Fault Redundant On-Board Computer using COTS components

Oliver Launchbury-Clark

Master of Engineering in Electronic Engineering
from the
University of Surrey

Department of Electronic Engineering
Faculty of Engineering and Physical Sciences
University of Surrey
Guildford, Surrey, GU2 7XH, UK

January 2015
Supervised by: Dr Chris Bridges

© Oliver Launchbury-Clark 2015
DECLARATION OF ORIGINALITY

I confirm that the project dissertation I am submitting is entirely my own work and that any material used from other sources has been clearly identified and properly acknowledged and referenced. In submitting this final version of my report to the JISC anti-plagiarism software resource, I confirm that my work does not contravene the university regulations on plagiarism as described in the Student Handbook. In so doing I also acknowledge that I may be held to account for any particular instances of uncited work detected by the JISC anti-plagiarism software, or as may be found by the project examiner or project organiser. I also understand that if an allegation of plagiarism is upheld via an Academic Misconduct Hearing, then I may forfeit any credit for this module or a more severe penalty may be agreed.

Fault Redundant On-Board Computer Using COTS Components

Oliver Launchbury-Clark

Author Signature

Date: 26/05/2015

Supervisor’s name: Dr Chris Bridges
WORD COUNT

Number of Pages: 92
Number of Words: 17671
ABSTRACT

Space is a harsh environment where it is difficult to ensure that semiconductor devices will operate for an extended time. It is also traditionally highly expensive; using Commercial-Of-The-Shelf (COTS) technologies has been proven as an effective method of reducing these costs. The project's main objective was to design and produce a fault tolerant On-Board-Computer (OBC) for a CubeSat.

Two Raspberry Pi Compute modules, miniature versions of the popular Raspberry Pi boards, are used to provide powerful and flexible processing capabilities on the OBC. The use of two Raspberry Pis also presents a fault tolerance benefit through redundancy. Finally, a microprocessor is also provided to control the operation of the two Raspberry Pis, reducing power consumption where possible.

First research into OBCs already available on the market are analysed and compared, highlighting features that may be important to include. Potential microcontroller technologies are also researched to ensure the microcontroller used is fault resilient whilst remaining affordable. An overview architecture is then designed and used to produce a circuit schematic. This is then used to create a PCB design that is produced by an external manufacturer.

The manufactured PCBs are then assembled and the basic functionality tested. Software for the Raspberry Pis and the microcontroller is developed and tested. The software and hardware is tested against the capabilities that should be available on the hardware.

The OBC successfully operates and both the Raspberry Pis can be used independently, whilst the microcontroller can control when they run. However, the microcontroller cannot directly control what processing is done by the Raspberry Pis, which would have been desirable. This also means that some of the potential redundancy schemes to be developed could not be implemented.

Overall, the project is a success, proving the concept of an On-Board-Computer with two Raspberry Pis. Considerations towards the next revision of the On-Board-Computer are also discussed.
# TABLE OF CONTENTS

Declaration of originality ........................................................................................................ ii
Word Count ............................................................................................................................... iii
Abstract ................................................................................................................................... iv
Table of Contents ...................................................................................................................... v
List of Figures ........................................................................................................................... vii
List of tables ............................................................................................................................. viii

1 Introduction .......................................................................................................................... 1
  1.1 Background and Context ................................................................................................. 1
  1.2 Scope and Objectives ....................................................................................................... 1
  1.3 Achievements ................................................................................................................... 3
  1.4 Overview of Dissertation ............................................................................................... 3

2 State-of-The-Art Review ...................................................................................................... 4
  2.1 Introduction ..................................................................................................................... 4
  2.2 Prior CubeSat On-Board-Computer systems ................................................................ 4
  2.3 Available Microcontroller technologies ...................................................................... 5
  2.4 Prior work on the use of two Raspberry Pi’s as a fault redundant system ................. 6
  2.5 Applicable Standards ..................................................................................................... 7
  2.6 Summary ......................................................................................................................... 8

3 RPi-CubeSat Development ................................................................................................. 9
  3.1 Physical constraints ......................................................................................................... 9
  3.2 Microcontroller selection ............................................................................................... 10
  3.3 Requirements for OBC’s interface busses .................................................................... 10
  3.4 Overview of the OBC system ......................................................................................... 10
  3.5 Pinouts of large components and main CubeSat kit bus header ................................ 12
  3.6 Schematic creation ......................................................................................................... 12
    3.6.1 Latching power switch with overcurrent detection ................................................. 13
    3.6.2 Overcurrent detecting automatic reset .................................................................... 15
  3.7 PCB Design ..................................................................................................................... 16
  3.8 Summary of design ......................................................................................................... 18

4 Build and Test ....................................................................................................................... 19
  4.1 Final board ...................................................................................................................... 19
  4.2 PCB assembly ................................................................................................................ 20
  4.3 Programming MSP430 (Embedded) .......................................................................... 22
  4.4 Debug LEDS .................................................................................................................. 23
  4.5 Programming Raspberry Pi’s (Linux) ............................................................................. 23
4.6 Verification of design, build, and operation ......................................................... 24
4.7 Problems encountered during build and development process ............................... 26
  4.7.1 Incorrect component footprints for SODIMM headers ..................................... 26
  4.7.2 Raspberry Pi’s failed to boot due to unused USB On-The-Go functionality ............ 27
  4.7.3 MSP-debug would not recognise the MSP430 processor .................................. 28
  4.7.4 GDB, part of the MSP430 toolchain, would not work ....................................... 28
  4.7.5 Damage to MSP430 debug and programming header ....................................... 29
  4.7.6 Raspberry Pi’s only have drivers for one UART ............................................... 29
  4.7.7 Raspberry Pi power switches problem .............................................................. 30
4.8 Summary ................................................................................................................. 31
5 Future work .............................................................................................................. 32
  5.1 Development work still required on this design iteration ...................................... 32
  5.2 Functionality to be added to the next design iteration ........................................... 32
  5.3 Alternative architectures ....................................................................................... 33
6 Conclusion .................................................................................................................. 35
7 References .................................................................................................................. 39
8 Appendices ................................................................................................................. 42
  Appendix 1. Deliverables, with expected and actual delivery dates ............................... 42
  Appendix 2. Pinout definitions ..................................................................................... 43
  Appendix 3. Complete circuit schematic ...................................................................... 46
  Appendix 4. PCB Layout designs ............................................................................... 53
  Appendix 5. First PCB design with incorrect SODIMM footprint ............................... 56
  Appendix 6. Setting up the MSP430 toolchain ......................................................... 57
  Appendix 7. C code for the MSP430 .......................................................................... 59
  Appendix 8. Custom Linux device tree file ................................................................. 82
  Appendix 9. C program to toggle GPIO pin output on Raspberry Pi ....................... 83
  Appendix 10. Daemon file to start a program upon boot .............................................. 84
LIST OF FIGURES

Figure 1 - Mechanical outline and connector position of PC/104 board [24] ........................................ 8
Figure 2 - Mechanical outline and connector position of GOMspace nanomind OBC [6] ...................... 8
Figure 3 - Mechanical outline and connector position of PCI/104 board [21] .............................. 8
Figure 4 - Physical layout of two Raspberry Pi compute modules on a CubeSat Kit sized board.
Derived from [22]. .......................................................................................................................... 9
Figure 5 - System diagram of OBC power, control and communication busses ................................ 11
Figure 6 – New latching switch design with overcurrent detection capabilities ............................... 14
Figure 7 - Automatic reset in the event of overcurrent ................................................................. 16
Figure 8 - Using Madparts to create a LQFP100 footprint ............................................................ 17
Figure 9 - Annotated top view of PCB with corrected SODIMM spacing ................................. 19
Figure 10 - Annotated bottom view of PCB with corrected SODIMM spacing ..................... 20
Figure 11 - Top side of the assembled PCB with two Raspberry Pis fitted ................................. 21
Figure 12 - Bottom side of the assembled PCB ............................................................................. 22
Figure 13 - Cross section view of incorrect pad spacing on the first design revision ............. 27
Figure 14 - Cross section view of corrected pad spacing on the second design revision .......... 27
Figure 15 - Repeating error message stopping the Raspberry Pi’s from booting ...................... 28
Figure 16 - Scope trace of voltage supply to OBC when a Raspberry Pi switch is turned on .... 30
Figure 17 - A potential alternative architecture to the current design .................................... 34
LIST OF TABLES

Table 1 - Comparison of On-Board-Computers currently available [6] [7] [8] ........................................ 4
Table 2 - Designed capabilities test methods and results ........................................................................ 25
Table 3 - Comparison of original objectives against project outcome .................................................. 35
Table 4 - Comparison of original desired capabilities against delivered capabilities ......................... 36
1 INTRODUCTION

1.1 Background and Context

The University of Surrey has a proud history of excellence in space research and development. The Surrey Space Centre (SSC) is the university’s own research department for space activities. Surrey Space Technology Limited (SSTL) was also started at the University and is still located nearby on the university owned Surrey Research Park. SSTL’s first satellite UoSAT-1 was launched in 1981, this mission’s success paved the way and SSTL has launched a total of 43 satellites to date [1].

In February 2013 [2] a joint SSC and SSTL mission called STRaND-1 was launched [3]. The STRaND-1 satellite is a 3U CubeSat [2], a form factor for nanosatellites that has become very popular. As SSC “has developed expertise in embedded and miniaturised CubeSat technologies and standards” [3] it is unlikely to be their last mission based around this form factor.

SSTL has a heritage of using commercial-off-the-shelf (COTS) parts in their projects [2]. The STRaND-1 satellite carried a Nexus One smartphone as a payload, with the potential that some of the satellites operational processing may be transferred to the phone when it has been proved as operational [2]. Another COTS device that will soon be used on the International Space Station (ISS) in an outreach and education role is a Raspberry Pi [4]. These are small and inexpensive processor boards that are capable of running Linux. They have become extremely popular for a wide variety of applications and hence benefit from a wide community support. It was proposed by the project supervisor Dr. Chris Bridges that these could be an excellent processor for a satellite as they are also cheap, readily available and come in a miniaturised form factor called a ‘Compute module.’ It was further suggested by a SSTL engineer that the Raspberry Pi’s Graphics Processing Unit (GPU) could be used for calculation of telemetry data, which could offer performance gains.

A challenge faced by anyone who designs electronics for use in space is ensuring that they remain operational despite the harsh environments of space where they will be exposed to a vacuum, extreme temperatures, charged particles, and radiation [5]. In this environment it is extremely difficult to ensure silicon based microprocessors will remain operational, one potential solution to this problem is redundancy within the processing system [5]. As such, it was also suggested by Dr. Chris Bridge that if a Raspberry Pi is to be used as a CubeSat processor, then another should be added to allow for fault tolerance in the satellites processing capabilities.

1.2 Scope and Objectives

The project is the study, design, build, and test a new fault-tolerant nanosatellite On-Board-Computer (OBC) made from two or more Raspberry Pi Compute modules and a small microcontroller for supervisory tasks. This was to be based upon the CubeSat specification and PC/104 format. The aim
of the project was to provide robust, reliable, and flexible computing for nanosatellite applications.

The objectives of the project were to:

- Prove that two Raspberry Pi compute modules can fit within the space available on a Cubesat.
- Produce an architecture design for the OBC using two Raspberry Pis and a smaller microcontroller
- Design the circuit schematic for the proof of concept OBC based upon the architecture design
- Design a Printed Circuit Board (PCB) for the proof of concept OBC using the circuit schematic
- Produce a proof of concept CubeSat OBC using the PCB design
- Produce software required in order to demonstrate the OBCs fault tolerant capabilities
- Test and document the capabilities of the OBC
- Produce documentation on both the physical OBC and the demonstration software, including the toolchains required for the software

These project goals were met by creating the deliverables defined in Appendix 1, the expected date of delivery and the actual date of delivery is shown.

The suggested operational capabilities required of the OBC were graduated into three levels of priority: Primary, Secondary, and Tertiary.

**Requirement OBC capability**

<table>
<thead>
<tr>
<th>Level</th>
<th>OBC capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>- Have two Raspberry-Pi compute modules available for use on the OBC. These should operate independently of each other</td>
</tr>
<tr>
<td></td>
<td>- Have a microcontroller to control the Raspberry Pi’s operation and monitor the Raspberry Pis for failure</td>
</tr>
<tr>
<td></td>
<td>- Should a Raspberry Pi fail, the other should be able to take over its tasks</td>
</tr>
<tr>
<td>Secondary</td>
<td>- Allow the Raspberry-Pis’ clock rates to be changed, according to the processing capabilities required and power available</td>
</tr>
<tr>
<td></td>
<td>- Allow the Raspberry-Pi’s processing state to be saved to their internal flash memory then suspended</td>
</tr>
<tr>
<td></td>
<td>- Allow the Raspberry-Pis to access communication links to the rest of the satellite and resources through arbitration</td>
</tr>
</tbody>
</table>
• Have a Real Time Clock (RTC)

Tertiary
• Breakout unused, but potentially useful Raspberry-Pi functionality such as the camera interface bus. To allow for extended usage
• Using the Raspberry-Pi’s communication link, allow for the two modules to operate in a fault redundant mode called lockstep

1.3 Achievements
A proof of concept OBC PCB was produced, populated, and tested. Software to test the OBC’s capability was also produced and used to test the board.

Throughout this process, the hardware and software, including the toolchain required, have been documented.

1.4 Overview of Dissertation
This dissertation report details the research performed in order to produce the OBC, considerations of the board’s physical design, design of the circuit schematic, and the complete PCB. Some elements of the circuits design are explained, and a complete list of capabilities the hardware should have is presented. Details regarding the physical assembly of the board, along with the development of the software required is then presented. This is then followed by the results of the tests performed against the hardware capabilities. Finally, the report is concluded with remarks about potential future development work for the project and an evaluation of the project as a whole.
2 STATE-OF-THE-ART REVIEW

2.1 Introduction

Before undertaking the project, several areas needed researching to allow a complete and successful design to be produced that will be of the most benefit to the end user.

In order to produce a working OBC that is beneficial to the end user it was important to understand any current implementations of similar systems and their use. Research was required in some areas to ensure that the most suitable parts and processes are selected to allow the device to operate for as long as possible under the harsh conditions of space.

Knowledge of what has already been designed and produced, using the technologies chosen, would allow the most advanced design to be produced in the minimum amount of time. Finally, as altering a design often requires a large amount of time it was also imperative to identify and understand any relevant standards with which the board will need to comply.

