

University of Southern Queensland
Faculty of Health, Engineering & Sciences

**Regulating Rescue Package Descent through Controlled
Autorotation**

A dissertation submitted by

Ian Michael Saxby

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Computer Systems Engineering

Submitted: Jan, 2015

Abstract

This dissertation documents the design, implementation and test of a rescue package that is intended to be carried and released by a Remotely Piloted Aircraft System (RPAS) from a height of at least 65m. Current commercial designs for controlled air-drop deliveries include automated parafoil devices. The rescue package physical size is sufficient to contain a commercial 500ml water bottle. When released from the RPAS, the rescue package utilises the helicopter autorotation technique to control a safe descent and landing to a nominated ground point minimising package damage so that a human can use all the water. The design process considers System Safety from both hardware and software perspectives.

The project required the design of both hardware and software of the host and package controllers and a ground based test facility.

University of Southern Queensland
Faculty of Health, Engineering & Sciences

ENG4111/2 <i>Research Project</i>
--

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled “Research Project” is to contribute to the overall education within the student’s chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Dean

Faculty of Health, Engineering & Sciences

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

IAN MICHAEL SAXBY

0050083462

Acknowledgments

Mark Phythian, my supervisor, for his continued patience and understanding as I progressed through this project. Don Luke, Bryan Walker, Harry Melnik and Keith Smith whom have been instrumental in translating my hardware ideas and sketches into prototype fabrication. To my wonderful parents who have inspired in me the commitment and desire to achieve the best . To my beautiful Susanne and my cherished children Michael, Elizabeth and Andrew for their constant support and encouragement.

IAN MICHAEL SAXBY

Contents

Abstract	i
Acknowledgments	iv
List of Figures	xii
List of Tables	xv
Nomenclature	xvi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Aim	2
1.3 Objectives	2
1.4 Context	2
1.5 Ethics and Implications	4
1.5.1 Engineering Ethics	4
1.5.2 Prototype Readiness Implications	4
1.6 Overview of the Dissertation	4

CONTENTS	vii
Chapter 2 Previous Work	5
2.1 Chapter Overview	5
2.2 Previous Work	5
2.2.1 Navigation	5
2.2.2 Transfer Alignment	8
2.2.3 Australian Aerospace Regulatory Regime	9
Chapter 3 Establishing System Requirements	11
3.1 Chapter Overview	11
3.2 Concept of Operations	11
3.3 System Safety Analysis	12
3.4 System Safety Requirements	14
3.5 System Requirements	17
3.6 Chapter Summary	18
Chapter 4 Rotary Deceleration System - Design and Construction	19
4.1 Chapter Overview	19
4.2 Physical Design	19
4.3 Electronics Design	23
4.3.1 System Architecture	23
4.3.2 Processor Selection	26
4.3.3 Power Controller	27

CONTENTS	viii
4.3.4 Pro mini Connector and Quadrature Encoder Sensor boards	28
4.4 Software	30
4.4.1 Interface Design	30
4.4.2 Software Design	32
4.4.3 Software Operation	34
4.5 Critical Design Analysis	37
4.6 Chapter Summary	40
Chapter 5 Verification	41
5.1 Chapter Overview	41
5.2 Verification Test Facilities	41
5.3 Verification Activities	46
5.4 Critical Analysis of Verification Facilities	48
5.5 Chapter Summary	50
Chapter 6 Conclusions and Future Work	51
6.1 Conclusion	51
6.2 Achievement of Project Objectives	51
6.3 Further Work	52
References	53
Appendix A Project Specification	55

CONTENTS	ix
Appendix B System Safety	58
B.1 Appendix Introduction	59
B.2 Safety Requirements Verification Matrix	61
Appendix C System Requirements and Architecture	65
C.1 Appendix Introduction	66
C.2 System Requirements	66
C.2.1 Navigation	66
C.2.2 Power Supply	66
C.2.3 Physical	67
C.2.4 Interface	67
C.2.5 Built In Test	68
C.2.6 Ground Test Facility	68
Appendix D RDS Mechanical Drawings	69
Appendix E Electrical Schematics	81
Appendix F RDS Software Design	85
Appendix G Risk Analysis	91
Appendix H Source Listings	96
H.1 Nameing Conventions	97
H.2 Sensor Manager Listings	99

H.3 The <code>SensorManager.ino</code> Code	100
H.4 The <code>Commander.h</code> Code	109
H.5 The <code>Commander.cpp</code> Code	110
H.6 The <code>BIT.h</code> Code	122
H.7 The <code>BIT.cpp</code> Code	123
H.8 The <code>I2CBuffer.h</code> Code	127
H.9 The <code>I2CBuffer.cpp</code> Code	130
H.10 The <code>Interface.h</code> Code	132
H.11 The <code>Interface.cpp</code> Code	134
H.12 The <code>PinoutConfigSM.h</code> Code	137
H.13 The <code>Power.h</code> Code	138
H.14 The <code>Power.cpp</code> Code	140
H.15 The <code>QuadEncoder.h</code> Code	146
H.16 The <code>BIT.cpp</code> Code	148
H.17 The <code>ServoTimer2.h</code> Code	152
H.18 The <code>ServoTimer2.cpp</code> Code	155
H.19 The <code>StateMachine.h</code> Code	160
H.20 The <code>StateMachine.cpp</code> Code	162
H.21 Release Controller Listings	164
H.22 The <code>HostReleaseController.ino</code> Code	165
H.23 The <code>Equates.h</code> Code	172

