["custom_function"]["custom_function_name"]](self.buffer,active_pins,data,task["custom_function"
    ]["arguments"])
                if d != None:
                    data = d

            #Run task
            if task["connection"]["type"] == "I2C":
                data = self.run_I2C(task,data)
            elif task["connection"]["type"] == "UART":
                data = self.run_UART(task, data)
            elif task["connection"]["type"] == "SPI":
                data = self.run_SPI(task, data)
            elif task["connection"]["type"] == "CAN":
                print("CAN not implemented")
```

```
            else:
                print("Unknown connection type {}".format(task["connection"]["type"]))
            if task["custom_function"]["custom"] and task["custom_function"]["after"]:
                data = cf.func_dict[task["custom_function"]["custom_function_name"]](self.buffer,active_pins,data,task["
    custom_function"]["arguments"] )

            #Store data
            if task["store_settings"]["store_to_buf"]:
                addr = int(task["store_settings"]["store_address"])
                if data != None:
                    for i in range(len(data)):
                        self.buffer[addr] = data[i]
                        addr += 1
                        if addr >= self.buffer_size:
                            print("Storage overflow")
                            break

            elif task["store_settings"]["store_to_file"]:
                if task["store_settings"]["file_append"]:
                    with open(task["store_settings"]["store_file"], "a") as f:
                        f.write(data)
                        f.write("\n")
                else:
                    with open(task["store_settings"]["store_file"], "w") as f:
                        f.write(data)

            for pin in active_pins:
                pin.deinit()

    def run_schedule(self, catch_err = True):
        sch_t = -1
        i =0
        if self.task_pattern[-1] != ("end", float("inf")):
            self.task_pattern.append(("end", float("inf")))

        n = len(self.task_pattern)
        print(self.task_pattern)
        t = time.monotonic_ns()-self.increment

        while True:
                if t + self.increment <= time.monotonic_ns():
                    sch_t = (sch_t+((time.monotonic_ns()-t)//self.increment))

                    if sch_t >= self.sch_period:
                        print("Reset at {}".format(sch_t))
                        sch_t = (sch_t%self.sch_period) - 1
                        i = 0

                    t = time.monotonic_ns()
                    if self.task_pattern[i][1] <= sch_t:
                        print("Running Task {} at {}".format(self.task_pattern[i][0], sch_t))
                        try:
                            self.run_task(self.task_pattern[i][0])
                        except Exception as e:
                            if catch_err == True:
                                print("Task failed {}".format(self.task_pattern[i][0]))
                                sys.print_exception(e)
                            else:
                                raise e

                        i = ((i+1)%n)

    def run_startup(self, schedule_file = "schedule.txt", base_folder = "", startup_folder = "startup/", tasks = "tasks/"):
        self.setup(base_folder = base_folder,  tasks_folder = startup_folder + tasks, schedule = ([],[]))
        with open(base_folder+ startup_folder+ schedule_file) as f:
            lines=f.readlines()
            for line in lines:
                line = line.strip()
                self.run_task(line)
        self.devices = None
        self.task_pattern = None
        self.sch_period = None

    def calculate_run(self, task, margin = 0.2):
        t1 = time.monotonic_ns()
        try:
            temp = self.buffer.copy()
            self.task_run(task)
            self.buffer = temp.copy()
        except:
            print("Error running task, estimating duration instead")
            return com.calculate_duration(task, self.devices)
        t2 = time.monotonic_ns()
```

```
        duration = t1-t2
        duration = (1+margin) * duration
        return duration

def testing():
    global com
    import testing.mock_com as com
    global cf
    import testing.mock_custom_functions as cf
    global intrf
    import testing.mock_interrupt_functions as intrf

def undo_testing():
    global com
    import com as com
    global cf
    import custom_functions as cf
    global intrf
    import interrupt_functions as intrf
```

A.4 main.py