2.2 Prior CubeSat On-Board-Computer systems

There were several commercial CubeSat OBCs available on the market at the start of the project; a comparison of some of these is shown in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>NanoMind A712D</th>
<th>On Board Computer</th>
<th>CubeComputer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Company</td>
<td>GOMspace</td>
<td>Innovative Solutions In Space (ISIS)</td>
<td>Electronic Systems Laboratory, University of Stellenbosch</td>
</tr>
<tr>
<td>Processor</td>
<td>ARM7 core</td>
<td>ARM9 core</td>
<td>Arm Cortex-M3</td>
</tr>
<tr>
<td>Processor speed</td>
<td>8-40 Mhz</td>
<td>400Mhz</td>
<td>4-48Mhz</td>
</tr>
<tr>
<td>Interface busses</td>
<td>CAN, I2C, SPI</td>
<td>I2C, SPI, UART, ADC, PWM, GPIO, USB host &amp; slave</td>
<td>2xI2C, CAN</td>
</tr>
<tr>
<td>Other features</td>
<td>RTC, PWM driver, sun-sensor inputs, rate-gyro inputs, microSD</td>
<td>microSD card socket and FRAM storage</td>
<td>microSD card,</td>
</tr>
<tr>
<td>Pinout available</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Fault tolerant capabilities</td>
<td>None specified</td>
<td>Redundant SD card storage and reliable FRAM storage</td>
<td>Single Event Upset (SEU) protection through an FPGA based error detection and correction system, Single event latchup protection through current monitoring</td>
</tr>
</tbody>
</table>
None of the OBCs compared in Table 1 feature multiple processors for fault redundant applications. This indicates that perhaps processors failing are less of a concern than originally thought. In these OBC’s product information, the processors used are not explicitly stated as being radiation and fault resistant. As this would be a major product feature, it is unlikely to have been omitted from the product’s literature, this further lends weight to the idea that processors failing in space are not of concern. Another potential view is that on a CubeSat mission, reliability is of lesser importance than a larger mission with a correspondingly higher cost is. As such, redundant processors may have been excluded simply due to cost.

It is interesting to note that the two boards with fault tolerant features specified in the product literatures appear to have only protected against errors in the on board memory storage. This is achieved through a variety of methods, including redundant storage, FRAM based storage (FRAM is better explained in Section 2.3, a dedicated error detection and correction system, and latchup protection.

Latchup is an error condition that can occur in CMOS based semiconductors. NPN and PNP transistors pairs enter a conducting state that creates a low impedance path that was not intended in the devices design [9]. The transistors can be part of the devices design, or more commonly parasitic [9]. Once in this state, it is often self-perpetuating and cannot be resolved without resetting the device’s power [9]. It is caused mainly by the devices power supplies not starting in the expected order, or due to a charged particle causing the intrinsic transistor to turn on [9]. The latter cause is referred to as a Single Event Latchup (SEL) as once reset, it will not re-occur unless another charged particle causes the error again [9]. Once in the latchup state a large current is able to flow across unintended paths in the device, even if quickly resolved this can easily lead to damage [9]. The Raspberry Pi is susceptible to a latchup error caused by the power supplies starting in the incorrect order; they should all be started at the same time or with the highest voltage supplies first [10].

Of the group of OBCs compared the ‘On-Board-Computer’ made by ISIS has the most powerful processor: an ARM9 core running at 400 MHz [7]. The Raspberry Pi uses a 700 MHz ARM11 core [11], meaning it will be more powerful than the OBCs’ compared. This alone adds a unique selling point to the device, when it is considered that the device has two of these processors it is definitely superior in terms of processing capabilities.

### 2.3 Available Microcontroller technologies

The OBCs compared in section 2.2 all appear to be standard COTS microcontrollers without any fault or error redundancy. Consideration needs to be applied to the choice of technology used in the overview microcontroller required and if a COTS device is appropriate, as this could be a single point
of failure rendering the dual Raspberry Pis useless.

Atmel have a radiation hardened processor family based around the European Space Agency’s (ESA) LEON2 processor capable of running at up to 100 MHz with has “radiation capabilities up to 300 krads” [12]. This appears to be quite specialised and will likely be difficult to acquire, certainly at a reasonable price. Atmel also have a commercial range of microprocessors called the AVR line, these are not explicitly marketed as being radiation resistant.

Similarly, Intersil also feature a specially radiation hardened processor capable of operating up to 5 MHz with radiation capabilities up to 100 krads [13]. Similarly to the Atmel radiation hardened processor this also appears to be quite specialised.

Texas Instruments market a range of their MSP430 microcontrollers as being “ultra low power,” this is achieved through the use of Ferroelectric Random Access Memory (FRAM) that is inherently very reliable, it is impervious to magnetic and electric fields, and has some radiation resistive properties [14]. It is interesting to note that the ISIS OBC uses FRAM as part of its fault resistant storage options. There is also an OBC system available using the MSP430 architecture, but not using the FRAM variant [15].

There is also a PIC variant of the MSP430 system also available [16]. The PIC line is produced by Microchip and a wide variety of microcontrollers from 8 bits to 32 bits, with or without DSP capabilities. Microchip does not explicitly market its PIC microcontrollers as radiation resistant.

The ARM Cortex-M series is almost universally available from every manufacturer. Each manufacturer has different peripherals available in their devices. Whilst the ARM Cortex-M devices can be extremely powerful, they can also be very complex, which may delay development of the embedded software. A device with known good support would need to be chosen.

Similar to the ARM Cortex-M series there is the ARM Cortex-R series. Designed for use in safety critical applications they have two processors that can operate in lockstep with automatic error detection [17]. These processors are based on the same architecture as the Cortex-M series and as such are equally as complex, compounded further by their dual core architecture.

2.4 Prior work on the use of two Raspberry Pi’s as a fault redundant system
SSC have been working on a system to allow two Raspberry Pi’s to communicate reliably together and to a microcontroller via the use of a multiplexer that is controlled by the microcontroller. This system is becoming more reliable, although they have had to overcome several hurdles in the communication via Universal Asynchronous Receiver/Transmitter (UART) by implementing a more fault redundant communication protocol.
They also have the ability for the microcontroller to control the power for the Raspberry Pi’s allowing them to be on only when required.

Elsewhere distributed computing clusters have been built using Raspberry Pi’s on several occasions [18]. Instructions are also available to create a failure resilient web server cluster using Raspberry Pis [19]. It appears that this application for Raspberry Pi’s is less common and hence there are little to no examples available.

2.5 Applicable Standards

Two of the OBCs compared in section 2.2 claimed to be compliant to the PC-104 standard [7] [8]. Figure 1 and Figure 2 show the mechanical outlines for a PC-104 compliant board and the GOMspace nanomind respectively. The other OBCs that have been compared have a similar outline. It is clear that main connector position and layout is different indicating that the OBCs are not PC-104 compliant.

Figure 3 shows the mechanical outline for a PCI-104 complaint board. This outline resembles that of the GOMspace nanomind indicating this is the standard on which the OBCs are based. The pinout however is radically different to that of the PCI-104 standard [20] [21]. As the board is not compliant to the electrical specifications of the standard, the board cannot claim to be PCI-104 compliant [21].

There is a CubeSat specific specification for this variety of OBC’s and other boards they interface with. Pumpkin Inc. have released a CubeSat Kit PCB Specification that is based on the PCI-104 standard [22]. This is the specification to which the OBC being built should comply with in order to ensure compatibility with other products available on the market.

Whilst the CubeSat Kit bus defines the outer dimensions header location on the PCB, it does not define requirements such as minimum track width and distance between tracks [22]. The ESA has released a standard created by the European Cooperation for Space Standardization (ECSS) that specifies design rules for PCBs [23]. This document should be considered whilst producing the PCB design, as compliance early on in the design process is much easier than modifying a product to comply.
2.6 Summary

Knowledge of the microprocessor technologies available has aided greatly in making an informed decision about which microcontroller should be used in the OBC. The standards discussed will ensure that the board is physically compatible with other boards that form a CubeSat. They should also ensure that the board is designed to a suitable standard such that it can be used in a space environment, even though this revision is unlikely to be used on a mission. It is crucial to understand the unique and challenging environment in space, knowledge of the likely failure modes have allowed for a design to be made that takes these into account.
3 RPI-CUBESAT DEVELOPMENT

3.1 Physical constraints

The project requires that two or more Raspberry Pi compute modules are placed on a CubeSat kit sized PCB. The PCBs maximum dimensions are 90.17 mm wide and 95.89 mm tall, although the available space is less due to cut-outs, connectors, and mounting holes [22]. There is also a height restriction on the top of the PCB of 8.78 mm [22] [24]. The bottom of the PCB is restricted to as little as 4 mm in some areas [22].

The Raspberry Pi compute module including mounting socket maximum dimensions were determined as 35 mm wide, 72 mm tall, and protruding from the PCB by 6 mm. These dimensions were determined by measuring a Raspberry Pi compute module mounted on a development board.

Due to the maximum height of the Raspberry Pi compute modules when mounted, they could not be placed onto the bottom of the PCB. Figure 4 shows that two Raspberry Pi compute modules can be placed onto the top side of a CubeSat kit sized PCB. The majority of the available space is occupied, meaning that most other components will require mounting on the bottom of the PCB.

![Figure 4 - Physical layout of two Raspberry Pi compute modules on a CubeSat Kit sized board. Derived from [22].](image-url)
3.2  Microcontroller selection

Following the examination of microcontroller technologies used in space in section 3.2 a decision of which microcontroller is best suited to this application needed to be taken. It was decided to use Texas Instruments MSP430 microcontrollers built using FRAM technology. The extreme reliability presented by this technology in combination with its inherent radiation resistant characteristics meant the microcontroller could be used with reduced risk of becoming a single point of failure in an otherwise resilient system. The fact that the microcontrollers are commercially available to consumers through large retailers such as Farnell makes it easy to acquire and likely to remain so. Being commercially available also presented the benefit of easy access to programming toolchains and community documentation in addition to that released by the manufacturer.

The part that was chosen in particular was a MSP430FR6989IPZR: it is one of the largest parts in the FRAM range, providing some of the most storage and processing capabilities. This part was chosen in particular as it has a large amount of general purpose input/output (GPIO) pins available despite being in a package that could be soldered relatively easily and was available from the major retailers ensuring an easy supply throughout the project.

3.3  Requirements for OBC’s interface busses

As the GomSpace nanomind has been previously used by SSC and SSTL on the STRaND-1 project [3] it was decided that the interface capabilities of the OBC should be based on it. The GomSpace nanomind has several connectors on that allow for a range of different functionality [20] however the main interface busses that will need replication are present on the CubeSat kit bus header. These busses are: CAN, SPI, UART (2x), and I2C (2x) [20]. As the OBC is a proof of concept device that requires three devices on one board able to communicate on the interface busses it was decided that not all of these busses needed implementation. As I2C could be used to develop the OBC and would also provide communication to multiple other devices within a CubeSat system with the minimum amount of connections it was chosen to be the main interface bus on the OBC.

3.4  Overview of the OBC system

The OBC has two microprocessors and a microcontroller that need to communicate between themselves and to other devices within the stack of boards, which comprises a CubeSat. To ensure that the OBC system being designed was manageable, would operate as expected, and to understand what elements of the circuit would need to be created an overview diagram of the system architecture was created. This is shown in Figure 5.
The ability to control the power supplies for the Raspberry Pi compute modules and the ancillary hardware allows the OBCs power consumption to be reduced to the bare minimum when required. Neither the Raspberry Pi compute modules nor the MSP430 have peripherals that allow for multiple master busses, hence multiplexers are required in order to enable multiple master devices to co-exist on the communication busses. The Raspberry Pi compute modules will need to request access to the communication busses.
busses through the MSP430; this does allow for the processors’ priorities for access to be configured dynamically. The UART bus will be used for control and setup of the Raspberry Pi compute modules by the MSP430, requests and other data may be sent in the other direction after the initial setup. It may be possible to expand this bus and its control scheme to allow the Raspberry Pi compute modules to take control in certain circumstances and then flash a new firmware onto the MSP430 via its built-in boot loader.

The FRAM memory has been included to allow for shared access to files between the Raspberry Pi compute modules. FRAM was chosen as the suitable technology for this purpose by the same rationale for the choice of the MSP430.

3.5 Pinouts of large components and main CubeSat kit bus header

To utilise as many features as possible on the Raspberry Pi compute modules and the MSP430 whilst allowing for more development work in the future the devices pinouts required defining. As each device has multiple uses for each pin, the pinout is designed to allow for the maximum amount of relevant functions to be used. This process was also completed for the main CubeSat kit bus header for the same reasons.

The pinout definitions created are shown in Appendix 2.

The initial development work described has enabled the creation of the schematic. The physical layout of the PCB with two Raspberry Pi compute modules on will be invaluable for the PCB layout.

3.6 Schematic creation

Due to prior experience in its use, KiCad was used for the schematic capture and PCB design processes. KiCad is a freely available open source software that can be used without restriction. It is one of the most popular free Electronic Design Automation (EDA) tools available and is under active development with CERN being a highly active contributor [25].

KiCad is a suite of several programs, each program specialising in a particular step in the electronics hardware design process. First, a schematic is captured in Eeschema, from which a netlist file describing the circuit is generated. The components used in the schematic are then associated with their device footprints using Cvpcb, this is then appended to the netlist file. Finally, the netlist is imported into Pcbnew where the PCB can be designed and routed. From Pcbnew the Gerber files required for having a PCB made are output. Other tools such as a Gerber file viewer and a PCB design calculations tool are included.

The complete schematic design is available in Appendix 3. All components used in the design are Commercial-Off-The-Shelf (COTS) and are widely available through major distributors, as with the MSP430FR6989 this ensured an easy supply of parts at short notice throughout the duration of the
project. The schematic is designed using a hierarchal structure: the first page is the top level of the hierarchy defining the connections between the different hierarchal blocks. The schematic blocks for the Raspberry Pi modules are mirrored, ensuring that the schematics are always identical. Similarly, the power switches in the power schematic page are contained in another sub-module and are mirrored.

The connections in the top-level hierarchy sheet mostly follow the design outlined in the system diagram Figure 5. The MSP430 schematic is strongly based on the device’s datasheet and is designed to use the Texas Instruments Spy-By-Wire (SBW) two-wire interface for programming [26]. Similarly, the Raspberry Pi schematics are based on the hardware design guide and the schematics of the Raspberry Pi Foundation’s compute module IO board [10] [27]. No peripheral lines that would require length, phase or impedance matching have been used due to the KiCad version being used not being capable of designing tracks with these restrictions. Conveniently, not many peripheral lines that require these considerations would have been immediately useful in this design iteration.

Figure 5 depicts the UART between the Raspberry Pis and the I2C bus access being controlled by multiplexers arbitrated by the MSP430. The circuit schematic deviates from this design slightly as analogue switch integrated circuits are used in place of multiplexers. This design choice was made in the interest of reliability as it distributes the control of the busses across several devices, meaning that if one ceases to operate the bus can still be accessed by the other two processors. Even if the switch failed in a mode that constantly connected the device to the bus, which could potentially cause failure of communications over that bus due to multiple masters, communications could be restored in software by disabling the I2C peripheral on the processor itself.

Many components used in the schematic are not provided as part of the default KiCad libraries. A custom library containing these components was created. Simple components were often based off similar components available in the default libraries and modified as required using the built in editor. Larger components that are more complex were created using an online tool that could assign the large amount of pins used far more quickly than the built in editor could. This online tool was also capable of importing the pin information from a spreadsheet, allowing the pinout information previously gathered in section 3.5 to be used.