CONTENTS	xi
H.24 The Commander.h Code	174
H.25 The Commander.cpp Code	175
H.26 The BIT.h Code	190
H.27 The BIT.cpp Code	191
H.28 The I2CBuffer.h Code	194
H.29 The I2CBuffer.cpp Code	197
H.30 The Interface.h Code	199
H.31 The Interface.cpp Code	201
H.32 The PinoutConfigRC.h Code	206
H.33 The Power.h Code	208
H.34 The Power.cpp Code	210
H.35 The BIT.cpp Code	216
H.36 The StateMachine.h Code	219
H.37 The StateMachine.cpp Code	220
H.38 The MsgBuff.h Code	223
H.39 The MsgBuff.cpp Code	224
 Appendix I RDS Interface Control Document	 226

List of Figures

2.1	Body Frame of RDS	6
2.2	ECI and ECEF Frames	7
4.1	Deployed Position	21
4.2	Conformal Position	21
4.3	(a)Nose Cone. (b) Nose Joiner.	22
4.4	(a)Rotor Tray Top View. (b) Rotor Tray Bottom View.	22
4.5	3D Rotor Lock	23
4.6	RDS Lift Guidance Mechanism Assembled	24
4.7	PowerControllerImage	28
4.8	ProMiniwithConnectorBoard	29
4.9	QuadratureSensor	29
5.1	Ground Test Lift Analysis Rig	42
5.2	Ground Test Lift Analysis Rig	43
5.3	Completed Ground Test Lift Analysis Rig, Ian Saxby 2014	43
5.4	Upper Beam showing pre-load drive unit, Ian Saxby 2014	44

5.5	Drive Unit Engagement fingers and Guide wire, Ian Saxby 2014	45
5.6	(a)Lower Beam Back View. (b)Lower Beam Side View.	46
B.1	Impact to Property or Human	59
B.2	Loss of Control	60
B.3	Power Failure	62
B.4	Mechanical Failure	63
B.5	Deployment Failure	64
D.1	Nose - 3D Printed	71
D.2	Rotor Lock Tray - 3D Printed	72
D.3	Rotor Lock - 3D Printed	73
D.4	Nose Joiner - 3D Printed	74
D.5	RDS Spindle	75
D.6	Bottom Thrust Stop	76
D.7	Lower Rest	77
D.8	Quadrature Encoder Mask	78
D.9	Rotor Head Top Rest	79
D.10	Sensor Assembly Platform	80
E.1	Schematic Battery Controller	82
E.2	Schematic of Quadrature Encoder and Pro Mini Connector Boards	83
E.3	Sensor Assembly Harness	84

LIST OF FIGURES	xiv
F.1 High Level Interconnect System Diagram	86
F.2 Host FMU State Diagram	87
F.3 Host Release Controller State Diagram	88
F.4 RDS FMU State Diagram	89
F.5 RDS Sensor Manager State Diagram	90

List of Tables

3.1	Excerpt from ref ARP4761, Figure D2 - Fault Tree Symbols	14
B.1	System Safety Traceability Matrix	61
G.1	Risk Management Chart for Ground Test Apparatus	92
G.2	Risk Management Chart for Working at Heights	93
G.3	Risk Management Chart for Work Place Safety	94
G.4	Risk Management Chart for Impact of package onto structure or personnel during descent	95
H.1	Type prefix	97
H.2	Type modifier	97
H.3	Scope modifier	98

Nomenclature

ΔN Number of observed pulses inside the observation window

Ω Rotor rotation rate

ΣT_{h-1} Time interval to the first observed pulse after the start of the observation window

ΣT_h Time interval between the last observed pulse before the end of the observation window

$\Sigma T_{sc,acc}$ Time of the basic or extended observation window

N_p Number of Quadrature Decoder pulses per revolution

Chapter 1

Introduction

1.1 Motivation

Established as a part of the rules of the UAVOutBackChallenge competition, is the requirement to deliver a rescue package from an Remotely Piloted Aircraft System (RPAS) from a minimum height of 65m. This rescue package is to contain a minimum of 500ml of water. The safe delivery to a target point on the ground must account for both minimal package damage and safe human interaction during the package's terminal trajectory phase. Existing methods which have been used are deployed parachutes or fixed autorotation vanes which reduce the impact velocity of the released rescue packages. These methods are prone to variations in landing accuracy due to pre release trajectory calculation errors, inconsistent trajectory paths due to poor package aerodynamics and importantly, the variations in trajectory due to wind.