```
import main_scheduler as ms
import testing.run_tests as t

BUFFER_SIZE = 124
PROJECT_FOLDER = "Demo/"

print("Running Tests")
t.run(verbose=True)
print("Creating Scheduler")
sch = ms.scheduler(BUFFER_SIZE, increment = 1000)
print("Runing Startup")
sch.run_startup(base_folder = PROJECT_FOLDER)
print("Runing Setup")
sch.setup(base_folder = PROJECT_FOLDER, step=1000000)
print("Runing Schedule")
sch.run_schedule()
```

B Test Cases

B.1 mono_sch.py Test Cases

```
# Write your code here :-)
import mono_sch as ms

def test_empty():
    tasks = {}
    assert ms.possible_schedule(tasks) == False

def test_possible_single():
    tasks = {"task":{"period":5, "duration":2}}
    assert ms.possible_schedule(tasks) == True

def test_possible_multi():
    tasks = {"task_1":{"period":5, "duration":2}, "task_2":{"period":6, "duration":1}, "task_3":{"period":7, "duration":1}}
    assert ms.possible_schedule(tasks) == True

def test_impossible_single():
    tasks = {"task":{"period":5, "duration":5}}
    assert ms.possible_schedule(tasks) == False

def test_impossible_multi():
    tasks = {"task_1":{"period":5, "duration":3}, "task_2":{"period":6, "duration":4}, "task_3":{"period":5, "duration":1}}
    assert ms.possible_schedule(tasks) == False

def test_malformed_single():
    tasks = {"task_1":{"period":5, "deadline":10}}
    err = False
    try:
        ms.possible_schedule(tasks)
    except KeyError as e:
```

```
        err = True
    assert err == True

def test_malformed_multi():
    tasks = {"task_1":{"period":5, "duration":1},"task_2":{"period":5, "deadline":1},"task_3":{"period":5, "duration":1} }
    err = False
    try:
        ms.possible_schedule(tasks)
    except KeyError as e:
        err = True
    assert err == True

def test_list():
    tasks = ["list", "items"]
    assert ms.possible_schedule(tasks) == False

def test_string():
    tasks = "String"
    assert ms.possible_schedule(tasks) == False

def test_schedule_single():
    tasks = {"task":{"period":5, "duration":2}}
    assert ms.schedule_order(tasks) == ([("task", 0)], 5)

def test_schedule_multi():
    tasks = {"task_1":{"period":3, "duration":1}, "task_2":{"period":4, "duration":1}, "task_3":{"period":6, "duration":1}}
    assert ms.schedule_order(tasks) == ([('task_1', 0), ('task_2', 1), ('task_3', 2), ('task_1', 3), ('task_2', 4), ('task_3', 6), ('
      task_1', 7), ('task_2', 8), ('task_1', 9)], 12)
```

B.2 main_scheduler.py Test Cases

```
import os
import main_scheduler as ms
import sys

def default_dict_linear():
    data = {"test":"value", "test2":"random", "test3":"entry"}
    default = {"test2": "value", "test4":"default"}
    data =ms.com.add_default_dict(data, default)
    assert data ==  {"test":"value", "test2":"random", "test3":"entry", "test4":"default"}

def default_dict_both_nested():
    data = {"test":"value", "test2":{"test4":"ran", "test5":"dom"}, "test3":"entry"}
    default = {"test2": {"test4":"foo", "test6":"bar"}, "test4":"default"}
    data =ms.com.add_default_dict(data, default)
    assert data == {"test":"value", "test2":{"test4":"ran", "test5":"dom", "test6":"bar"}, "test3":"entry", "test4":"default"}

def default_dict_data_nested():
    data = {"test":"value", "test2":{"test4":"ran", "test5":"dom"}, "test3":"entry"}
    default = {"test2": "foobar", "test4":"val"}
    data =ms.com.add_default_dict(data, default)
    assert data == {"test":"value", "test2":{"test4":"ran", "test5":"dom"}, "test3":"entry", "test4":"val"}