Whilst most modules in the schematic could be based off the manufacturers’ information, or the overview architecture shown in Figure 5, some modules needed to be completely custom designed.

3.6.1 Latching power switch with overcurrent detection

As discussed in section 2.2, space is a harsh environment where device failures caused by charged particles is a considerable problem. Of the many single event effects, we will focus on two primary failure types: Firstly a Single-Event-Upset (SEU), caused by a charged changing the state of a signal
line or bit in a semiconductor. The other is a Single-Event-Latchup (SEL) where a charged particle causes a unintentional high current path to form in a CMOS device due to the intrinsic PNP and NPN transistors in the devices substrate. This high current path can quickly destroy a device, and as such needs careful consideration to avoid it. One method of handling a latchup event is to detect its occurrence, characterised by an abnormally large current draw, and immediately removing the power from the affected device [28]. A shunt switch can also be closed to redirect any remaining current away from the affected device [28].

![Figure 6](image.png)

**Figure 6 – New latching switch design with overcurrent detection capabilities**

The circuit shown in Figure 6 combines this latchup avoidance methodology with a latching power switch, allowing the power to be turned on and off as demanded. Should an overcurrent event occur the power switch is automatically turned off and cannot be turned back on until the enable line is first brought low and then high again.

The circuit is constructed from several simple circuits, first the current is sensed by amplifying the voltage dropped across a small value resistor. The output from this current sense amplifier is also connected to an analogue to digital peripheral pin on the MSP430, allowing the current draw to be monitored in software. A second operational amplifier is then used as a comparator. Should the voltage output from the current sense amplifier rise above the threshold that is set by the potential divider constructed from two resistors then the latching power switch is forced off. The operational amplifiers chosen for this purpose have internal current limiting on their power supplies, which
should avoid them being subject to a SEL themselves.

The latching power switch is based on a design used in UOSat2 where a set of PNP and NPN transistors form a self-sustaining loop to switch on the power rail [29]. The overcurrent cut out is caused by the transistors’ self-sustaining loop breaking down when too much current is drawn [29]. As it was desired to be able to monitor the current draw in software a separate current sense amplifier had been introduced. As the overcurrent condition no longer needed to be handled by the transistors themselves it was decided to use MOSFETS instead, this simplified the circuit design and reduced power loss as the MOSFETS can conduct in their saturation region.

The software control input on the power switch is edge triggered, as it was in the original UOSat2 design [29]. When the fault signal from the comparator is low, and the diode D1 is therefore reverse biased, a rising edge on the enable line causes the n-type MOSFET Q2 to turn on. This is turn turns on the p-type MOSFET by connecting its gate to ground, connecting the output line to the power supply rail, which in turn forces the n-type MOSFET to remain on. Thus completing the self-sustaining loop. Similarly, a falling edge at then enable input causes the power switch to be turned off. A high signal from the comparator also forces off the p-type MOSFET.

A shunt switch has not been added in this design due to limited amount of space available on the PCB. A redundant system of power switches formed of two p-type MOSFETS in parallel on the high side of the load and two n-type MOSFETS on the low side of the load could have been used. This was also not added due to limited space.

3.6.2 Overcurrent detecting automatic reset

As the MSP430 is designed to be constantly powered, the previous switch design is unsuitable for controlling its power rail. As such, a modified design was created that would reset the MSP430 power in the event of an overcurrent condition, otherwise it is permanently on. This design is shown in Figure 7. It is similar in design to the previous circuit, with the current measured by the amplification of voltage across a sense resistor and the threshold implemented by a comparator. The circuit differs in the last section, which is a monostable circuit formed of discrete components. The use of discrete components was chosen in this design to mitigate the risk of latchup within this circuit.

The monostable is triggered when the output from the output of the comparator falls low in the event of an overcurrent. This forces the p-type MOSFET Q2 on, which then forces the output p-type MOSFET Q3 off. The N-type MOSFET Q1 is also forced on, maintaining the low signal at the input even when the output of the comparator returns high. The RC network formed by C1 and R8 then charges up, once it reaches the threshold voltage of Q2 the output is then re-enabled and the circuit reset.
The monostable is designed to turn off the power to the MSP430 in the event of an overcurrent condition for 200ms. This timing was chosen as 100ms was suggested as a suitable length of time for fully extinguishing a latchup condition [28]. As the reset will have disrupted the MSP430’s operation its recovery time is not important; hence a longer time was chosen to ensure a safe margin for error.

![Figure 7 - Automatic reset in the event of overcurrent](image)

### 3.7 PCB Design

Once the schematic capture process was completed the components used needed to be associated with their device footprints. This was done using the Cvpcb program built into KiCad. Many components using standard footprints, such as the 0603 surface mount passive devices, were quickly added from the stock footprint libraries. Several components footprints were not available in the standard libraries and were instead their footprints were designed using information available in their datasheets. Instead of using the built in footprint editor in KiCad it was decided to use an alternative program called Madparts [30]. Instead of placing each pin individually as in KiCad, using Madparts the entire footprint could be designed using code to describe the footprint. This greatly sped up the design of some footprints. An example of the LQFP100 footprint being designed for the OBC is shown in Figure 8. Whilst these footprints could have been located online, the licences they are released under would have to have been considered carefully. They also would have needed to be checked against the datasheets with greater scrutiny than a custom footprint, negating some of the time saving of using them.
As the Cubesat kit bus PCB specification required a PCB of specific dimensions with several cutouts, the outline of the PCB could not be designed in KiCad. Instead, the diagram in Figure 4 was modified to contain only the outline of the PCB. The diagram was created in an open source program called QCAD, which outputs .dxf files. The .dxf file was then converted to the KiCad .brd format using an online converter. This could then be opened directly by KiCad and imported into the design.

As well as the Cubesat kit bus PCB specification, the rules for PCB designs released by ESA are also discussed. It was decided to not design the PCB to comply with this standard due to the initial time investment that would be required.

The board was designed as a four layer PCB. The layers from bottom to top consist of: signal and components, ground plane, power planes, and signals and components. The ground plane covers the entire layer, providing an element of signal isolation between the layers. On the power plane layer there are several planes, as the four switched power lines also run through this level. Due to the low amount of clearance between the Raspberry Pi compute modules and the PCB, very few components were placed in this gap. This meant that the majority of components needed to be placed on the underside of the PCB, reducing the space available.

It became obvious as the PCB design came to completion that the initial design was too complex for a four-layer PCB design in the time available. The SPI communications shown in Figure 5 between the Raspberry Pi’s were removed as they were unrequired for the initial proof of concept design and needed to cross the entire board which would have caused difficulties for the remainder of the layout.
3.8 Summary of design

The circuit schematic and PCB layout has been fully designed, the primary tools used during this process were KiCad, QCAD and Madparts. The use of Madparts to automate the creation of footprints for use in the PCB design saved a significant amount of time. Dedicating the time during this project to learn this new tool was most certainly a good investment.

Once it has been assembled, the PCB that has been designed should be capable of the following functionality:

- Programming and operation of the MSP430 microcontroller
- Operation of the Raspberry Pi’s
- Independent UART communication for the MSP430 and Raspberry Pi’s via the debug header
- UART communication between the MSP430 and the Raspberry Pi’s via the analogue switches
- I2C communication from the MSP430 and Raspberry Pi’s to the FRAM memory and the Cubesat header, via the analogue switches using the MSP430 as a bus arbiter.
- MSP430 can start or stop the Raspberry Pi’s by the enable line.
- Independent power switching of: both Raspberry Pi’s, and the UART and I2C bus switches.
- Overcurrent detection and response individually on the MSP430, Raspberry Pi’s, and the UART and I2C switches.
- MSP430 can detect operation failure of the Raspberry Pi’s via a watchdog timer
- Independent current monitoring of the MSP430, both Raspberry Pi’s, and the UART and I2C switches.
- Using a low frequency oscillator use the MSP430 as a RTC.

The PCBs were sent for production by a Chinese facility due to the cost of a UK based manufacturer. The lead times were not greatly affected due to this and the quality of the PCBs received were acceptable for a proof of concept.
4 BUILD AND TEST

4.1 Final board

As mentioned in section 4.7 two design iterations of the PCB were required due to a mistake with the SODIMM module footprints. The PCB design used to construct the final proof of concept is shown in Figure 9 and Figure 10. The first iteration of PCB design is shown in Appendix 5.

![Annotated top view of PCB with corrected SODIMM spacing](image)

Figure 9 - Annotated top view of PCB with corrected SODIMM spacing
4.2 PCB assembly

The proof of concept OBC was assembled using mostly hand soldering. The only component that was not hand soldered is the MSP430FR6989; instead, the focused IR rework station available in the undergraduate labs was used.

To ensure the PCB was assembled correctly it was assembled in stages. Firstly, the two most complex components, the MSP430, and the Raspberry Pi SODIMM headers were soldered on. Once these were successfully soldered on, the risk of the board being written off was reduced for the remainder of the components. The decoupling capacitors, power circuit, and programming header were then soldered onto the board. It was then checked that the MSP430 could be accessed and programmed. A simple demonstration program was programmed onto the MSP430 that turned on all the power supplies on the OBC and flashed one of the MSP430 debug LEDs to indicate the processor is running.

A Raspberry Pi power supply was the next to be soldered on; its output voltage was checked before soldering on the other power supply. Similarly, the UART and I2C buses power switch was soldered and checked. Finally, support components such as the Raspberry Pi decoupling capacitors and the debug UARTs header were added.

This building and testing in stages allowed for the individual power rails on the board to be
checked easily and problems easily identified and rectified if they occurred. During the build process, only two resistors were placed in swapped positions, which was quickly found and rectified.

Once the PCB had been fully assembled, the Raspberry Pi compute module was placed onto the board for the first time. After a problem that stopped the Raspberry Pis booting properly was found and rectified as will be discussed in section 4.7.2, the Raspberry Pi could be accessed via the debug UART interface. A load test program that simply calculated factorials of ever-increasing numbers was written in C and compiled on the Raspberry Pi whilst still on the board. This load test caused the Raspberry Pi CPU to max out at 100%, causing a power draw of roughly 0.6W in total. The load test was successfully left overnight as a soak test.

The fully assembled PCB with two Raspberry Pi compute modules fitted is shown in Figure 11 and Figure 12.
The MSP430 is programmed in C, which is then compiled using the GCC toolchain for the MSP430 series released by Texas Instruments. The MSP430 on the OBC board is then programmed using an open source program called MSP-debug using a FRAM experimenter board as the hardware programmer.

Instructions on setting up this toolchain are available in Appendix 6.

The C code written for the MSP430 is available in Appendix 7. It is written to be primarily event driven using interrupts. As such there is a timer interrupt every 500ms that flashes the debug LED and increments the watchdog timers if the Raspberry Pis are turned on. There are interrupts on GPIO pins changing state, such as the watchdog timer reset pins. These serve several different functions, the most notable being resetting the Raspberry Pi watchdog timers and detecting if the power switches turn off without being instructed to, indicating an overcurrent condition. UART receive functions are also interrupt driven, the received character is processed and if it matched with a known control character then a flag against the requested action is set. UART transmit functionality is interrupt driven, but the functions that send the characters wait on the confirmation interrupt, causing the code to wait upon the transmit to complete anyway.

Most interrupts simply change a state variable, which is then handled later by the main process.
continually running on the MSP430. This keeps the time servicing interrupts short, avoiding conflicts occurring when two or more interrupts are called within a small space of time.

There are some elements of the MSP430 software that have not been fully developed; primarily the MSP430 to Raspberry Pi communication link via UART is non-functional. Calling the function to transmit a character on this communication link would cause both the MSP430 UARTS to stop operating and would occasionally stop the flashing debug LED indicating the timer interrupt would also stop. The cause behind this is currently unknown.

The I2C communication on the board, which is arbitrated by the MSP430, has not yet been implemented due to time restrictions. This would have operated by the processor requiring the I2C bus requesting access from the MSP430, including the MSP430 itself, if the I2C bus is available the relevant analogue switch would be turned on and the bus could be used.

Finally, the current monitoring available in the anti-latchup switch circuits has not been implemented. This is again due to time restrictions. In order to implement this feature code would need to be written that would allow the ADC input pins to be read, this would then need to be converted into a current reading using known constants.

4.4 Debug LEDS

As will be discussed in section 4.7.4 the debug capabilities on the MSP430 were limited. As such, the debug LEDs highlighted in Figure 9 became invaluable in the development process. The centre LED was set to flash when the timer interrupt on the MSP430 is called, indicating the MSP430 is operating correctly. Similarly, the outer MSP430 debug LEDs were set to flash when the Raspberry Pi sent a signal to reset the watchdog timer, there is an LED for each Raspberry Pi.

Initially one of the Raspberry Pi debug LEDs was set up to flash once the Raspberry Pi had successfully booted. This proved useful in indicating whether the Raspberry Pi was operational if the debug UART was not operational or disconnected. This LED function remains as it occasionally proves useful and the LED is not required for another purpose yet.

4.5 Programming Raspberry Pi’s (Linux)

The Raspberry Pi software is best considered as split into two separate elements: the platform software, and the application software.

Platform software is the low-level software such as the boot loader and Linux OS. Arch Linux was chosen for use on the Raspberry Pis due to prior experience with this distribution. It was also the most suitable due to its characteristic of coming with the absolute minimum software required. This meant that unnecessary software that would only use power without adding value in this instance, such as a desktop environment, would not need to be removed.
Setting up the platform software needed to be done on the Raspberry Pi Compute module IO board due to the USB functionality. The USB host socket was used to access a wireless dongle, allowing any extra software required to be installed. The operating system itself was initially loaded onto the Raspberry Pi by using a jumper to place it in USB boot mode. A program on the computer called “rpiboot” was started and the power to the Raspberry Pi was then connected. The on-board memory was then available on the host computer to be written to directly. The arch Linux OS was then loaded onto the on-board memory using the process described on the official Arch Linux ARM installation guide for the Raspberry Pi [31].

Arch Linux for the Raspberry Pi is set up to be used on a Model A or B Raspberry Pi, the compute modules IO had to be setup for use with the OS. The Linux device tree sets up which pins are assigned for what peripheral use. A custom device tree file, shown in Appendix 8, was used to setup these IO pins. This needed to be compiled and loaded into the boot directory on the Raspberry Pi before it could be used. The command to do this on the Raspberry Pi is:

```
dtc -I dts -O dtb -o /boot/dt-blob.bin dt-custom-blob.dts
```

Once the platform software was setup, the application software needed to be created and setup. As the application software is likely to be highly custom for each application the OBC could be used for the software for this proof of concept is kept simple. Limited only to resetting the MSP430 watchdog and flashing the Raspberry Pi’s debug LED. A simple C application, shown in Appendix 9, was written to toggle a GPIO pin, this was compiled twice, once to toggle the LED pin, and the other to toggle the watchdog reset pin.