This project proposed to control and decelerate a package using an active autorotation technique from the rotary helicopter environment. The technique has been researched by many for military and civil purposes and is based upon the theory and research related to safe landing of helicopter aircraft following engine power failure via the use of cyclic and collective pitch control during the autorotation phase of flight.

1.2 Aim

To design and prototype a device that can control and decelerate a package using an active autorotation technique from the rotary helicopter environment.

1.3 Objectives

The resultant prototype system shall be capable of:

1. carrying a quantity of 500ml of water within a commercial container,
2. accept and actively descent along a defined path to given target co-ordinates,
3. constrain the terminal landing velocity below 3 m/sec, and
4. provide an environment for safe carriage and purposeful release of the rescue package.

The subordinate objectives include undertaking the project development considering a safe development environment and ultimately production of a safe product at the conclusion.

1.4 Context

CASA Civil Aviation Order 40.3.0 defines autorotative flight means a condition of flight without power when lift and rotor speed are derived from the action of the airflow upwards through the rotor system.

The aerodynamic forces developed through autorotation are the basis for the autogyro which was first developed by Juan de la Cierva who designed and built the Autogyro (de la Cierva & Ray 1931). Cierva's craft used an axially aligned propeller to provide thrust and a rotor for lift instead of a fixed wing. Autorotation is a condition whereby torque is no longer imparted to the rotor by internal means and is instead developed by upward airflow through the rotor. Autorotation is achievable during un-powered descent and is a possible emergency landing technique for helicopters with engine or driven train

failures. Given the emergency situation, the upward movement of airflow imparts kinetic energy into the rotor as the craft descends which is then translated into thrust close to the ground through a complex pilot initiated flare manoeuvre.

The autorotation technique has been pursued as a possible alternate deceleration technique for landing spacecraft (Wernicke 1959), personnel (Lambermont & Pirie 1959) and provisions (AIA 2003). Interest in helicopter safety and Unmanned Aircraft Systems (UAVs) landing control have seen researchers pursue controlled autorotation as a means for safe landing of helicopters or UAVs in emergencies or as the basis for accurate delivery to a given location. Research outcomes have proposed and validated algorithms to plan and execute the controlled glidepath and safe landing using optimisation techniques. Johnson, (Johnson 1977) derived a non-linear model that accounted for vertical and longitudinal movement. Johnson's optimal control used a cost function that minimised horizontal and vertical speeds on landing. This optimisation used forwards and backwards numerical integration between defined boundaries using the steepest descent method.

The objective of this project can be broken down into multiple facets:

1. Power Control
2. Communication
3. Target Transfer
4. Release Preparation
5. Release
6. Blade Deployment
7. Rotor Spin-Up
8. Autorotation Glide
9. Autorotation Flare
10. Path Planning and Control

1.5 Ethics and Implications

1.5.1 Engineering Ethics

The Engineers Australia Code of Ethics defines the values and principles that shape the decisions engineers make in engineering practice. Within the Code of Ethics are guidelines on Professional Conduct that provide a framework for members of Engineers Australia to use when exercising judgment in the practice of engineering.

The pursuit of an engineering solution to improve the process of rescuing people or sustaining stranded people is seen by the Author as a humanitarian goal.

1.5.2 Prototype Readiness Implications

Though this engineering effort is focused on providing an alternate solution to methods of accurately delivering supplies to stranded personnel, there are many issues to consider within the current aerospace regulatory environment. These issues such as software assurance must be addressed to satisfy airworthiness and safety regulations. This project provides a small insight into the broader effort a commercial product would require to be considered safe and fit for purpose.

1.6 Overview of the Dissertation

This dissertation is organised as follows:

Chapter 2 Previous Work,

Chapter 3 Establishing System Requirements,

Chapter 4 Rotary Deceleration System - Design and Construction,

Chapter 5 Prototype Verification, and

Chapter 6 Conclusions and Future Work.

Chapter 2

Previous Work

2.1 Chapter Overview

This chapter discusses the basics of Navigation, Transfer Alignment and the Regulatory Environment that must be considered when developing the RDS prototype.

2.2 Previous Work

2.2.1 Navigation

Navigation control requires the understanding of basic navigation reference frames and the earths geometry. Positional control of the RDS both pre and post release requires the translation of the navigation information between these reference frames. The basic frame is the body of the RDS. This body frame is defined by three axes in which on board sensors are aligned that measure accelerations and angular rates. To control the RDS from a point in flight to a point on the earth, these body co-ordinates must be translated to the earth frame. Each of the navigation frames have specific purposes.

The Body Frame. The axes of the Body Frame at Figure 2.1 are:

1. The origin coincides with the Center of Gravity of the RDS.
2. The x-axis points laterally across the RDS. This axis is called the pitch axis.