def default_dict_default_nested():
    data = {"test":"value", "test2":"random", "test3":"entry"}
    default = {"test2": {"test4":"foo", "test6":"bar"}, "test4":{"default":"val"}}
    data = ms.com.add_default_dict(data, default)
    assert data == {"test":"value", "test2":"random", "test3":"entry", "test4":{"default":"val"}}

def default_dict_empty_data():
    default = {"test2": "value", "test4":"default"}
    data =ms.com.add_default_dict({}, default)
    assert data ==  {"test2": "value", "test4":"default"}

def default_dict_empty_default():
    data = {"test":"value", "test2":"random", "test3":"entry"}
    data =ms.com.add_default_dict(data, {})
    assert data ==  {"test":"value", "test2":"random", "test3":"entry"}

def create():
    sch = ms.scheduler(64)
    return sch

def setup(Case, sch = None):
    if sch == None:
        sch = create()
    sch.setup(base_folder = ("testing/test_Cases/" + Case + "/"),
              devices = "devices.txt",
              default_task="default_task.txt",
```

```
            default_device="default_devices.txt",
            default_interrupt = "default_interrupt.txt",
            step=1)
    return sch

def test_setup_no_tasks():
    sch = setup("case_0")
    assert sch.task_pattern == None
    assert sch.sch_period == None

def test_setup_possible_tasks():
    sch = setup("case_1")
    assert sch.task_pattern == [('task_1', 0), ('task_2', 1), ('task_3', 2), ('task_1', 3), ('task_2', 4), ('task_3', 6), ('task_1', 7), ('task_2', 8), ('task_1', 9)]
    assert sch.sch_period == 12

def test_setup_impossible_tasks():
    sch = setup("case_2")
    assert sch.task_pattern == None
    assert sch.sch_period == None

def test_setup_interrupt_tasks():
    sch = setup("case_3")
    assert sch.task_pattern == [('task_1', 0), ('intr_1', 1), ('task_2', 2), ('task_1', 3), ('intr_1', 4), ('task_2', 6), ('task_1', 7), ('intr_1', 8), ('task_1', 9)]
    assert sch.sch_period == 12

def test_run_SPI_read_store_buffer():
    sch = setup("case_4")
    sch.run_task("read_SPI")
    assert sch.buffer[0:8] == bytearray("a"*8)

def test_run_UART_read_store_buffer():
    sch = setup("case_4")
    sch.run_task("read_UART")
    assert sch.buffer[8:16] == bytearray("b"*8)

def test_run_I2C_read_store_buffer():
    sch = setup("case_4")
    sch.run_task("read_I2C")
    assert sch.buffer[16:24] == bytearray("c"*8)

def test_run_SPI_read_store_file():
    sch = setup("case_5")
    sch.run_task("read_SPI")

    with open("testing/test_cases/case_5/files/test_data_spi.txt") as f:
       line = f.readlines()[0].strip()
    assert "a"*8 == line

def test_run_UART_read_store_file():
    sch = setup("case_5")
    sch.run_task("read_UART")

    with open("testing/test_cases/case_5/files/test_data_uart.txt") as f:
       line = f.readlines()[0].strip()
    assert "b"*8 == line

def test_run_I2C_read_store_file():
    sch = setup("case_5")
    sch.run_task("read_I2C")

    with open("testing/test_cases/case_5/files/test_data_i2c.txt") as f:
       line = f.readlines()[0].strip()
    assert "c"*8 == line

def test_custom_function():
    sch = setup("case_6")
    sch.buffer[0:15] = bytearray("1\n12\n3\n14\n5\n16\n")
    sch.run_task("sum_data")
    assert sch.buffer[16:18] == bytearray("58") #1+12+3+14+5+16+7

def test_interrupt():
    sch = setup("case_6")
    sch.buffer[0:15] = bytearray("1\n12\n3\n14\n5\n16\n")
    sch.run_task("intr_product")
    assert sch.buffer[16:22] == bytearray("282240") #1*12*3*14*5*16*7

def test_run_startup():
    sch = create()
    sch.run_startup(schedule_file = "schedule.txt",base_folder = "testing/test_cases/case_7/")
    assert sch.buffer[0:8] == bytearray("a"*8)
    assert sch.buffer[8:16] == bytearray("b"*8)
```