Both of the GPIO toggle programs written as application software then had a Linux daemon service created that would start them upon system start up. The contents of one of these daemon files is shown in Appendix 10. This file needed to be placed in the directory:

```
/etc/systemd/system
```

It is then setup to start the program upon every boot using the command:

```
systemctl enable ledtest.service
```

### 4.6 Verification of design, build, and operation

In section 3.8 the capabilities of the hardware designed was listed. If appropriate, methods of testing each capability have been devised and carried out. These tests and their results are shown in Table 2.
Table 2 - Designed capabilities test methods and results

<table>
<thead>
<tr>
<th>Hardware capability</th>
<th>Operating as intended</th>
<th>Verification Method &amp; result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP430 powered &amp; operational</td>
<td>Yes</td>
<td>Power supply to MSP430 was confirmed by measurement taken at the relevant test point. The MSP430 was successfully programed and a test program written to it that flashed one of the debug LEDs, showing it is operational.</td>
</tr>
<tr>
<td>Raspberry Pi powered &amp; operational</td>
<td>Yes</td>
<td>Both power supplies to Raspberry Pis were confirmed by measurements taken at the relevant test points. The Raspberry Pi was proven as operational in both sockets by writing known working software to it on the IO board, then placing the module on the OBC. The flashing debug LED indicated it had booted correctly and was operational.</td>
</tr>
<tr>
<td>MSP430 debug UART</td>
<td>Yes</td>
<td>Communication with the MSP430 has been established via a USB to UART converter. This is currently the main interface to demonstrate the MSP430 functionality.</td>
</tr>
<tr>
<td>Raspberry Pi debug UART</td>
<td>Yes</td>
<td>Communication with the Raspberry Pi has been established via a USB to UART converter. This is currently the main interface to demonstrate the Raspberry Pi functionality.</td>
</tr>
<tr>
<td>MSP430 controls if the Raspberry Pi runs using the enable line</td>
<td>Yes</td>
<td>The MSP430 software was used to set the enable line high, the Raspberry Pi was then shown as operational. The enable line was then set low and the Raspberry Pi was shown to no longer be operational.</td>
</tr>
<tr>
<td>Communication via UART between the MSP430 and the Raspberry Pis</td>
<td>No</td>
<td>Currently does not work, due to MSP430 UART not operating as expected. The root cause has not yet been identified.</td>
</tr>
<tr>
<td>Independent power switching for Raspberry Pis and UART/I2C busses</td>
<td>Partial</td>
<td>The MSP430 was used to set the enable lines high and low to test the functionality. Raspberry Pi power switching causes voltage drop issues, which are discussed in section 4.7.7. The power switch for Raspberry Pi 2 often resets back on when turned off. This indicates there may be a physical difference in construction; no difference has been identified yet. Bus power switching appears to be permanently on. The root cause has not been identified. Either this could be a design fault or more likely, as the Raspberry Pi switches based on the same design partially operate, there is a build error.</td>
</tr>
<tr>
<td>Power is turned off individually if an overcurrent event occurs in a Raspberry Pi or the UART/I2C bus devices.</td>
<td>Partial</td>
<td>Dead short on raspberry pi power switches causes power to be turned off. Raspberry Pi 1 remains off; Raspberry Pi 2 resets back on, suggesting a physical difference in their assembly. Overcurrent detection appears to be non-functional on the bus power switch. The root cause of this is yet to be identified, but is likely to be caused by the same fault that stops the switching function operating correctly. MSP430 overcurrent state is operational, but currently disabled due to nuisance tripping caused by a too conservative current limit.</td>
</tr>
<tr>
<td>Current monitoring</td>
<td>No</td>
<td>Software on the MSP430 has not yet been developed to test this capability due to time constraints.</td>
</tr>
<tr>
<td>MSP430 I2C communication</td>
<td>No</td>
<td>Software on the MSP430 has not yet been developed to test this capability due to time constraints.</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Raspberry Pi I2C</td>
<td>No</td>
<td>Communication No</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>MSP430 software setup to increment the watchdog counter every 500ms. If the Raspberry Pi does not reset the counter by toggling a pin, it is considered to have failed. Currently this occurs after an arbitrary choice of 30 seconds.</td>
</tr>
<tr>
<td>MSP430 can detect</td>
<td>No</td>
<td>Raspberry Pi failure</td>
</tr>
<tr>
<td>MSP430 RTC</td>
<td>No</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

### 4.7 Problems encountered during build and development process

Several problems were encountered during the build and development process. Some had to be overcome to continue the development process. The most notable problems are detailed in this section.

#### 4.7.1 Incorrect component footprints for SODIMM headers

When the boards first arrived, they were designed using an incorrect footprint for the SODIMM socket, which is used to interface the Raspberry Pi with the main board. This is a large complex component and the design error caused the pads to be hidden underneath it: meaning it could not be fixed readily through an adapter board or the use of kynar wire straps. It was attempted to solder these components on using a reflow oven in the hope that the rear of the SODIMM sockets pins overlapped the pads: this proved to be unsuccessful. The cross section photographs of the SODIMM sockets on the old and new PCBs shown in Figure 13 and Figure 14 show that the pins did not line up with the pads and hence why the reflow oven soldering failed. Fortunately, this issue was quickly caught, as a new revision of board had to be produced to rectify the issue.
4.7.2 Raspberry Pi’s failed to boot due to unused USB On-The-Go functionality

The Raspberry Pi compute module was proved as operational on the official IO board. However when transferred to the OBC PCB it would not load correctly. The cause of this issue only became apparent when the Raspberry Pi’s boot debug information was successfully accessed via the UART. A repeating error message was being constantly output as shown in Figure 15. This issue was caused by the Raspberry Pi’s USB On-The-Go functionality being activated, which the Raspberry Pi’s do not have a driver for, causing them to fail to boot properly. USB On-The-Go functionality is activated when the USB id pin is left floating, standard USB operation as the host is selected by tying the id line to ground [32]. Fortunately, a similar issue has been seen previously on another project whilst the author was on placement at Tactiq Ltd [33]. As such, a resistor was included in the schematic for both Raspberry Pi’s USB id line to be tied to ground. It was marked as a do not fit component as it was unknown if its inclusion would have unwanted effects. Fitting a 0Ω link on this unpopulated...
footprint forced the Raspberry Pi’s USB interface into host mode and allowed it to boot completely.

![TTYUSB0 PuTTY](image)

**Figure 15 - Repeating error message stopping the Raspberry Pi’s from booting**

4.7.3 MSP-debug would not recognise the MSP430 processor

The MSP430 toolchain was first set up on an Arch Linux installation using the “mspgcc-ti” package from the Arch User Repository (AUR.) As will be discussed in Section 4.7.4 GDB would not run under this environment. As such, it could not be used to program the MSP430FR6989. Instead, an alternative open source program called MSP-debug was used. Ideally, this command should connect to the MSP430 on the target board and automatically detect which device type it is via the programmer being used:

```bash
mspdebug rf2500
```

The command instead provides an error stating that the target device is not recognised. To resolve the issue the target device detected needed to be manually overridden. It was found that the id of the lesser device type in the same family could be used instead of the automatic detection. The command that enabled the target device to be connected to and programmed is:

```bash
mspdebug rf2500 --fet-force-id MSP430FR5969
```

It is unknown if this issue is a result of a software or hardware issue as GDB running on Windows also provided a similar error message. If it is a hardware issue, it is likely because the programmer being used is relatively old and its firmware may not support the microcontroller used.

4.7.4 GDB, part of the MSP430 toolchain, would not work

Part of the GNU Compiler Collection (GCC) toolchain for the MSP430 is the GNU project debugger
(GDB.) The GDB program would provide several error messages about missing python dependencies when run. The GDB software would then appear to enter the command line interface successfully but could not be used to create a debug interface onto the MSP430 device. This was attempted for both the MSP430FR6989 used on the OBC and the MSP430FR5739 available on the FRAM experimenter board being used as a programmer.

The GCC toolchain for the MSP430 was also installed in a Windows 7 environment. When GDB was then run no errors were presented. However when attempting to connect to the MSP430FR6989 an error message was presented claiming that the identified device was not supported. The method used to force a MSP430 programming software to recognise the device used is discussed in section 4.7.3.

It was also attempted to recompile the GCC toolchain in the Linux environment, as it had been originally compiled for use on a RedHat distribution. The compilation process failed whilst trying to compile the GDB software with an error message claiming that the library “expat” could not be found. This library was correctly installed however. Following the several failures in getting a full operational toolchain running it was chosen to use MSP-debug for programming and to proceed without debug capabilities.

4.7.5 Damage to MSP430 debug and programming header

As the debug and programming header for the Spy-Bi-Wire (SBW) interface on the OBC had to use a small 1 mm pitch header due to size constraints an adapter had to be produced. This would allow the 1 mm pitch header to connect to the 2.54 mm programming header on the MSP430 FRAM experimenter board. The adapter board was subject to rotational forces due to its design, which damaged the 1 mm pitch receptacle on the adapter board. The adapter board was repaired and the receptacle repaired using super glue. Unfortunately, it was mated with the OBC before the super glue had fully cured and when they were unmated, the 1 mm pitch receptacle was damaged irreparably.

To continue developing the MSP430 software without a second adapter board it was decided to add a second programming header overhanging the side of the OBC, this time 2.54 mm pitch to avoid another adapter being required. Ironically, this header is affixed to the OBC PCB using super glue.

4.7.6 Raspberry Pi’s only have drivers for one UART

Raspberry Pi’s have two UART peripherals available, one full UART and a mini UART with reduced functionality. It was intended to use the full UART for communication between the MSP430 and the Raspberry Pi’s, whilst the mini UART provided the debug interface via the debug header. It was found that the full UART operated as expected, by the mini UART would not operate due to a lack of drivers in Linux.
Further research revealed that these drivers are not available officially; however, an individually developed driver is available on a blog [34]. This driver needs to be compiled against the Linux kernel for the Raspberry Pi and then inserted as a kernel module [34]. The Linux kernel was downloaded and compiled overnight on the Raspberry Pi, the driver module was then compiled against the kernel. When it was attempted to insert the driver as a kernel module a version mismatch error was given. Despite the kernel being the same version as what was running on the Raspberry Pi it had a marginally different version string, causing the module insertion to fail.

Whilst the driver could have been re-compiled or the kernel being used on the Raspberry Pi replaced with the compiled kernel, it was decided that the use of the mini UART was not worth the investment in time required. Instead the debug UART port would not have the Raspberry Pi UART assigned to it most of the time.

4.7.7 Raspberry Pi power switches problem

During the development of the MSP430 embedded software it was found that switching on the Raspberry Pi’s power switches would cause the MSP430 to reset. It was found that the voltage across the entire OBC would drop rapidly when one of the switches was turned on. The drop in voltage across the board is shown in Figure 16.

![Scope trace of voltage supply to OBC when a Raspberry Pi switch is turned on](image)

**Figure 16 - Scope trace of voltage supply to OBC when a Raspberry Pi switch is turned on**

The root cause of this voltage drop has not yet been identified, it is thought that drop in supply voltage is caused by an inrush current. Either caused by the capacitance on the output of the switch or the switch mode dc-to-dc supply required for the Raspberry Pi’s 1.8V rail.

The MSP430 user guide states that the brown out voltage is around 1.8V, the scope trace in Figure 16 shows the voltage only drops to 2.4V. However, the user guide also states that the brown out reset on the MSP430 can also be triggered if the input voltage drops too quickly; it suggests limiting the voltage drop to 0.05V/µs. The falling edge shown in Figure 16 falls at a rate of 0.3V/µs, indicating
this is likely the cause of the MSP430 resetting. It was attempted to use extra capacitance at the MSP430 supply to reduce the rate of the voltage drop, this proved to be insufficient.

The Raspberry Pis had previously been operational before this issue was located as their power was being previously switched on during the MSP430 boot sequence, the MSP430 would continue to reset until the switch successfully latched on. Whilst non-optimal, returning to this strategy allowed the development of the OBC’s functionality to continue.

4.8 Summary

The OBC has been fully assembled and any functionality that is not dependant on software yet to be implemented has been tested. There were several problems discovered as part of the build and testing process, some of these issues have been resolved, whilst others have been worked around.

The core functionality of the board has been proven, with the MSP430 and the two Raspberry Pis operating as expected, independently of each other. However, the UART communication link between the MSP430 and the Raspberry Pis is not operational, meaning that fault redundancy more than simply booting the other Raspberry Pi and having it run the same default program is not possible. For some use cases, this may be all that is required. Because the I2C communication links have also not been implemented in the MSP430 software, the usefulness of the board as a practical OBC is severely limited.

The power switches have displayed unusual behaviour, with three switches of the same design acting in different manners. This suggests that there is a difference in their physical assembly, most likely due to assembly errors. Combined with the voltage drop caused by turning on either of the Raspberry Pi switches, their operation is severely limited.
5 FUTURE WORK

5.1 Development work still required on this design iteration

Before the next iteration of the OBC is designed, several aspects of this design need to be corrected or completed. These are:

- Resolve the issue where the Raspberry Pi power switches cause the voltage input to the board to drop and reset the MSP430
- Locate the cause of the bus power switch being constantly turned on
- Resolve the issue with the second MSP430 UART to enable the communications between the MSP430 and the Raspberry Pis
- Implement the current monitoring functionality in the MSP430 software

Once these have been proven as operational this design iteration can be used as a basis for a second revision. Should it not be possible to get one or more of these functionalities working a different implementation may be required.

5.2 Functionality to be added to the next design iteration

Due to time restrictions, several functionalities that were originally considered have not been implemented in this hardware revision. Similarly, several new functionalities have been suggested for addition.

The functionalities that should be considered for inclusion in the next revision of hardware are:

- A high-speed bus on the Cubesat kit bus header to enable a wider variety of uses for the OBC. (This feature was suggested by a SSTL engineer)
- CAN communication via the Cubesat kit bus header to enable the OBC to be used with a wider variety of other Cubesat boards
- If possible due to space constraints, more robust power switching, with redundant power MOSFETS
- Communication directly between the Raspberry Pis, ideally as high speed as possible
- Potentially a low power communication link between the Raspberry Pis
- A large non-volatile memory that can be written to by both Raspberry Pis
- Make the Raspberry Pi’s on-board memory more fault tolerant using RAID or a similar system
- Use an RTOS on the MSP430 to make additional functionality easier to add and to ensure that all functionality is done in a defined amount of time
It is suggested that in the next hardware revision raised SODIMM sockets are used to enable components to be placed underneath the Raspberry Pi Compute module. This will allow more components to be fitted, which in turn should enable more functionality. If this is done, it may be necessary to increase the amount of layers used in the PCB to accommodate the higher density of signals.

5.3 Alternative architectures

During the development process two alternative architectures have been discussed that may provide advantages over this current system.

One alternative architecture simplifies the current design by making all the devices communicating via I2C on the OBC slave devices and using the I2C device in the radio module as the master. This is suggested by an SSTL engineer in order to improve to reliability of the system as the ground station has direct access to the I2C device in the radio module and hence to the I2C master.