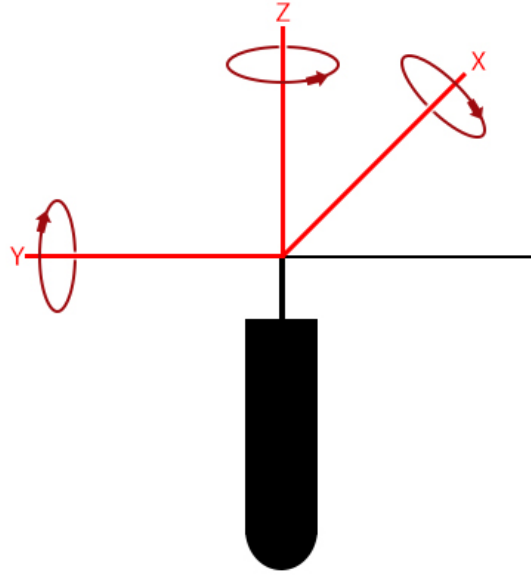


Figure 2.1: Body Frame of RDS

3. The y-axis points forward longitudinally. This axis is called the roll axis.
4. The z axis points towards the vertical direction. Yaw angle is measured around this axis.

This frame is referred to as the b-frame. The Roll, Pitch and Yaw angles (RPY) around these Body Frame axes are known as the Euler angles. The RPY rotation angles correspond to rotations around each respective axis using the right hand rule.

Earth-Centered Inertial (ECI) Frame. The inertial frame as defined by (Grewal, Weill, & Andrews 2007) is stationary in space with its origin at the center of gravity of the earth. The axes of the ECI Frame are:

1. The X_{ECI} is in the equatorial plane and points in the direction of the vernal equinox ,
2. The Z_{ECI} is parallel to the rotation of the earth coincident with the North polar axis, and
3. The Y_{ECI} is orthogonal to the X_{ECI} and Z_{ECI} axes and completes the right handed system.

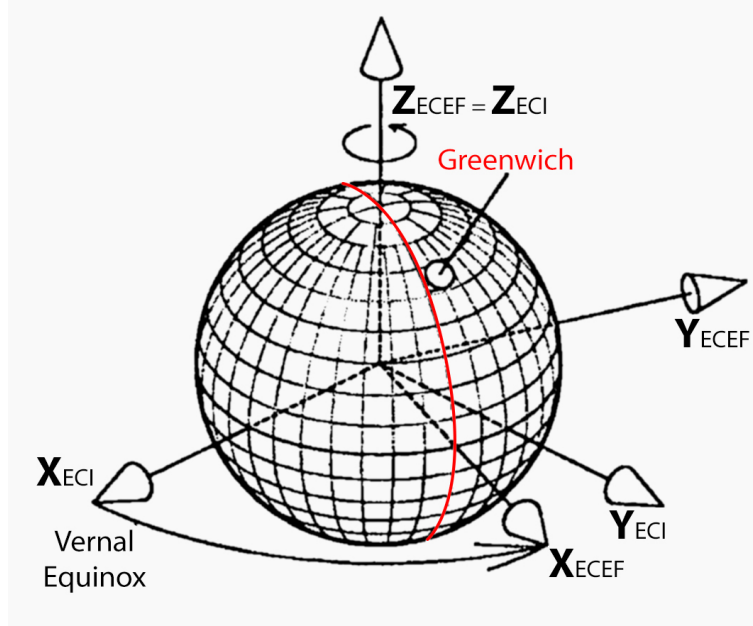


Figure 2.2: ECI and ECEF Frames

This is the frame of choice for near Earth environments. This frame is referred to as the i-frame.

Earth-centered Earth Fixed (ECEF) Frame. Referred to as the e-frame, the ECEF Frame shares the same origin and z-axis as the ECI Frame though rotates with the earth. The axes of the ECEF Frame and their relationship with the axes from the ECI frame are shown in Figure 2.2 taken from (Grewal et al. 2007). Specifically, these axes are:

1. The X_{ECEF} , which passes through the equatorial plane and the Greenwich meridian.
2. The Z_{ECEF} passing through through the North polar axis, and
3. The Y_{ECEF} , and in the equatorial plane orthogonal to the X_{ECEF} and Z_{ECEF} axes and completes the right handed system.

Local Tangent Plane (LTP) coordinate systems represent local reference directions for RDS attitude and velocities. Two such right handed LTPs are the East-North-Up (ENU) and North-East-Down (NED) coordinate systems. The NED coordinate axes coincide with RDS Euler Angles (Roll Pitch Yaw) coordinates when level and the RDS is facing the North direction.

Transforming one co-ordinate frame to another is achieved by techniques using direction cosines, Euler rotations or quarternions.

2.2.2 Transfer Alignment

To control and guide a RDS from an aircraft requires the knowledge of position, attitude, velocity, angular rate and accelerations. The aircraft and RDS will utilise IMU and Global Positioning Systems (GPS) to provide these inputs. The IMU measures then integrates specific forces and angular rates to gain knowledge of current position, velocity and attitude. These actions must be initialised with an accurately known starting point. As bias errors caused through measurement, manufacturing or introduced noise affects these integration algorithms IMU systems do not have long term accuracy. To compensate, long term stable information from GPS systems can be blended into the measurement process.