```
    assert sch.buffer[16:24] == bytearray("c"*8)

def test_run_startup_with_follow_up_setup():
    sch = create()
    sch.run_startup(schedule_file = "schedule.txt", base_folder = "testing/test_cases/case_8/")
    sch = setup("case_8", sch=sch)
    sch.run_task("read_I2C")
    assert sch.buffer[0:8] == bytearray("a"*8)
    assert sch.buffer[8:16] == bytearray("b"*8)
    assert sch.buffer[16:24] == bytearray("c"*8)
```

C Demo Program

C.1 Demo Code

C.1.1 Demo Payload

```
import machine
import time
import random
import demo_adc

class Payload():
    def __init__(self, sw=0, sw1=2, sw2=4, intr_out=32, intr_in = 33,  therm=13, unit=2, led=12, led1=14, led2=27, sda=21,scl=22, addr
     =32):
        self.adc = demo_adc.temperature(pin=therm, unit=unit)
        self.sw = machine.Pin(sw, handler=self.switch, trigger=machine.Pin.IRQ_ANYEDGE)
        self.sw1 = machine.Pin(sw1, handler=self.switch1, trigger=machine.Pin.IRQ_ANYEDGE)
        self.sw2 = machine.Pin(sw2, handler=self.switch2, trigger=machine.Pin.IRQ_ANYEDGE)
        self.intr = machine.Pin(intr_out,   mode = machine.Pin.OUT, pull=machine.Pin.PULL_DOWN, value = 0)
        self.led = machine.Pin(led,   mode = machine.Pin.OUT, pull=machine.Pin.PULL_DOWN, value = 0)
        self.led1 = machine.Pin(led1, mode = machine.Pin.OUT, pull=machine.Pin.PULL_DOWN, value = 0)
        self.led2 = machine.Pin(led2, mode = machine.Pin.OUT, pull=machine.Pin.PULL_DOWN, value = 0)
        self.interupt = machine.Pin(intr_in, handler=self.tidy_interupt, pull = machine.Pin.PULL_DOWN, trigger=machine.Pin.IRQ_RISING)

        self.i2c = machine.I2C(1, mode=machine.I2C.SLAVE, sda=sda, scl=scl, slave_addr = addr)
        self.i2c.callback(self.i2c_cb, self.i2c.CBTYPE_ADDR | self.i2c.CBTYPE_TXDATA)
        self.led_count = [0,0,0]

    def led_increment(self):
        if self.led_count[2] == 1:
            self.led_count[2] = 0
            if self.led_count[1] == 1:
                self.led_count[1] = 0
                if self.led_count[0] == 1:
                    self.led_count[0] = 0
                else:
                    self.led_count[0] = 1
            else:
                self.led_count[1] = 1
        else:
            self.led_count[2] = 1

        self.led.value(self.led_count[2])
        self.led1.value(self.led_count[1])
        self.led2.value(self.led_count[0])

    def i2c_cb(self, res):
        cbtype = res[0] # i2c slave cllback type
        if cbtype == machine.I2C.CBTYPE_TXDATA:
            print("[I2C] Data sent to master: addr={}, len={}, ovf={}, data={}".format(res[1], res[2], res[3], res[4]))
        elif cbtype == machine.I2C.CBTYPE_ADDR:
            print("[I2C] Addres set: addr={}".format(res[1]))
            if res[1] == 0:
                temp = bytearray(str(int(self.adc.get_temp_C()))+ "    ")
                self.i2c.setdata(temp, 0)
                print("set temp {}".format(temp))
        else:
            print("Unknown CB type, received: {}".format(res))

    def tidy_interupt(self, pin):
        print("Tidying")
        self.led_increment()
        self.intr.value(0)
        self.i2c.setdata(bytearray(1),100)
```