The other alternative architecture is designed to increase the redundancy available in the system and remove the single point of failure of the MSP430 in the current design. It also reduces the complexity of bus arbitration currently required. Either the microcontrollers chosen will need multi-master I2C capabilities, or to be used as I2C slaves as previously suggested. An overview diagram for this potential architecture is shown in Figure 17. This design is also interesting because it is infinitely repeatable, the only limitation being physical space and the amount of power available.
Figure 17 - A potential alternative architecture to the current design
6 CONCLUSION

A new proof of concept On-Board-Computer utilising two COTS Raspberry Pi compute modules has been successfully designed, implemented, and tested. There are still some elements of the board’s functionality that do not operate as desired, particularly the power switches that were designed for this project. The main processing functions of the board have been proven as operational and are highly versatile in their application. Unfortunately, due to the main communication link between the MSP430 and the Raspberry Pi not being operational it is not possible to alter the programs run on the Raspberry Pi remotely. This also means that some fault handling schemes are not currently possible. A failure of a Raspberry Pi can be detected by the MSP430 and in certain applications, if the software on the Raspberry Pi does not need to change, the other Raspberry Pi can be used in compensation.

At the start of the project a set of objectives were set out, the projects progress has been compared against these objectives in Table 3.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Completed</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prove that two Raspberry Pi compute modules can fit within the space available on a Cubesat.</td>
<td>Yes</td>
<td>A diagram using measurements of a Raspberry Pi compute module and details from the CubeSat kit bus PCB specification is shown in Figure 4 in Section 3.1.</td>
</tr>
<tr>
<td>Produce an architecture design for the OBC using two Raspberry Pis and a smaller microcontroller</td>
<td>Yes</td>
<td>A diagram showing the architecture design has been produced and is shown in Figure 5 in Section 3.4.</td>
</tr>
<tr>
<td>Design the circuit schematic for the proof of concept OBC based upon the architecture design</td>
<td>Yes</td>
<td>A circuit diagram has been produced; its production is discussed in Section 3.6. The circuit diagram is shown in Appendix 3.</td>
</tr>
<tr>
<td>Design a Printed Circuit Board (PCB) for the proof of concept OBC using the circuit schematic</td>
<td>Yes</td>
<td>A PCB design has been produced; its production is discussed in Section 3.7. The PCB design is shown in Appendix 4.</td>
</tr>
</tbody>
</table>
Produce software required in order to demonstrate the OBCs fault tolerant capabilities  

Yes  
Software for both the MSP430 and the Raspberry Pi has been produced. It production is discussed in Sections 4.3 and 4.5 respectively. The MSP430 software is listed in Appendix 7. Software for the Raspberry Pi is listed in Appendices 8, 9 and 10.

Test and document the capabilities of the OBC  

Yes  
The hardware capabilities have been tested and the results discussed in Section 4.6.

Produce documentation on both the physical OBC and the demonstration software, including the toolchains required for the software  

Yes  
The hardware design is documented in the circuit diagram and PCB design. The software used is also documented. The toolchain used for the MSP430 software is documented in Appendix 6.

The review of the original objectives show that the project was successfully completed and the process was documented. It is also important to consider the original capabilities desired of the proof of concept OBC. These capabilities were graduated into three levels of priority: Primary, Secondary, and Tertiary. The proof of concept hardware’s capabilities have been compared in Table 4 against the original list.

Table 4 - Comparison of original desired capabilities against delivered capabilities

<table>
<thead>
<tr>
<th>Desired capability</th>
<th>Implemented</th>
<th>Results/proof/comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primary</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Have two Raspberry-Pi compute modules available for use on the OBC. These should operate independently of each other</td>
<td>Yes</td>
<td>The Raspberry Pis can operate when enabled by the MSP430. Their operation is not dependant on the other Raspberry Pi.</td>
</tr>
<tr>
<td>Have a microcontroller to control the Raspberry Pi’s operation and monitor the Raspberry Pis for failure</td>
<td>Yes</td>
<td>The MSP430 operates whenever the board is powered. It can control when the Raspberry Pis can run and detect if one of the fails.</td>
</tr>
<tr>
<td>Should a Raspberry Pi fail, the other should be able to take over its tasks</td>
<td>Partial</td>
<td>There is currently no direct communication between the Raspberry</td>
</tr>
</tbody>
</table>
Pis. The MSP430 can detect if a Raspberry Pi were to fail and boot the other one. The Raspberry Pis would have to automatically start the same program for this to operate as fault redundancy.

<table>
<thead>
<tr>
<th><strong>Secondary</strong></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow the Raspberry-Pis’ clock rates to be changed, according to the processing capabilities required and power available</td>
<td>No</td>
<td>The communication link between the MSP430 and the Raspberry Pis is not operational. As such, the Raspberry Pis clock rates cannot be changed, as there is no communication to control the clock rates.</td>
</tr>
<tr>
<td>Allow the Raspberry-Pi’s processing state to be saved to their internal flash memory then suspended</td>
<td>No</td>
<td>The communication link between the MSP430 and the Raspberry Pis is not operational. Without the communication link, the command to suspend to flash memory cannot be sent.</td>
</tr>
<tr>
<td>Allow the Raspberry-Pis to access communication links to the rest of the satellite and resources through arbitration</td>
<td>No</td>
<td>Access due to communication links to the rest of the satellite have not been implemented due to time. The communication link between the MSP430 and the Raspberry Pis was prioritised.</td>
</tr>
<tr>
<td>Have a Real Time Clock (RTC)</td>
<td>No</td>
<td>The communication link between the MSP430 and the Raspberry Pis is not operational. Without the communication link, the RTC information cannot be sent to the Raspberry Pis and therefore has little value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Tertiary</strong></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakout unused, but potentially useful Raspberry-Pi functionality such as the camera interface bus. To allow for extended usage</td>
<td>No</td>
<td>It was decided against this at the circuit design stage. The primary reason is that most of the extra functionality desired required matched length, phase, or impedance tracks. Which the version of</td>
</tr>
</tbody>
</table>
Using the Raspberry-Pi’s communication link, allow for the two modules to operate in a fault redundant mode called lockstep

<table>
<thead>
<tr>
<th>KiCad used could not support.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the Raspberry-Pi’s communication link, allow for the two modules to operate in a fault redundant mode called lockstep</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>The direct communication link between the Raspberry Pis was removed at the PCB design phase as it made the design too complex for a four layer PCB in the time available.</td>
</tr>
</tbody>
</table>

As the tertiary capabilities were removed at the hardware design phase it is reasonable to state that these aims were too advanced for the time available and the tools being used. It becomes immediate apparent with the secondary level of capabilities that the lack of an operating communication link between the MSP430 and the Raspberry Pis has been a significant hindrance. With more time, the underlying issue causing the peripheral not to operate on the MSP430 could probably have been located. Three to four weeks were lost during the project at a vital stage when a second PCB revision had to be manufactured. This delayed the start of the software development process significantly. It is difficult to postulate which capabilities would have been completed had this delay not occurred.

The primary capabilities were graded as such because they were necessary to prove the concept of an OBC with two Raspberry Pis as a fault redundant system. As these capabilities have been mostly completed the original concept has been proven, all that is now required is more development to enhance the capabilities. In its current form, the OBC is not ready for use on a mission. However, with potential consideration towards modified system architectures and further development the OBC could be highly valuable on a mission where its capabilities are required.

Throughout the project, lessons have been learned and skills honed. The primary lesson learned is that all footprints must be checked extremely carefully. Ideally by a second person. It is suggested that in future projects if the components are immediately available they should be placed on a 1:1 printout of the footprint design. If they are not available then every dimension on the datasheet drawing should be checked and marked off, even if it appears inconsequential to the design.

The project has also proven that open source tools are prevalent and powerful enough to be used to great effect within the electronics industry. The primary programs used were KiCad, QCAD and Madparts. Using free open source tools has meant that the large community some of these tools can attract can be harnessed should a problem be encountered. The knowledge gained using these tools is likely to prove highly valuable in future projects, in either industry or academia.

It has been a pleasure to work on this project and with the project supervisor Dr. Chris Bridges. It is exciting to see the potential for the OBC should more development work be done on the project.
7 REFERENCES


8 APPENDICES

Appendix 1. Deliverables, with expected and actual delivery dates

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Expected delivery date *</th>
<th>Actual delivery date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schematic design</td>
<td>26&lt;sup&gt;th&lt;/sup&gt; January</td>
<td>26&lt;sup&gt;th&lt;/sup&gt; January</td>
</tr>
<tr>
<td>PCB layout</td>
<td>23&lt;sup&gt;rd&lt;/sup&gt; February</td>
<td>27&lt;sup&gt;th&lt;/sup&gt; February</td>
</tr>
<tr>
<td>Assembled PCB</td>
<td>30&lt;sup&gt;th&lt;/sup&gt; March</td>
<td>19&lt;sup&gt;th&lt;/sup&gt; April</td>
</tr>
<tr>
<td>MSP430 toolchain guide and software</td>
<td>23&lt;sup&gt;rd&lt;/sup&gt; March</td>
<td>10&lt;sup&gt;th&lt;/sup&gt; May</td>
</tr>
<tr>
<td>Raspberry Pi toolchain guide and software</td>
<td>23&lt;sup&gt;rd&lt;/sup&gt; March</td>
<td>10&lt;sup&gt;th&lt;/sup&gt; May</td>
</tr>
<tr>
<td>Test results</td>
<td>13&lt;sup&gt;th&lt;/sup&gt; April</td>
<td>17&lt;sup&gt;th&lt;/sup&gt; May</td>
</tr>
<tr>
<td>Final report</td>
<td>26&lt;sup&gt;th&lt;/sup&gt; May</td>
<td>26&lt;sup&gt;th&lt;/sup&gt; May</td>
</tr>
</tbody>
</table>

*At the time of writing the interim report, when the expected delivery dates were defined, the author believed the final report due date to be 12<sup>th</sup> May. Hence there is a larger than expected disparity between the expected delivery dates and the actual delivery dates.*
**Appendix 2. Pinout definitions**

Only pins with a specific peripheral or signal that will require assigning for use in the OBC system are shown.

**Raspberry Pi compute modules**

<table>
<thead>
<tr>
<th>Pin#</th>
<th>Pin name</th>
<th>Peripheral use</th>
<th>Signal name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>GPIO-0</td>
<td>SDA0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GPIO-1</td>
<td>SCL0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>GPIO-2</td>
<td>SDA1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>GPIO-3</td>
<td>SCL1</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>GPIO-7</td>
<td>SPI0_CE1_N</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>GPIO-8</td>
<td>SPI0_CE0_N</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>GPIO-28</td>
<td></td>
<td>PWM_CTS</td>
</tr>
<tr>
<td>29</td>
<td>GPIO-9</td>
<td>SPI0_MISO</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>GPIO-29</td>
<td></td>
<td>PWM_RTS</td>
</tr>
<tr>
<td>33</td>
<td>GPIO-10</td>
<td>SPI0_MOSI</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>GPIO-30</td>
<td>CTS1</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>GPIO-11</td>
<td>SPI0_SCLK</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>GPIO-31</td>
<td>RTS1</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>GPIO-12</td>
<td>PWM0</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>GPIO-32</td>
<td>TXD1</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>GPIO-13</td>
<td>PWM1</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>GPIO-33</td>
<td>RXD1</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>GPIO-14</td>
<td></td>
<td>I2C_0_CTS</td>
</tr>
<tr>
<td>53</td>
<td>GPIO-15</td>
<td></td>
<td>I2C_0_RTS</td>
</tr>
<tr>
<td>58</td>
<td>GPIO-36</td>
<td>TXD0</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>GPIO-37</td>
<td>RXD0</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>GPIO-18</td>
<td>BSCSL_MOSI</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>GPIO-38</td>
<td>RTS0</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>GPIO-19</td>
<td>BSCSL_SCLK</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>GPIO-39</td>
<td>CTS0</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>GPIO-20</td>
<td>BSCSL_MISO</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>GPIO-40</td>
<td>SPI2_MISO</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>GPIO-21</td>
<td>BSCSL_CE_N</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>GPIO-41</td>
<td>SPI2_MOSI</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>GPIO-22</td>
<td></td>
<td>SPI0_CTS</td>
</tr>
<tr>
<td>76</td>
<td>GPIO-42</td>
<td>SPI2_SCLK</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>GPIO-23</td>
<td></td>
<td>SPI0_RTS</td>
</tr>
<tr>
<td>78</td>
<td>GPIO-43</td>
<td>SPI2_CE0_N</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>GPIO-24</td>
<td></td>
<td>SPI1_CTS</td>
</tr>
<tr>
<td>82</td>
<td>GPIO-44</td>
<td>SPI2_CE1_N</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>GPIO-25</td>
<td></td>
<td>SPI1_RTS</td>
</tr>
<tr>
<td>84</td>
<td>GPIO-45</td>
<td>SPI2_CE2_N</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>GPIO-26</td>
<td></td>
<td>I2C1_CTS</td>
</tr>
<tr>
<td>88</td>
<td>GPIO_46_1V8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>GPIO-27</td>
<td></td>
<td>I2C1_RTS</td>
</tr>
</tbody>
</table>
### MSP 430FR6989 pinout