Both the RPAS and RDS will contain their own IMU and GPS units, and will be calculating separate navigation solutions. The RDS however will be carried at 90° to forward flight such that the GPS unit may not correctly receive signals to provide this long term stability to the RDS's Navigation solution. Accordingly, drift may occur with the RDS Navigation Solution. To correct this drift, Transfer Alignment can be used to calibrate and maintain the RDS (Slave) Inertial Navigation System (INS) from the host aircraft's Master INS.

The quality of Transfer Alignment is a significant factor in the Military aerospace environment as weapons are attached to the underside of metal aircraft. This position masks the weapon slave INS from receiving GPS signals (if fitted). A further factor in the military environment is that flight weapons that require control usually have quite small INS due to space constraints. These systems invariably are not activated until just prior to release and require initialisation of their navigation solution from the Master (more accurate) INS onboard the parent aircraft.

(Groves 2003) identifies the optimised Transfer alignment estimates:

1. Attitude and velocity;
2. Accelerometer and gyro static and dynamic biases;
3. Accelerometer and gyro scale factor and cross coupling errors; and
4. Static relative lever arm and force-dependent relative orientation coupling.

The transfer alignment algorithms utilise Kalman filters to provide estimates of the errors from the Slave INS to that of the Master. Further, as sighted within (Groves 2003) manoeuvres undertaken by the host aircraft during the alignment affects the performance of the estimation process. These manoeuvres isolate the states that are estimated by the Kalman filter. For example by changing altitude, error sources within attitude and velocity measurements can be observed.

2.2.3 Australian Aerospace Regulatory Regime

The Civil Aviation Safety Authority (CASA) has published regulations that pertain to Model and Unmanned aircraft within (CASA 2002). These current regulations were established in Jan 2002. The extension objectives of this project were proposed to be achieved through carriage of a RDS from a small UAV aircraft operating below 400 feet AGL outside controlled airspace in accordance with the (CASA 2002) regulations.

In May 2014 CASA released Notice for Proposed Rule Making (NPRM) 1309OS - Remotely Piloted Aircraft Systems (CASA 2014) that proposes changes to these regulations. The NPRM introduces specific regulations for Remotely Pilot Aircraft Systems (RPAS) with aircraft weight between 2kg and 150kg that require CASA approvals and Operator Certification if operated outside the following standard operating conditions:

1. the RPA's remote pilot can directly see the RPA, with or without corrective lenses, but without the use of binoculars, a telescope or other similar device; and
2. the RPAS is being operated below 400 ft AGL in VMC by day; and
3. the RPAS is not being operated within 30 m of a person who is not directly associated with the operation of the RPA; and
4. the RPAS is not being operated:
 - (a) in controlled airspace; or
 - (b) in or over a prohibited or restricted area; or
 - (c) over a populous area; or
 - (d) within 3 nautical miles of an aerodrome.

The impact of these proposed rules were to be on the extension objectives of this task which were not completed.

Chapter 3

Establishing System Requirements

3.1 Chapter Overview

This chapter describes the development of system requirements for the delivery platform and the expected interaction with the host RPAS. This development accounts for considerations of system safety and the chapter details the corresponding safety related requirements that are imposed. The requirements encompass the hardware, software and interface between sub components.

3.2 Concept of Operations

To satisfy the objectives of the UAVOutbackChallenge a 500ml water container must be accurately dropped from a minimum height of 65m and the container must remain intact following the resultant ground impact. The rescue package size and shape must therefore be streamlined if externally carried to the drop point and needs to include some form of deceleration system to minimise the ground impact speed to an acceptable level. A further expectation of this project is to include navigation, guidance and control systems to ensure accurate delivery of the rescue payload under varying wind conditions. The likely carriage time during the transit to the release point is up to 50 minutes with a final descent flight time of about 30 seconds.

A further natural expectation is that the payload can be easily loaded or accessed by both

dispatch and receiving ground personnel respectively.

Given this projects objective is to utilise Autorotation as the deceleration methodology and using helicopter rotor blades as lift surfaces, the drag from these rotor blades must be minimised during the carriage phase to the release point. Due to the expected length of rotor blades, the system is expected to be suspended external to a host RPAS. Accordingly, the system design must retain the blades conformal to the system body during transit to the release point, for deployment in the descent phase.

The ground objective position is expected to be passed to the system as part of the pre-release communication. Post release requires controlled deployment of the autorotation decelerator, build-up and management of kinetic energy in the form of rotor angular velocity and translation of this kinetic energy into lift to minimise ground impact speeds.

The project is to include a system safety based requirements analysis phase to consider possible hazards related to safe carriage, authorised release and flight modes of the deployed system.