```
        self.i2c.setdata(bytearray(1),101)
        self.i2c.setdata(bytearray(1),102)
        print(self.i2c.getdata(100, 3))

    def switch(self,pin):
        print("Interupted SW")
        self.intr.value(1)
        b = bytearray(1)
        b[0] = 1
        self.i2c.setdata(b,100)
        print(self.i2c.getdata(100, 3))

    def switch1(self,pin):
        print("Interupted SW1")
        self.intr.value(1)
        b = bytearray(1)
        b[0] = 1
        self.i2c.setdata(b,101)
        print(self.i2c.getdata(100, 3))

    def switch2(self,pin):
        print("Interupted SW2")
        self.intr.value(1)
        b = bytearray(1)
        b[0] = 1
        self.i2c.setdata(b,102)
        print(self.i2c.getdata(100, 3))

    def deinit(self):
        self.adc.deinit()
        self.i2c.deinit()

    def __del__(self):
        self.deinit()
```

C.1.2 Demo Coms

```
import display
import machine
import time

class Coms():
    def __init__(self, sda=21, scl=22, miso=13, mosi=12, clk=14,cs=15,dc=0,rst_pin=27, backl_pin=2, backl_on=1, bgr=True, tx=17, rx
      =16, buffer_size=333, timeout=5000):
        self.i2c = machine.I2C(1,mode=machine.I2C.SLAVE, sda=21, scl=22)
        self.scr = display.TFT()
        self.scr.init(self.scr.ILI9341, miso=13, mosi=12, clk=14, cs=15,dc=0,rst_pin=27, backl_pin=2, backl_on=1, bgr=True)
        self.scr.clear(self.scr.BLACK)
        self.scr.orient(self.scr.LANDSCAPE)
        self.width = self.scr.screensize()[0]
        self.height = self.scr.screensize()[1]

        self.t = 10
        self.current = 0
        self.current_y = self.height//2
        self.maxi = None
        self.mini = None
        self.i2c.callback(self.i2c_cb, self.i2c.CBTYPE_ADDR|self.i2c.CBTYPE_RXDATA)
        self.origin = (10, self.height//2)
        self.background = self.scr.BLACK
        self.foreground = self.scr.GREEN
        self.flipped = False
        self.uart = machine.UART(1, tx=tx, rx=rx, timeout = timeout, buffer_size = buffer_size)
        self.uart.callback(self.uart.CBTYPE_DATA, self.uart_cb,data_len=3)
        self.setup()
        self.maxi = -float(inf)
        self.mini = float(inf)

    def setup(self):
        self.scr.clear(self.background)
        self.t = 10
        self.current = 0
        self.scr.line(10,10, 10, self.height//2, self.foreground)
        self.scr.line(10,self.origin[1], self.width-10, self.origin[1], self.foreground)
        text = "Temperature  mesaurement"
        t_width = self.scr.textWidth(text)
        self.scr.text((self.width//2-t_width//2),self.origin[1]+20, text)
        text = "Current value:    "
        t_width = self.scr.textWidth(text)
        self.scr.text((self.width//2-t_width//2),self.origin[1]+40, text)
```