<table>
<thead>
<tr>
<th>Pin number</th>
<th>Pin name</th>
<th>Peripheral use</th>
<th>Signal name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P4.3</td>
<td></td>
<td>UART_DEBUG</td>
</tr>
<tr>
<td>4</td>
<td>P1.6</td>
<td></td>
<td>UCB0SDA</td>
</tr>
<tr>
<td>5</td>
<td>P1.7</td>
<td></td>
<td>USB0SCL</td>
</tr>
<tr>
<td>7</td>
<td>P6.0</td>
<td></td>
<td>RPI1_I2C2_CTS</td>
</tr>
<tr>
<td>8</td>
<td>P6.1</td>
<td></td>
<td>RPI2_I2C_CTS</td>
</tr>
<tr>
<td>11</td>
<td>P6.4</td>
<td></td>
<td>TB0.0</td>
</tr>
<tr>
<td>13</td>
<td>P6.6</td>
<td></td>
<td>TB0.2</td>
</tr>
<tr>
<td>15</td>
<td>P2.5</td>
<td></td>
<td>TB0.4</td>
</tr>
<tr>
<td>23</td>
<td>P3.0</td>
<td></td>
<td>RPI1_UART_RTS</td>
</tr>
<tr>
<td>24</td>
<td>P3.1</td>
<td></td>
<td>RPI2_SPI_RTS</td>
</tr>
<tr>
<td>25</td>
<td>P3.2</td>
<td></td>
<td>RPI1_CAN_RTS</td>
</tr>
<tr>
<td>28</td>
<td>TEST</td>
<td></td>
<td>SBWTCK</td>
</tr>
<tr>
<td>29</td>
<td>nRST</td>
<td></td>
<td>SBWTDIO</td>
</tr>
<tr>
<td>30</td>
<td>PJ.0</td>
<td></td>
<td>TDO</td>
</tr>
<tr>
<td>31</td>
<td>PJ.1</td>
<td></td>
<td>TDI</td>
</tr>
<tr>
<td>32</td>
<td>PJ.2</td>
<td></td>
<td>TMS</td>
</tr>
<tr>
<td>33</td>
<td>PJ.3</td>
<td></td>
<td>TCK</td>
</tr>
<tr>
<td>35</td>
<td>P7.5</td>
<td></td>
<td>RPI1_SPI_CTS</td>
</tr>
<tr>
<td>36</td>
<td>P7.6</td>
<td></td>
<td>RPI1_CAN_CTS</td>
</tr>
<tr>
<td>38</td>
<td>P7.7</td>
<td></td>
<td>RPI1_I2C_CTS</td>
</tr>
<tr>
<td>39</td>
<td>P3.3</td>
<td></td>
<td>RPI1_I2C1_RTS</td>
</tr>
<tr>
<td>40</td>
<td>P3.4</td>
<td></td>
<td>RPI1_I2C2_RTS</td>
</tr>
<tr>
<td>41</td>
<td>P3.5</td>
<td></td>
<td>RPI2_UART_RTS</td>
</tr>
<tr>
<td>42</td>
<td>P3.6</td>
<td></td>
<td>RPI2_SPI_RTS</td>
</tr>
<tr>
<td>43</td>
<td>P3.7</td>
<td></td>
<td>RPI2_CAN_RTS</td>
</tr>
<tr>
<td>44</td>
<td>P8.0</td>
<td></td>
<td>RPI2_UART_CTS</td>
</tr>
<tr>
<td>45</td>
<td>P8.1</td>
<td></td>
<td>RPI2_SPI_CTS</td>
</tr>
<tr>
<td>46</td>
<td>P8.2</td>
<td></td>
<td>RPI2_CAN_CTS</td>
</tr>
<tr>
<td>47</td>
<td>P8.3</td>
<td></td>
<td>RPI2_I2C_CTS</td>
</tr>
<tr>
<td>50</td>
<td>P2.1</td>
<td></td>
<td>UCA0RXD</td>
</tr>
<tr>
<td>51</td>
<td>P2.0</td>
<td></td>
<td>UCA0TXD</td>
</tr>
<tr>
<td>52</td>
<td>P7.0</td>
<td></td>
<td>RPI1_Power_on</td>
</tr>
<tr>
<td>53</td>
<td>P7.1</td>
<td></td>
<td>TA0.0</td>
</tr>
<tr>
<td>54</td>
<td>P7.2</td>
<td></td>
<td>TA0.1</td>
</tr>
<tr>
<td>55</td>
<td>P7.3</td>
<td></td>
<td>TA0.2</td>
</tr>
<tr>
<td>56</td>
<td>P7.4</td>
<td></td>
<td>RPI1_UART_CTS</td>
</tr>
<tr>
<td>62</td>
<td>P8.7</td>
<td></td>
<td>A4</td>
</tr>
<tr>
<td>63</td>
<td>P1.3</td>
<td></td>
<td>A3</td>
</tr>
<tr>
<td>64</td>
<td>P1.2</td>
<td></td>
<td>A2</td>
</tr>
<tr>
<td>67</td>
<td>P9.0</td>
<td></td>
<td>A8</td>
</tr>
<tr>
<td>68</td>
<td>P9.1</td>
<td></td>
<td>A9</td>
</tr>
<tr>
<td>69</td>
<td>P9.2</td>
<td></td>
<td>A10</td>
</tr>
<tr>
<td>70</td>
<td>P9.3</td>
<td></td>
<td>A11</td>
</tr>
<tr>
<td>71</td>
<td>P9.4</td>
<td></td>
<td>A12</td>
</tr>
<tr>
<td>72</td>
<td>P9.5</td>
<td></td>
<td>A13</td>
</tr>
<tr>
<td>73</td>
<td>P9.6</td>
<td></td>
<td>A14</td>
</tr>
<tr>
<td>74</td>
<td>P9.7</td>
<td></td>
<td>A15</td>
</tr>
<tr>
<td>81</td>
<td>PJ.7</td>
<td></td>
<td>HFXOUT</td>
</tr>
<tr>
<td>82</td>
<td>PJ.6</td>
<td></td>
<td>HFXIN</td>
</tr>
<tr>
<td>84</td>
<td>PJ.4</td>
<td></td>
<td>LFXIN</td>
</tr>
<tr>
<td>85</td>
<td>PJ.5</td>
<td></td>
<td>LFXOUT</td>
</tr>
<tr>
<td>87</td>
<td>P5.4</td>
<td></td>
<td>UCA1SIMO</td>
</tr>
<tr>
<td>88</td>
<td>P5.5</td>
<td></td>
<td>UCA1SOMI</td>
</tr>
<tr>
<td>89</td>
<td>P5.6</td>
<td></td>
<td>UCA1CLK</td>
</tr>
<tr>
<td>90</td>
<td>P5.7</td>
<td></td>
<td>UCA1STE</td>
</tr>
<tr>
<td>Pin number</td>
<td>Pin use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.1</td>
<td>CAN (low)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.2</td>
<td>GPIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.3</td>
<td>CAN (high)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.9</td>
<td>SPI CS1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.10</td>
<td>SPI CS2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.11</td>
<td>SPI CS3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.12</td>
<td>SPI CS4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.13</td>
<td>SPI CS5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.14</td>
<td>SPI CS6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.18</td>
<td>GPIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.19</td>
<td>UART2 RX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.20</td>
<td>UART2 TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.21</td>
<td>SPI CLK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.22</td>
<td>SPI MOSI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.23</td>
<td>SPI MISO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.24</td>
<td>SPI CS7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.26</td>
<td>GPIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.29</td>
<td>GPIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.32</td>
<td>+5V_USB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.33</td>
<td>UART1_RX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.35</td>
<td>UART1_TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.41</td>
<td>I2C SDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.43</td>
<td>I2C SCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.45</td>
<td>I2C SDA2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.47</td>
<td>I2C SCL2/5V in opt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.48</td>
<td>3V3 in opt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.49</td>
<td>5V in opt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.50</td>
<td>3V3 in opt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.51</td>
<td>5V in opt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H1.52</td>
<td>3V3 in opt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.25</td>
<td>+5V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.26</td>
<td>+5V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.27</td>
<td>+3.3V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.28</td>
<td>+3.3V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.29</td>
<td>GND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.30</td>
<td>GND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.31</td>
<td>AGND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.32</td>
<td>GND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.45</td>
<td>VBATT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.46</td>
<td>VBATT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H2.50</td>
<td>GPIO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CubeSat kit bus header pinout**
Appendix 3. Complete circuit schematic

rpicomputeOBC.sch
boardIO.sch

Debug.sch
Power.sch
Antilatchup.sch

When an incorrect state is detected by the current sense amplifier and comparator the output is turned off for about 200ms by the monostable constructed from N0P07155.

Antilatchup_switch.sch

When there is a rising edge at FA the 6-channel MOSFET turns on, which then turns on the 4-channel MOSFET which then maintains the 6-channel in on state.

If the external pulse high, the 4-channel is switched off, turning off the 6-channel as well. This can then be reset by another rising edge at the 5th input. A falling edge at the 5th input will turn off the circuit.

Rpi_anti_latchup_switch.sch

When an incorrect state is detected by the comparator the output is turned off. This fault state inhibiting the output is maintained by the use of a 50-turn reset from discrete MOSFETs. To reset the output, the 5th signal needs resetting.
Bus_peripherals.sch

256K Fujitsu FRAM. WP, A0, A1, A2 are all internally pulled down.

busMuxes.sch

UARTmux.sch
RPI.sch
Appendix 4. PCB Layout designs

Top copper layer, signals and components

Top silkscreen
Middle copper layer, power planes

Middle copper layer, Ground plane
Bottom copper layer, signals, and components

Bottom silkscreen
Appendix 5. First PCB design with incorrect SODIMM footprint
Appendix 6. Setting up the MSP430 toolchain

The toolchain was set up in an Arch Linux environment. These instructions should easily be adaptable to another Linux distribution.

After the “mspgcc-ti” package has been installed, the toolchain is installed in the /opt/ti/mspgcc/bin directory. This directory needs to be added to the PATH variable.

The open source integrated development environment (IDE) Eclipse was used to develop the MSP430 C code. It needed to be set up to use the mspgcc toolchain.

- First install the Eclipse IDE with the c/c++ development environment
- Then create a new C project. Select “Cross GCC” as the compiler
- If requested for the compiler prefix enter msp430-elf-

- Once the project has been made, right click on the project name in the project explorer plane and click “preferences”
- Select “settings” on the left, then “Cross Settings” confirm the settings are as shown:

![Eclipse Cross Settings](image)

- Next select Cross GCC Compiler and change the command setting to read:
  ```
gcc -mmcu=msp430fr6989
  ```
- Add this directory to the include paths under the “includes” menu:
  ```
  /opt/ti/mspgcc/include
  ```
- Next select the Cross GCC Linker and change the command to read:
  \texttt{gcc –mcpu=msp430fr6989}

- Add this directory to the library search paths under the “libraries” menu:
  \texttt{/opt/ti/mspgcc/lib}

- Next select the Cross GCC Assembler and ensure the commands reads:
  \texttt{as}

- Add this directory to the library search paths under the “General” menu:
  \texttt{/opt/ti/mspgcc/include}

- Click ok.

The compilation toolchain should now be set up such that the project in eclipse can be compiled. If the C code already developed for the OBC has been imported, it may be necessary to add the “headers” and “MSP430FR5xx_6xx” folders to the include paths under the Cross GCC Compiler and the Cross GCC Assembler.

To program the MSP430FR6989 on the OBC a FRAM experimenter board is used as the programmer. Any MSP430 Launchpad should by suitable for this purpose. The five headers towards the top of the Launchpad board need to be removed. Wires then need connecting from the “Test” and “Rst” lines nearest the USB socket on the Launchpad to the programming interface on the OBC. Do not forget to connect the grounds as well!

The software used to program the MSP430 on the OBC is “MSP-debug”, this is also available in the Arch User Repository (AUR). The command to enter the debug interface onto the target MSP430 is:

\texttt{mspdebug rf2500 --force-fet-id MSP430FR5969}

Next to actually load the compiled program enter:

\texttt{load <compiled_program_file_name>}

Finally to run the compiled program on the target device enter:

\texttt{run}
Appendix 7.  C code for the MSP430

The C code for the MSP430 is split across several files. The MSP430 driver libraries released by TI are used in this code, included with the header driverlib.h, it is kept in a separate folder and its location is included at compile time. Similarly, the msp430fr6989.h header is included at compile time and is supplied with the compiler.

Main.c
#include <msp430fr6989.h>
#include <init.h>
#include <system_ctl.h>

int main() {
    WDTCTL = WDTPW | WDTHOLD; // Stop WDT
    __disable_interrupt();

    clock_init();
    interrupt_init();
    timer_init();
    gpio_init();
    uart_init();
    __enable_interrupt();

    system_ctl(); // Infinite loop, controls the system once setup

    return(1);
}
Init.c

#include <driverlib.h>

#include <gpio_defs.h>
#include <init.h>
#include <config.h>

void gpio_init(void) {
    // Setup pins needed as outputs

    // Port 1:
    P1DIR |= 0;
    // Port 2:
    P2DIR |= 0;
    // Port 3:
    P3DIR |= 0;
    // Port 4:
    P4DIR |= 0;
    // Port 5:
    P5DIR |= 0;
    // Port 6: RPI1_RUN, RPI2_RUN, RPI_USB_BOOT_EN,
    P6DIR |= BIT0 | BIT1 | BIT2;
    // Port 7: I2C_bus_PWR_EN, UART_mux_PWR_EN, RPI1_PWR_EN, RPI2_PWR_EN, RPI1_UART_CTS,
    P7DIR |= BIT0 | BIT1 | BIT2 | BIT3 | BIT4 | BIT7;
    // Port 8:
    P8DIR |= 0;
    // Port 9:
    P9DIR |= 0;
    // Port 10: LED1, LED2, LED3,
    P10DIR |= BIT0 | BIT1 | BIT2;
    // Port 11:
    PJDIR |= 0;

    // Setup output pin states on boot
    RPI1_RUN_CLR;
    RPI2_RUN_CLR;
    RPI1_PWR_EN_SET;
    RPI2_PWR_EN_SET;
    RPI_USB_BOOT_EN_CLR;
    I2C_BUS_PWR_EN_CLR;
}
UART_MUX_PWR_EN_CLR;

LED1_CLR;
LED2_CLR;
LED3_CLR;

RPI1_UART_CTS_CLR;
RPI2_UART_CTS_CLR;

// Configure UART1 pins (debug)
// Set P5.4 and P5.5 as Primary Module Function Input.
/*
* Select Port 5
* Set Pin 4, 5 to input Primary Module Function, (UCA0TXD/UCA0SIMO, UCA0RXD/UCA0SOMI).
*/
GPIO_setAsPeripheralModuleFunctionInputPin(
    GPIO_PORT_P5,
    GPIO_PIN4 + GPIO_PIN5,
    GPIO_PRIMARY_MODULE_FUNCTION
);

// Configure UART0 pins (rpi)
// Set P2.0 and P2.1 as Primary Module Function Input.
/*
* Select Port 2
* Set Pin 0, 1 to input Primary Module Function, (UCA0TXD/UCA0SIMO, UCA0RXD/UCA0SOMI).
*/
GPIO_setAsPeripheralModuleFunctionInputPin(
    GPIO_PORT_P2,
    GPIO_PIN0 + GPIO_PIN1,
    GPIO_PRIMARY_MODULE_FUNCTION
);

// Disable the GPIO power-on default high-impedance mode to activate previously configured port settings
PM5CTL0 &= ~LOCKLPM5;
}

void gpio_deinit(void) {
    // Make sure all output pins are in a safe state
    RPI1_RUN_CLR;
    RPI2_RUN_CLR;
    RPI1_PWR_EN_SET;
    RPI2_PWR_EN_CLR;
    RPI_USB_BOOT_EN_CLR;
}
I2C_BUS_PWR_EN_CLR;
UART_MUX_PWR_EN_CLR;

RPI1_UART_CTS_CLR;
RPI2_UART_CTS_CLR;

LED1_CLR;
LED2_CLR;
LED3_CLR;
}

void clock_init(void) {

//Set DCO Frequency to 8MHz
CS_setDCOFreq(CS_DCORSEL_0,CS_DCOFSEL_6);

//configure MCLK, SMCLK to be source by DCOCLK
CS_initClockSignal(CS_SMCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_2); // 4 Mhz
CS_initClockSignal(CS_MCLK,CS_DCOCLK_SELECT,CS_CLOCK_DIVIDER_1); // 8 Mhz
}

void interrupt_init(void) {

// Port 1 interrupt pins

//
// P1IE    |=     BIT2    |    BIT3;   // Enable interrupts on these pins
// P1IES   &=     ~BIT2   &    ~BIT3; // Rising edge pins
// P1IES   |=  0;            // Falling edge pins