3.3 System Safety Analysis

Development of products that have safety implications require conformance to a System / Software Safety Standard. This project utilised elements of an earlier version (E) standard (DOD 2012) due to the disclosure of hazard analysis tasks that are missing from the current version. (DOD 2012) defines a series of management, development and reporting tasks to provide a consistent means of reporting risks. The limited subset of tasks incorporated into this development effort were:

1. Task 106 Hazard Tracking System
2. Task 201 Preliminary Hazard List
3. Task 202 Preliminary Hazard Analysis
4. Task 301 Safety Assessment Report

This subset was chosen from the standard due to the short duration of the task and the single individual involved limiting the ability to incorporate any independence of review.

The risk acceptance criteria was modified to incorporate involvement of the Project Supervisor in consideration of the exposed and treated risks, though no such involvement was necessary.

To satisfy the operational concept a System Safety analysis was undertaken. The results of this analysis determined that there were hazards related to the program of development and the eventual product.

Undertaking the safety analysis exposed safety related issues from the Workplace Health and Safety (WHS), and system safety disciplines. The WHS issues pertained to the phases of both development and test. The development phases included manufacturing requiring the manipulation of hand tools and machinery, whilst people involved in ground testing were exposed to moving parts. A further WHS hazard exposed was the requirement to briefly work at heights during the install the ground test stand apparatus at a height of 4m.

From a System Safety perspective, the solution can mitigate hazards exposed during development and test through a system safety design order of precedence:

1. Elimination through design selection,
2. Incorporation of safety devices,
3. Provision of warning devices, and/or
4. Development of procedures and training.

Workplace Health and Safety hazards are disclosed within Appendix G of this report.


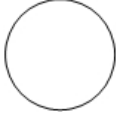

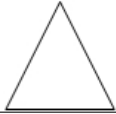
The initial ground test phase exposed the physical hazards of moving parts, machinery and working at heights. System Requirements were raised to account for the mitigation to these hazards and are included in design of the ground test rig.

1. Moving parts;
2. Working at heights;and
3. Work place safety.

System Safety Analysis was undertaken using Functional Hazard Analysis (FHA) and the associated Fault Tree Analysis (FTA) techniques from (Aircraft, Dev & Committee 1996). As defined within (Aircraft et al. 1996) a FHA looks at system functions and combinations of system functions to identify and classify the associated failure condition(s). The FTA method is an analysis which focuses on one particular undesired event and deduces causes of this event. The FTA considers both loss of functions and malfunctions.

The FTA output is a graphical hierarchical representation of the relationships between failure effects and failure modes underpinning the single project system safety hazard. FTA are defined by two types of symbols, logic and event. Logic symbols, Boolean logic AND-gate or OR-gate, are used to link the various events of the Fault Tree together. The FTA event symbols are defined within Table 3.1.

Table 3.1: Excerpt from ref ARP4761, Figure D2 - Fault Tree Symbols

	Event	description of an output of a logic symbol or of an event
	Basic Event	Event which is internal to the system under analysis, requires no further development
	Undeveloped Event	Event which is not developed further because it has little impact on the top level event or because the details necessary for further event development are not readily available.
	Transfer	Indicates transfer of information.

3.4 System Safety Requirements

The process of undertaking an FHA identified a single hazard; Impact to Property or Human.

A System Safety FTA was conducted on this identified hazard. The resultant FTA is shown in Appendix B Figures B.1, B.2, B.3, B.4 and B.5 .

The system safety requirements raised as mitigations and the associated rationale to

mitigate the Hazard of impact to Property or Human are summarised below.

Figure B.1 identifies two possible causes of impact to property or humans; that of passing an incorrect Ground objective to the RDS and secondly, when given a valid ground target objective the RDS undertakes incorrect flight control actions during descent. The localisation and determination of the actual ground objective is not part of this project and is assumed to be valid, however, this project must account for correct transfer of target information from the host to the RDS. Accordingly, a System Safety requirement was established to ensure correct target transfer before the host RPAS releases the RDS.

SSR1: The System shall validate the specified target co-ordinates have been correctly passed to the RDS prior to release.

The hazards second causal factor of incorrect flight control actions has multiple sources which are exposed in a series of subordinate FTA branches, Figure B.2, Figure B.3 and Figure B.4. Hierarchically the highest branch is Figure B.2. Incorrect RDS flight control can be caused by either loss of control or release of the RDS outside of controllability limits.

Breaching controllability limits could be due to the lack of readiness of the RDS flight state or the release point is too low, airspeed too fast or the RDS simply can not reach the target due to the distance between release point and target.

To mitigate the breach of controllability limits the RDS or host could validate that the launch region and release flight criteria are within defined limits prior to release. Accordingly safety requirements are defined as:

SSR2: The System shall ensure the RDS release conditions are within defined limits prior to release. SSR3: The System shall ensure that the RDS is in a functional state appropriate for release.

A further two factors underpin the possible loss of control; a critical component failure or the failure of the system to adequately control the descent phase of the RDS.