```
        text = "Highest value:    "
        t_width = self.scr.textWidth(text)
        self.scr.text((self.width//2-t_width//2),self.origin[1]+60, text)
        self.scr.text((self.width//2+38),self.origin[1]+80, " " + str(self.maxi))
        text = "Lowest value:     "
        t_width = self.scr.textWidth(text)
        self.scr.text((self.width//2-t_width//2),self.origin[1]+80, text)
        self.scr.text((self.width//2+38),self.origin[1]+80, " " + str(self.mini))

    def reset(self, call):
        self.current_y = self.height//2
        self.maxi = None
        self.mini = None
        self.setup()
        self.maxi = float(inf)
        self.mini = -float(inf)
        print("Reset")

    def flip(self,call):
        if self.flipped:
            self.scr.orient(self.scr.LANDSCAPE)
            self.flipped = False
        else:
            self.scr.orient(self.scr.LANDSCAPE_FLIP)
            self.flipped = True
        self.setup()
        print("Flipped")

    def invert(self,call):
        temp = self.foreground
        self.foreground = self.background
        self.background = temp
        self.setup()
        print("Inverted")

    def printing(self,call):
        print("UART Recieved {}".format(call))

    def uart_cb(self,call):
        print(call[2][2])
        if call[2][0] == "1":
            self.flip(call)
        if call[2][1] == "1":
            self.reset(call)
        if call[2][2] == "1":
            self.invert(call)

    def i2c_cb(self,call):
        cbtype = call[0]

        if cbtype == machine.I2C.CBTYPE_RXDATA:
            print("Recieved [{}] to address {} of length {} and with an overflow of {}".format(call[4], call[1], call[2], call[3]))
            self.plot_update(call[4])
        elif cbtype == machine.I2C.CBTYPE_ADDR:
            print("Set new address to {}".format(call[1]))

    def plot_update(self,new):
        new_t = self.t+8
        new = int((str(new)[2:6]))
        new_y = self.origin[1] - new
        if new_t > self.width -10:
            self.setup()
            new_t = self.t + 8
            self.scr.text((width//2+41),self.origin[1]+60, " " + str(self.maxi))
            self.scr.text((width//2+38),self.origin[1]+80, " " + str(self.mini))

        self.scr.line(self.t, self.current_y, new_t, new_y, self.scr.WHITE)
        self.scr.circle(new_t, new_y, 2, self.scr.GREEN, self.scr.GREEN)
        self.scr.textClear((self.width//2+41),self.origin[1]+40, " " + str(self.current))
        self.scr.text((self.width//2+41),self.origin[1]+40, " " + str(new))

        if new > self.maxi:
            self.scr.textClear((self.width//2+41),self.origin[1]+60, " " + str(self.maxi))
            self.scr.text((self.width//2+41),self.origin[1]+60, " " + str(new))
            self.maxi = new

        if new < self.mini:
            self.scr.textClear((self.width//2+38),self.origin[1]+80, " " + str(self.mini))
            self.scr.text((self.width//2+38),self.origin[1]+80, " " + str(new))
            self.mini = new

        self.t = new_t
        self.current = new
```

```
        self.current_y = new_y

    def dinit(self):
        self.i2c.deinit()

    def __del__(self):
        self.deinit()
```

C.1.3 therm.py

```
from machine import ADC
import math

class temperature():
    def __init__(self, pin=13, unit=2, atten = ADC.ATTN_11DB, T0 = 298.15, B = 3984, R0 = 10000, Rd = 10000, Vs = 3300):
        self.adc = ADC(pin, unit=unit)
        self.adc.atten(atten)
        self.T0 = T0
        self.B  =  B
        self.R0 = R0
        self.Rd = Rd
        self.Vs = Vs

    def get_temp_K(self):
        Rt = self.Rd*((self.Vs/self.adc.read())-1)
        T_inv = 1/self.T0 + 1/self.B * math.log(Rt/self.R0)
        return 1/T_inv

    def K_to_C(self,k_temp):
        return k_temp-272.15

    def get_temp_C(self):
        return self.K_to_C(self.get_temp_K())

    def deinit(self):
        self.adc.deinit()

    def __del__(self):
        self.deinit()
```

C.2 Demo tasks

```
DEFAULT DEVICES:
{"I2C":{"address":32, "SDA":"D20", "SCL":"D21", "frequency":100000, "timeout":255},