// Setup P1.2 as an interrupt pin (WDT timer reset)
GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN2);
GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN2);
GPIO_selectInterruptEdge(GPIO_PORT_P1, GPIO_PIN2, GPIO_LOW_TO_HIGH_TRANSITION);
GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN2);

// Setup P1.3 as an interrupt pin (WDT timer reset)
GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN3);
GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN3);
GPIO_selectInterruptEdge(GPIO_PORT_P1, GPIO_PIN3, GPIO_LOW_TO_HIGH_TRANSITION);
GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN3);

// Setup P4.0 as an interrupt pin (I2C power monitor)
GPIO_setAsInputPin(GPIO_PORT_P4, GPIO_PIN0);
GPIO_enableInterrupt(GPIO_PORT_P4, GPIO_PIN0);
GPIO_selectInterruptEdge(GPIO_PORT_P4, GPIO_PIN0, GPIO_HIGH_TO_LOW_TRANSITION);
void timer_init(void) {

    // Setup the Timer A to cause in interrupt every now and then
    // Time in ms between interrupts defined by TIMER_ACOMPARE_VALUE

    // Start timer in continuous mode sourced by SMCLK
    Timer_A_initContinuousModeParam initContParam = {0};
    initContParam.clockSource = TIMER_A_CLOCKSOURCE_SMCLK;
    initContParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_64;  // 62.5 kHz
    initContParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_DISABLE;
    initContParam.timerClear = TIMER_A_DO_CLEAR;
    initContParam.startTimer = false;
    Timer_A_initContinuousMode(TIMER_A1_BASE, &initContParam);

    // Initialize compare mode
    Timer_A_clearCaptureCompareInterrupt(TIMER_A1_BASE,
    TIMER_A_CAPTURECOMPARE_REGISTER_0);

    Timer_A_initCompareModeParam initCompParam = {0};
    initCompParam.compareRegister = TIMER_A_CAPTURECOMPARE_REGISTER_0;
    initCompParam.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE;
    initCompParam.compareOutputMode = TIMER_A_OUTPUTMODE_OUTBITVALUE;
    initCompParam.compareValue = TIMER_A_COMPARE_VALUE;
    Timer_A_initCompareMode(TIMER_A1_BASE, &initCompParam);

    Timer_A_startCounter(TIMER_A1_BASE, TIMER_A_CONTINUOUS_MODE);
}

void uart_init(void) {

    // Set P4.2 as an interrupt pin (RPI 1 power monitor)
    GPIO_setAsInputPin(GPIO_PORT_P4, GPIO_PIN2);
    GPIO_enableInterrupt(GPIO_PORT_P4, GPIO_PIN2);
    GPIO_selectInterruptEdge(GPIO_PORT_P4, GPIO_PIN2, GPIO_HIGH_TO_LOW_TRANSITION);
    GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN2);

    // Set P4.3 as an interrupt pin (RPI 2 power monitor)
    GPIO_setAsInputPin(GPIO_PORT_P4, GPIO_PIN3);
    GPIO_enableInterrupt(GPIO_PORT_P4, GPIO_PIN3);
    GPIO_selectInterruptEdge(GPIO_PORT_P4, GPIO_PIN3, GPIO_HIGH_TO_LOW_TRANSITION);
    GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN3);
}

GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN0);

// Set P4.2 as an interrupt pin (RPI 1 power monitor)
GPIO_setAsInputPin(GPIO_PORT_P4, GPIO_PIN2);
GPIO_enableInterrupt(GPIO_PORT_P4, GPIO_PIN2);
GPIO_selectInterruptEdge(GPIO_PORT_P4, GPIO_PIN2, GPIO_HIGH_TO_LOW_TRANSITION);
GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN2);

// Set P4.3 as an interrupt pin (RPI 2 power monitor)
GPIO_setAsInputPin(GPIO_PORT_P4, GPIO_PIN3);
GPIO_enableInterrupt(GPIO_PORT_P4, GPIO_PIN3);
GPIO_selectInterruptEdge(GPIO_PORT_P4, GPIO_PIN3, GPIO_HIGH_TO_LOW_TRANSITION);
GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN3);

// Start timer in continuous mode sourced by SMCLK
Timer_A_initContinuousModeParam initContParam = {0};
    initContParam.clockSource = TIMER_A_CLOCKSOURCE_SMCLK;
    initContParam.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_64;  // 62.5 kHz
    initContParam.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_DISABLE;
    initContParam.timerClear = TIMER_A_DO_CLEAR;
    initContParam.startTimer = false;
    Timer_A_initContinuousMode(TIMER_A1_BASE, &initContParam);

    // Initialize compare mode
    Timer_A_clearCaptureCompareInterrupt(TIMER_A1_BASE,
    TIMER_A_CAPTURECOMPARE_REGISTER_0);

    Timer_A_initCompareModeParam initCompParam = {0};
    initCompParam.compareRegister = TIMER_A_CAPTURECOMPARE_REGISTER_0;
    initCompParam.compareInterruptEnable = TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE;
    initCompParam.compareOutputMode = TIMER_A_OUTPUTMODE_OUTBITVALUE;
    initCompParam.compareValue = TIMER_A_COMPARE_VALUE;
    Timer_A_initCompareMode(TIMER_A1_BASE, &initCompParam);

    Timer_A_startCounter(TIMER_A1_BASE, TIMER_A_CONTINUOUS_MODE);
}
// Configure UARTs

// Settings are as prescribed here:

// Debug uart (A1) pins 5.4 and 5.5
EUSCI_A_UART_initParam param = {0};
param.selectClockSource = EUSCI_A_UART_CLOCKSOURCE_SMCLK;
param.clockPrescalar = 2;
param.firstModReg = 2;
param.secondModReg = 187;
param.parity = EUSCI_A_UART_NO_PARITY;
param.msborLsbFirst = EUSCI_A_UART_LSB_FIRST;
param.numberofStopBits = EUSCI_A_UART_ONE_STOP_BIT;
param.uartMode = EUSCI_A_UART_MODE;
param.overSampling = EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION;
if(STATUS_FAIL == EUSCI_A_UART_init(EUSCI_A1_BASE, &param))
{
    return;
}

EUSCI_A_UART_enable(EUSCI_A1_BASE);
EUSCI_A_UART_clearInterrupt(EUSCI_A1_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG);
EUSCI_A_UART_clearInterrupt(EUSCI_A1_BASE, EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG);

// Debug uart (A0) pins 2.0 and 2.1
EUSCI_A_UART_initParam param2 = {0};
param.selectClockSource = EUSCI_A_UART_CLOCKSOURCE_SMCLK;
param.clockPrescalar = 2;
param.firstModReg = 2;
param.secondModReg = 187;
param.parity = EUSCI_A_UART_NO_PARITY;
param.msborLsbFirst = EUSCI_A_UART_LSB_FIRST;
param.numberofStopBits = EUSCI_A_UART_ONE_STOP_BIT;
param.uartMode = EUSCI_A_UART_MODE;
param.overSampling = EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION;
if(STATUS_FAIL == EUSCI_A_UART_init(EUSCI_A0_BASE, &param2))
{ 
    return;
}

EUSCI_A_UART_enable(EUSCI_A0_BASE);
EUSCI_A_UART_clearInterrupt(EUSCI_A0_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG);
EUSCI_A_UART_clearInterrupt(EUSCI_A0_BASE, EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG);

// Enable USCI_A0 RX interrupt
EUSCI_A_UART_enableInterrupt(EUSCI_A0_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT); // Enable interrupt
EUSCI_A_UART_enableInterrupt(EUSCI_A0_BASE, EUSCI_A_UART_TRANSMIT_INTERRUPT);
}

Init.h
#define HEADERS_INIT_H_

void gpio_init(void);
void gpio_deinit(void);
void clock_init(void);
void interrupt_init(void);
void timer_init(void);
void uart_init(void);

#endif /* HEADERS_INIT_H_ */

System_ctl.c
#include <system_ctl.h>
#include <gpio_defs.h>
#include <debug_uart_cli.h>
#include <config.h>

// Global state variables, can be edited elsewhere to change the system state. These are the default boot up states for the system
power_state rpi_1_pwr = on;
power_state rpi_2_pwr = on;
power_state uart_mux_pwr = off;
power_state I2C_bus_pwr = off;

rpi_run_state rpi_1_run = stop;
rpi_run_state rpi_2_run = stop;
power_state RPI1_uart_fwd = off;
print_debug_menu_state debug_menu_to_print = print_true;

uint16_t rpi1_wd_timer = 0;
uint16_t rpi2_wd_timer = 0;

void system_ctl(void) {

    // Local previous state variables, can't be edited elsewhere. We just check for a change, then do!
    power_state prev_rpi_1_pwr = off;
    power_state prev_rpi_2_pwr = off;
    // power_state prev_uart_mux_pwr = off;
    power_state prev_I2C_bus_pwr = off;

    rpi_run_state prev_rpi_1_run = stop;
    rpi_run_state prev_rpi_2_run = stop;

    print_debug_menu_state prev_debug_menu_to_print = print_false;

    while(1) {
        // If requested, change the power state of RPI1
        if (prev_rpi_1_pwr != rpi_1_pwr) {
            if (rpi_1_pwr == on) {
                RPI1_PWR_EN_SET;
                uart1_write("RPI 1 on\n");
            } else if (rpi_1_pwr == tripped) {
                uart1_write("RPI 1 power has tripped\n");
                rpi_1_run = off;
            } else {
                RPI1_PWR_EN_CLR;
                rpi_1_run = off;
            }
            prev_rpi_1_pwr = rpi_1_pwr;
        }

        // If requested, change the power state of RPI2
        if (prev_rpi_2_pwr != rpi_2_pwr) {
            if (rpi_2_pwr == on) {
                RPI2_PWR_EN_SET;
                uart1_write("RPI 2 on\n");
            } else if (rpi_2_pwr == tripped) {
                uart1_write("RPI 2 power has tripped\n");
                rpi_2_run = off;
            } else {
                RPI2_PWR_EN_CLR;
                rpi_2_run = off;
            }
        }
    }
}
uart1_write("RPI 2 off\n");
}
prev_rpi_2_pwr = rpi_2_pwr;
}

if (prev_rpi_1_run != rpi_1_run) {
    if (rpi_1_run == boot) {
        RPI1_RUN_SET;
        uart1_write("RPI 1 booting\n");
    } else if (rpi_2_run == run) {
        uart1_write("RPI 1 booted\n");
    } else if (rpi_1_run == WDexpired) {
        uart1_write("RPI 1 WD expired, turning off\n");
        RPI1_RUN_CLR;
    } else if (rpi_1_run == stop) {
        RPI1_RUN_CLR;
        uart1_write("RPI 1 stopped\n");
    }
    prev_rpi_1_run = rpi_1_run;
}

if (prev_rpi_2_run != rpi_2_run) {
    if (rpi_2_run == boot) {
        RPI2_RUN_SET;
        uart1_write("RPI 2 booting\n");
    } else if (rpi_2_run == run) {
        uart1_write("RPI 2 booted\n");
    } else if (rpi_2_run == WDexpired) {
        uart1_write("RPI 2 WD expired, turning off\n");
        RPI2_RUN_CLR;
    } else if (rpi_2_run == stop) {
        RPI2_RUN_CLR;
        uart1_write("RPI 2 stopped\n");
    }
    prev_rpi_2_run = rpi_2_run;
}

// If requested, change the power state of the UART muxes
/*if (prev_uart_mux_pwr != uart_mux_pwr) {
    if (uart_mux_pwr == on) {
        UART_MUX_PWR_EN_SET;
        uart1_write("uart mux on\n");
    } else if (uart_mux_pwr == tripped) {
        uart1_write("uart mux power has tripped\n");
    } else {
UART_MUX_PWR_EN_CLR;
uart1_write("uart mux off\n");
}
prev_uart_mux_pwr = uart_mux_pwr;
*/

// If requested, change the power state of the I2C bus
if (prev_I2C_bus_pwr != I2C_bus_pwr) {
  if (I2C_bus_pwr == on) {
    I2C_BUS_PWR_EN_SET;
    uart1_write("I2C bus on\n");
  } else if (I2C_bus_pwr == tripped) {
    uart1_write("I2C bus power has tripped\n");
  } else {
    I2C_BUS_PWR_EN_CLR;
    uart1_write("I2C bus off\n");
  }
  prev_I2C_bus_pwr = I2C_bus_pwr;
}

// If requested print out the debug menu
if (prev_debug_menu_to_print != debug_menu_to_print) {
  if (debug_menu_to_print == print_true) {
    print_debug_menu();
  }
  prev_debug_menu_to_print = print_false;
  debug_menu_to_print = print_false;
}

// Raspberry Pi watchdog timer monitoring
if (rpi_1_run == boot) {
  if (rpi1_wd_timer > RPI_BOOT_TIMEOUT_VALUE) {
    rpi_1_run = WDexpired;
  }
} else if (rpi_1_run == run) {
  if (rpi1_wd_timer > RPI_WD_TIMEOUT_VALUE) {
    rpi_1_run = WDexpired;
  }
}

if (rpi_1_run == boot) {
  if (rpi2_wd_timer > RPI_BOOT_TIMEOUT_VALUE) {
    rpi_2_run = WDexpired;
  }
} else if (rpi_2_run == run) {
if (rpi2_wd_timer == RPI_WD_TIMEOUT_VALUE) {
    rpi_2_run = WExpired;
}
}
System_ctl.h

#ifndef HEADERS_SYSTEM_CTL_H_
#define HEADERS_SYSTEM_CTL_H_

#include <msp430fr6989.h>
#include <driverlib.h>

typedef enum {
    on,
    tripped,
    off
} power_state;

typedef enum {
    boot,
    run,
    WDexpired,
    stop
} rpi_run_state;

typedef enum {
    print_true,
    print_false
} print_debug_menu_state;

extern uint16_t rpi1_wd_timer;
extern uint16_t rpi2_wd_timer;
power_state rpi_1_pwr;
power_state rpi_2_pwr;
power_state uart_mux_pwr;
power_state I2C_bus_pwr;

rpi_run_state rpi_1_run;
rpi_run_state rpi_2_run;
power_state RPI1_uart_fwd;

print_debug_menu_state debug_menu_to_print;

void system_ctl(void);