In considering Critical component failure the possible causes were identified as Actuator failure and flight mechanism failure. Actuator failure may be caused by electronic, power supply or mechanical sources, whereas flight mechanism failure relates to rotor blades,

swashplate or linkages and the rotor deployment mechanism. Within Actuator failure, electronic failures could be within the servo itself or the origin command system within the Flight Management Unit. Servos directly manipulate the swashplate, controlling roll, pitch and collective. Failure of the servos to accurately translate command signals to physical swashplate movements through delay or malfunction would cause loss of control. No mitigation exists should any actuator fail during descent. No failure detail could be located about the specific servos used in the design or FMU failure rates so this event was identified as undeveloped. A solution may be the undertaking of specific failure analysis on these items before product commercialisation is undertaken.

The exposure timeframe to a power supply failure is limited to the descent of about 30 seconds. Power Supply failure analysis identified battery failure and circuit failure as possible causes. No evidence was found of manufacturers Mean Time Between Failure (MTBF) data of battery systems used in model equipment so this event was identified as undeveloped. Analysis focused on defining a requirement to establish confidence in RDS supply availability and remaining capacity during the carriage phase:

SSR4: The System shall verify RDS power supply availability and remaining capacity are within correct boundaries prior to authorising release.

The remaining causes of Critical Component Failure were predominately mechanical in nature and are constrained by commercial availability. To mitigate component failures within actuators and flight mechanism (rotor blades, swashplate or linkages), commercial hobby resellers were approached to gather information on the nominal hobby swashplate, rotor heads, rotor blades, servo torque and speed capacities that are used in commercial model helicopters of a similar physical size and weight. This informed the purchasing process of items used in this project. As the rotor deployment mechanism is to be designed within this project so its development must consider mitigations to the causes of rotor blade deployment, hinge mechanism and rotor lock engagement failures.

SSR5: The Rotor Deployment mechanism shall force the rotor blades into the airstream following release.

The safety requirements related to hinge and locking mechanism failures could not be articulated with any relevant discipline experience, however, the design and manufacturing process allowed the proof testing of hinge and lock artefacts as described in the later

Verification chapter, Chapter 5.

Two causes were identified for the Failure to adequately control event; Sensor failure and anomalous Software function. The window of exposure to Sensor failure is both during the pre-release and descent phases as sensor data is expected to be used when confirming RDS release readiness as well as for guidance and navigation during descent. Anomalous software function, through incorrect specification, or failure to satisfy the specification could contribute to a failure event. Software Testing is presented within this project to provide an increased level of assurance in mitigating this possible event.

A component of self check is considered appropriate for inclusion to mitigate these two possible events. Accordingly, the last safety requirements are:

SSR6: The System shall carry out periodic Built in Test (BIT) functions on guidance, control and navigation sensors prior to authorising release.

SSR7: The system shall utilise self checking software logic to confirm correct operation prior to authorising release.

3.5 System Requirements

System Requirements, disclosed within Appendix C, have been segmented into the following objective areas:

1. Navigation,
2. Power Supply,
3. Physical,
4. Interface,
5. Built In Test, and
6. Ground Test Facility.

3.6 Chapter Summary

This chapter described the process and outcomes of the System Safety analysis and Requirements development phases. Seven System Safety Requirements have been identified to mitigate the single hazard Impact to Property or Human.

Chapter 4

Rotary Deceleration System - Design and Construction

4.1 Chapter Overview

This chapter discloses the design process and outcomes within the physical, electrical and software discipline related activities undertaken within this project to design and construct the Rotary Deceleration System (RDS). The detail is referenced in diagrams, drawings and software listings contained within the attached Appendices. Finally, a section on critical analysis is discussed that identifies recognised flaws in the design and implementation process and resultant outcomes.

4.2 Physical Design

The RDS design is segmented into six sub-assemblies:

1. Nose Assembly, containing the power supply, controller and regulator;
2. Rotor Lock Assembly, retention of Rotor blades conformal to the RDS body;
3. Body Assembly containing the payload, host interconnect and wiring loom;
4. Electronics Assembly containing the Flight Management Unit and Sensor Manager (Sensor Manager).

5. Rotor Assembly containing the actuators, spindle and rotor lift and control components (swashplate, alignment, rotor head and blades); and
6. Sensor Assembly situated above the rotor head. The Wiring loom routes to the sensor assembly from the Electronics assembly through the rotor spindle.

These assemblies were designed using the 3D CAD modeling package AutoDesk Innovator (Student Version). The Innovator CAD application allows the export of completed designs to .stl format which were dispatched for 3D printing. In all, the following components were manufactured using 3D printing services:

1. Nose,
2. Battery holder,
3. Lock tray,
4. Rotor lock,
5. Rotor lock interconnect,
6. Nose joiner, and
7. Rotor blade holder, pivot and lock.

3D printing allowed for complex shapes to be developed and manufactured simply and quickly. An example is that of the Nose cone which has a cylindrical slot that the battery holder slides into, Figure D.1. This allows for bench testing of the battery system without the bulk of the Nose cone. The available space within the Nose cone is used to stow the Regulator, batteries and Power Controller. Close inspection of shows the set of four keyed attachment lugs that allow simple detachment of the Nose from the Rotor Lock assembly to access power supply and Rotor Lock components.