 "UART":{"RX":"D0", "TX":"D1", "baud":9600, "bits":8, "parity":null, "stop":1, "timeout":1,
         "receiver_buffer_size":64}

  "SPI" : {"CLK":"D52", "MOSI":null, "MISO":null, "slave":null, "write_value" : 0,  "baud":100000, "polarity":0, "phase":0, "bits":8}
}
DEVICES:
{
 "Coms" : {"I2C":{"address":32}, "UART":{"baud":115942}},
 "Payload" : {"I2C":{"address":33}}
}
TASKS:
{
 "read_sensor":{
                "duration": 1000000,
                "period":   6000000,
                "connection" :{
                                "device_name":"Payload",
                                "type": "I2C",
                                "receive": true,
                                "data_size" : 6,
                },
                "store_settings":{
                                "store_to_buf": true,
                                "store_address" : 0 ,
                }
          }
}
{
 "write_screen":{
                "duration": 1000000,
                "period":   6000000,
```

```
                "connection" :{
                                "device_name":"Coms",
                                "type": "I2C",
                                "send" : true,
                                "data_size" 6
                },
                "connection_settings" : {
                                        "front_data_padding" : 1,
                },
                "load_settings" : {
                                    "load_from_buf": true,
                                    "load_address_start" : 0,
                                    "load_address_end" : 6,
                }
 }
}
INTERRUPTS:
{
 "payload_int" : {"interrupt_function": "payload_int", "period":3000000, "duration":1000000, "pins":["D19", "D18"], "arguments":[]}
}
```

C.2.1 interrupt_functions.py

```
import busio
from pins import pins
import digitalio
import time

def payload_interrupt_handler(buffer,devices,intr_pins,args):
    SCL = pins[devices["Payload"]["I2C"]["SCL"]]
    SDA = pins[devices["Payload"]["I2C"]["SDA"]]
    frequency = devices["Payload"]["I2C"]["frequency"]
    timeout = devices["Payload"]["I2C"]["timeout"]
    addr = devices["Payload"]["I2C"]["address"]
    i2c = busio.I2C(SCL, SDA, frequency=frequency, timeout=timeout)
    i = 0
    while (i < 100):
        lock = i2c.try_lock()
        if lock:
            break
        else:
            i += 1
    if not lock:
        i2c.deinit()
        return
    mem = bytearray(1)
    mem[0] = 100
    i2c.writeto(addr, mem)
    received = bytearray(3)
    received = bytearray(3)
    i2c.readfrom_into(addr, received)
    i2c.unlock()
    intr_pins[0].direction = digitalio.Direction.OUTPUT
    intr_pins[0].value = True
    TX = pins[devices["Coms"]["UART"]["TX"]]
    RX = pins[devices["Coms"]["UART"]["RX"]]
    baud=devices["Coms"]["UART"]["baud"]
    bits= devices["Coms"]["UART"]["bits"]
    parity= devices["Coms"]["UART"]["parity"]
    stop= devices["Coms"]["UART"]["stop"]
    timeout= devices["Coms"]["UART"]["timeout"]
    receiver_buffer_size= devices["Coms"]["UART"]["receiver_buffer_size"]
    uart = busio.UART(TX,RX, baudrate=baud, bits=bits, parity=parity, stop=stop, timeout=timeout, receiver_buffer_size=
      receiver_buffer_size)
    send = ["0","0","0"]
    if received[0] == 1:
        send[0] = "1"
    if received[1] == 1:
        send[1] = "1"
    if received[2] == 1:
        send[2] = "1"
    if send != ["0","0","0"]:
        print(uart.write(bytearray("".join(send)) + bytearray(2)))
    uart.deinit()
    i2c.deinit()

func_dict = {"payload_int":payload_interrupt_handler}
```