#endif /* HEADERS_SYSTEM_CTL_H_ */

ISR.s.c
```c
#include <ISRs.h>
#include <config.h>
#include <uart_io.h>
#include <debug_uart_cli.h>
#include <system_ctl.h>

//****************************************************************************
//This is the PORT1_VECTOR interrupt vector service routine
//****************************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT1_VECTOR
__interrupt
#else defined(__GNUC__)
__attribute__((interrupt(PORT1_VECTOR)))
#endif
void Port_1(void)
{
    if (GPIO_getInterruptStatus(GPIO_PORT_P1, GPIO_PIN2)) {
        if (LED2_VAL) {
            LED2_CLR;
        } else {
            LED2_SET;
        }
        if (rpi_1_run == boot) {
            rpi_1_run = run;
        }
        rpi1_wd_timer = 0;
        GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN2);
    }

    if (GPIO_getInterruptStatus(GPIO_PORT_P1, GPIO_PIN3)) {
        if (LED3_VAL) {
            LED3_CLR;
        } else {
            LED3_SET;
        }
        if (rpi_2_run == boot) {
            rpi_2_run = run;
        }
        rpi2_wd_timer = 0;
        GPIO_clearInterrupt(GPIO_PORT_P1, GPIO_PIN3);
    }
}
```
/**************************************************************************
//
//This is the PORT4 Vector interrupt vector service routine
//
/***************************************************************************/
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=PORT4_VECTOR
__interrupt
#elif defined(__GNUC__)
__attribute__((interrupt(PORT4_VECTOR)))
#endif
void Port_4(void)
{
    if (GPIO_getInterruptStatus(GPIO_PORT_P4, GPIO_PIN0)) // I2C_BUS_PWR_MON
        if (I2C_BUS_PWR_MON_VAL == 0 && I2C_bus_pwr == on) {
            I2C_bus_pwr = tripped;
            uart_mux_pwr = tripped;
        }
        GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN0);
    
    if (GPIO_getInterruptStatus(GPIO_PORT_P4, GPIO_PIN2)) // RPI1_PWR_MON
        if (RPI1_PWR_MON_VAL == 0 && rpi_1_pwr == on) {
            rpi_1_pwr = tripped;
        }
        GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN2);
    
    if (GPIO_getInterruptStatus(GPIO_PORT_P4, GPIO_PIN3)) // RPI2_PWR_MON
        if (RPI2_PWR_MON_VAL == 0 && rpi_2_pwr == on) {
            rpi_2_pwr = tripped;
        }
        GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN3);
}

/**************************************************************************
//
//This is the TIMER1_A3 interrupt vector service routine.
//
/******************************************************************************/
```c
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=TIMER1_A0_VECTOR
__interrupt
#elif defined(__GNUC__)
__attribute__((interrupt(TIMER1_A0_VECTOR)))
#endif

void TIMER1_A0_ISR(void)
{
    uint16_t compVal = Timer_A_getCaptureCompareCount(TIMER_A1_BASE, TIMER_A_CAPTURECOMPARE_REGISTER_0) + TIMER_A_COMPARE_VALUE;

    if (LED1_VAL) {
        LED1_CLR;
    } else {
        LED1_SET;
    }

    // Increment the watchdog timers if the Raspberry Pi's are running
    if ((rpi_1_run == run || rpi_1_run == boot) && rpi_1_pwr == on) {
        rpi1_wd_timer++;
    }

    if ((rpi_2_run == run || rpi_2_run == boot) && rpi_2_pwr == on) {
        rpi2_wd_timer++;
    }

    // Add Offset to CCR0
    Timer_A_setCompareValue(TIMER_A1_BASE, TIMER_A_CAPTURECOMPARE_REGISTER_0, compVal);
}
```

char RxChar = 0;
if (EUSCI_A_UART_getInterruptStatus(EUSCI_A0_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG)) {
    RxChar = EUSCI_A_UART_receiveData(EUSCI_A0_BASE);
    if (RPI1_uart_fwd == on) {
        uart1_write_char(RxChar);
    }
    EUSCI_A_UART_clearInterrupt(EUSCI_A0_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG);
}
if (EUSCI_A_UART_getInterruptStatus(EUSCI_A0_BASE, EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG)) {
    UART0_TX_state = Ready;
    EUSCI_A_UART_clearInterrupt(EUSCI_A0_BASE, EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG);
}

//****************************************************************************
//This is the USCI_A1 interrupt vector service routine.
//****************************************************************************
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCI_A1_VECTOR
__interrupt
#else defined(__GNUC__)
__attribute__((interrupt(USCI_A1_VECTOR)))
#endif
void USCI_A1_ISR(void)
{
    char RxChar = 0;
    if (EUSCI_A_UART_getInterruptStatus(EUSCI_A1_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG)) {
        RxChar = EUSCI_A_UART_receiveData(EUSCI_A1_BASE);
        if (RPI1_uart_fwd == off) {
            // Quickly check if the received character is a recognised command character and if it is, send it to the parser to be used.
        }
    }
}
if ( (RxChar == '1') || (RxChar == '2') || (RxChar == 's') || (RxChar == '?') || (RxChar == '3')
|| (RxChar == 'y') ) {
    command_parser(RxChar);
} else {
    uart0_write_char(RxChar);
}

EUSCI_A_UART_clearInterrupt(EUSCI_A1_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG);
}

if (EUSCI_A_UART_getInterruptStatus(EUSCI_A1_BASE, EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG)) {
    UART1_TX_state = Ready;
    EUSCI_A_UART_clearInterrupt(EUSCI_A1_BASE, EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG);
}

ISR.h

#ifndef HEADERS_ISR_H_
#define HEADERS_ISR_H_

#include <msp430fr6989.h>
#include <driverlib.h>
#include <gpio_defs.h>

void Port_1(void);
void TIMER1_A0_ISR(void);
void USCI_A1_ISR(void);

#endif /* HEADERS_ISR_H_ */
```c
#include <uart_io.h>
#include <string.h>

// Global state variables, used in other files too!
UART_TX_state UART0_TX_state = Ready;
UART_TX_state UART1_TX_state = Ready;

void uart0_write(char *buffer) {
    int i;
    for (i=0; i < strlen(buffer); i++) {
        uart0_write_char(buffer[i]);
    }
}

void uart0_write_char(char character) {
    UART0_TX_state = Transmitting;
    EUSCI_A_UART_transmitData(EUSCI_A0_BASE, character);
    while(UART0_TX_state == Transmitting) {}; // Wait until the character has been sent, state reset by ISR.
}

void uart1_write(char *buffer) {
    int i;
    for (i=0; i < strlen(buffer); i++) {
        uart1_write_char(buffer[i]);
    }
}

void uart1_write_char(char character) {
    UART1_TX_state = Transmitting;
    EUSCI_A_UART_transmitData(EUSCI_A1_BASE, character);
    while(UART1_TX_state == Transmitting) {}; // Wait until the character has been sent, state reset by ISR.
}
```

Uart_io.h

#ifndef HEADERS_UART_IO_H_
#define HEADERS_UART_IO_H_

#include <driverlib.h>

typedef enum {
    Transmitting,
    Ready
} UART_TX_state;

extern UART_TX_state UART0_TX_state;
extern UART_TX_state UART1_TX_state;

void uart0_write(char *buffer);
void uart0_write_char(char character);
void uart1_write(char *buffer);
void uart1_write_char(char character);

#endif /* HEADERS_UART_IO_H_ */

Debug_uart_cli.h

#ifndef HEADERS_DEBUG_UART_CLI_H_
#define HEADERS_DEBUG_UART_CLI_H_

#include <uart_io.h>

void print_debug_menu(void);
void command_parser(char command_char);

#endif /* HEADERS_DEBUG_UART_CLI_H_ */
Debug_uart_cli.c

#include <debug_uart_cli.h>
#include <system_ctl.h>
#include <uart_io.h>
#include <gpio_defs.h>

void print_debug_menu(void) {
    uart1_write("\r\nRaspberry Pi OBC debug console\r\n\r\n1. Turn on/off Raspberry Pi 1\r\n2. Turn on/off Raspberry Pi 2\r\n3. Turn on UART pwr\r\nf. Start uart forwarding mode to RPIs\r\n?. Print out options list again\r\n\r\n//char* test = "zext";
//uart0_write(test);
}

void command_parser(char command_char) {
    switch (command_char) {
    case '1':
        if (rpi_1_run == run || rpi_1_run == boot || rpi_1_run == WDexpired) {
            rpi_1_run = stop;
        } else {
            rpi_1_run = boot;
        }
        break;
    case '2':
        if (rpi_2_run == run || rpi_2_run == boot || rpi_2_run == WDexpired) {
            rpi_2_run = stop;
        } else {
            rpi_2_run = boot;
        }
        break;
    case 's':
        break;
    case '3':
        if (I2C_bus_pwr == on || I2C_bus_pwr == tripped) {
            I2C_bus_pwr = off;
        } else {
            I2C_bus_pwr = on;
        }
        break;
    }
case 'f':
    if (RPI1_uart_fwd == on) {
        RPI1_uart_fwd = off;
        RPI1_UART_CTS_CLR;
    } else {
        UART_MUX_PWR_EN_SET;
        //uart1_write("UART forwarding mode to RPI1 enabled\n\r\n");
        RPI1_uart_fwd = on;
        RPI1_UART_CTS_SET;
    }
    break;

    case '?':
    debug_menu_to_print = print_true;
    break;
    default:
    break;
}

Config.h

#ifndef HEADERS_CONFIG_H_
#define HEADERS_CONFIG_H_

#define timer_val_ms(freq,s)   ((freq/10)*(s/100))
#define TIMER_ACOMPARE_VALUE    timer_val_ms(62500,500)

#define RPI_WD_TIMEOUT_VALUE    60 // 30*2
#define RPI_BOOT_TIMEOUT_VALUE  (RPI_WD_TIMEOUT_VALUE*2)
#endif /* HEADERS_CONFIG_H_ */
```
#include <msp430fr6989.h>

// Pin macros to set the output pins high or low
// Only for pins used!

// Port 6
#define RPI1_RUN_SET     P6OUT |= BIT0
#define RPI1_RUN_CLR     P6OUT &= ~BIT0
#define RPI2_RUN_SET     P6OUT |= BIT1
#define RPI2_RUN_CLR     P6OUT &= ~BIT1
#define RPI_USB_BOOT_EN_SET P6OUT |= BIT2
#define RPI_USB_BOOT_EN_CLR P6OUT &= ~BIT2

// Port 7
#define I2C_BUS_PWR_EN_SET P7OUT |= BIT0
#define UART_MUX_PWR_EN_SET P7OUT |= BIT0
#define UART_MUX_PWR_EN_CLR P7OUT &= ~BIT0
#define RPI1_PWR_EN_SET   P7OUT |= BIT2
#define RPI2_PWR_EN_SET   P7OUT |= BIT3
#define RPI1_UART_CTS_SET P7OUT |= BIT4
#define RPI2_UART_CTS_SET P7OUT |= BIT7
#define RPI1_UART_CTS_CLR P7OUT &= ~BIT4
#define RPI2_UART_CTS_CLR P7OUT &= ~BIT7

// Port 10
#define LED1_SET     P10OUT |= BIT0
#define LED1_CLR     P10OUT &= ~BIT0
#define LED2_SET     P10OUT |= BIT1
#define LED2_CLR     P10OUT &= ~BIT1
#define LED3_SET     P10OUT |= BIT2
#define LED3_CLR     P10OUT &= ~BIT2
```

// Pin macros to get the current pin state (high or low)
// Defined for pins with assigned functions
```c
#define RPI1_WDT_RST_VAL (P1IN & BIT2)
#define RPI2_WDT_RST_VAL (P1IN & BIT3)

#define I2C_BUS_PWR_MON_VAL (P4IN & BIT0)
#define UART_MUX_PWR_MON_VAL I2C_BUS_PWR_MON //P4IN & BIT1
#define RPI1_PWR_MON_VAL (P4IN & BIT2)
#define RPI2_PWR_MON_VAL (P4IN & BIT3)

#define RPI1_RUN_VAL (P6IN & BIT0)
#define RPI2_RUN_VAL (P6IN & BIT1)
#define RPI_USB_BOOT_EN_VAL (P6IN & BIT2)

#define I2C_BUS_PWR_EN_VAL (P7IN & BIT0)
#define UART_MUX_PWR_EN_VAL (P7IN & BIT0)
#define RPI1_PWR_EN_VAL (P7IN & BIT2)
#define RPI2_PWR_EN_VAL (P7IN & BIT3)
#define RPI1_UART_CTS_VAL (P7IN & BIT4)
#define RPI2_UART_CTS_VAL (P7IN & BIT7)

#define LED1_VAL (P10IN & BIT0)
#define LED2_VAL (P10IN & BIT1)
#define LED3_VAL (P10IN & BIT2)
```

```
#define LED1_VAL (P10IN & BIT0)
#define LED2_VAL (P10IN & BIT1)
#define LED3_VAL (P10IN & BIT2)
```

```
#endif /* HEADERS_GPIO_DEFS_H_ */
```
Appendix 8.  Custom Linux device tree file

dt-custom-overlay.dts
/dts-v1/;

/ {
  videocore {
    pins_cm {
      pin_config {
        pin@default {
          polarity = "active_high";
          termination = "pull_down";
          startup_state = "inactive";
          function = "input";
        }; // pin
        pin@p32 { function = "uart0"; termination = "no_pulling"; }; // TX uart0
        pin@p33 { function = "uart0"; termination = "pull_up"; }; // RX uart0
        pin@p36 { function = "uart1"; termination = "no_pulling"; }; // TX uart0
        pin@p37 { function = "uart1"; termination = "pull_up"; }; // RX uart0
        pin@p48 { function = "sdcard"; termination = "pull_up"; drive_strength_mA = < 8 >; }; // SD CLK
        pin@p47 { function = "output"; termination = "pull_up"; polarity = "active_low"; }; // activity LED
        pin@p49 { function = "sdcard"; termination = "pull_up"; drive_strength_mA = < 8 >; }; // SD CMD
        pin@p50 { function = "sdcard"; termination = "pull_up"; drive_strength_mA = < 8 >; }; // SD D0
        pin@p51 { function = "sdcard"; termination = "pull_up"; drive_strength_mA = < 8 >; }; // SD D1
        pin@p52 { function = "sdcard"; termination = "pull_up"; drive_strength_mA = < 8 >; }; // SD D2
        pin@p53 { function = "sdcard"; termination = "pull_up"; drive_strength_mA = < 8 >; }; // SD D3
      }; // pin_config
      pin_defines {
        pin_define@LEDS_DISK_ACTIVITY {
          type = "internal";
          number = <47>;
        }
      }; // pin_defines
    }; // pins_cm
  }
};
Appendix 9. C program to toggle GPIO pin output on Raspberry Pi

Blink.c
/*
 * blink.c:
 * Simple test program to blink an LED on pin 7
 */

#include <wiringPi.h>
#include <stdio.h>

int main (void)
{
    int pin = 4;

    if (wiringPiSetup() == -1)
        exit (1);

    pinMode(pin, OUTPUT);

    for (;;){
        //printf("LED On\n");
        digitalWrite(pin, 1);
        delay(250);
        //printf("LED Off\n");
        digitalWrite(pin, 0);
        delay(250);
    }

    return 0;
}
Appendix 10. Daemon file to start a program upon boot

This file needs to be placed in a certain directory:
/etc/systemd/system

If also needs to have the file extension:
.service

Once placed in the directory, the file needs to be enabled so that it is loaded on boot:

systemctl enable wdt_reset.service

wdt_reset.service
[Unit]
Description=Toggle GPIO 4 to indicate to MSP430 that the PI is still alive

[Service]
Type=简单
ExecStart=/home/rpi/wdt_reset

[Install]
WantedBy=multi-user.target