The printed Rotor Holder, Pivot and Lock is shown in Figures 4.1 and 4.2 .

Figure 4.3 provides a cross section view of the Nose and Nose joiner component internal structure.

The Designed Rotor Tray component can be seen at Figure 4.4.



Figure 4.1: Deployed Position



Figure 4.2: Conformal Position

The Rotor Tray bolts to the Nose Joiner and positions the Rotor Lock shown at Figure 4.5. When in the locked or closed position the ends of the Rotor Lock fit into slots within the Rotor Tray to provide strength and stability against release due to vibration.

An original project expectation was to machine as many components from Aluminum as possible to establish a prototype that would survive as many test activities as required. Due to cost restrictions this was not achievable. The risk of using the 3D printed rotor holder, pivot and lock components was considered too high and all verification activities were undertaken without those components. Regardless, as the project did not achieve as many original objectives, the fact of not using these components did not detract from the results of testing undertaken.

The Rotor Assembly required the design and manufacturing of the following components from Al Alloy 7075-T6 or Al light alloy:

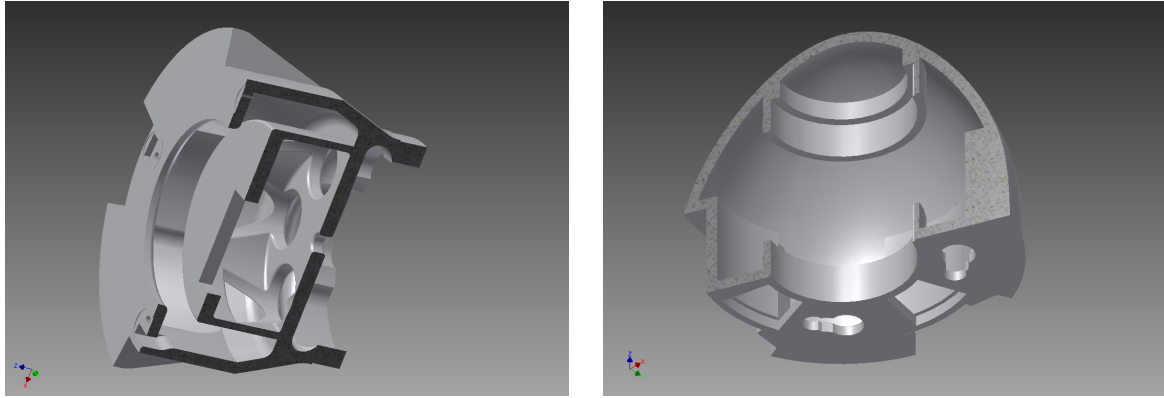


Figure 4.3: (a)Nose Cone. (b) Nose Joiner.

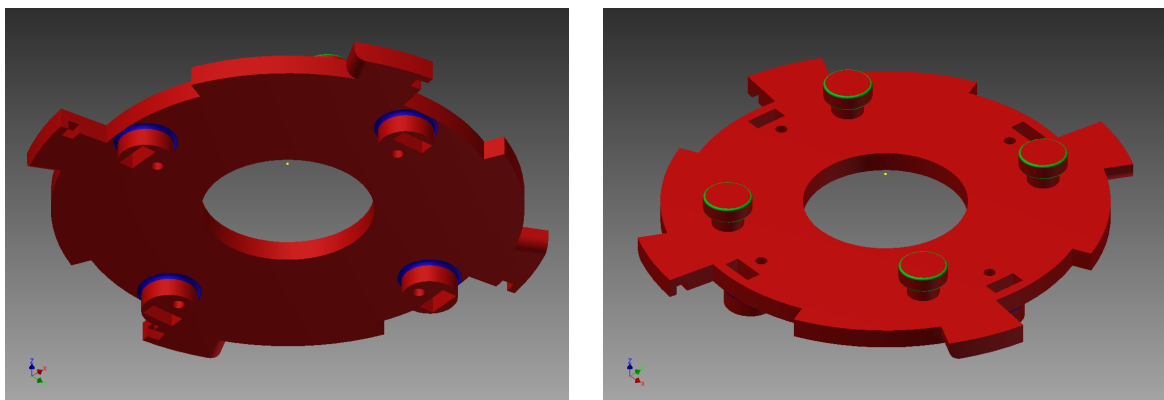


Figure 4.4: (a)Rotor Tray Top View. (b) Rotor Tray Bottom View.

1. Spindle,
2. Actuator bracing,
3. Bottom thrust stop,
4. Rotor lower rest,
5. Rotor holder,
6. Quadrature encoder mask, and
7. Rotor top rest.

The mechanical drawings for each of these items are contained within Appendix D. The complete assembly can be seen in Figure 4.6.

Small 5mm Inside Diameter (ID) x 8mm Outside Diameter (OD) bearings are dispersed along the spindle separated by 2mm wide spacers of 5mm ID x 6mm OD to allow for spindle flexure without binding the outer bearing races against the next bearing. On

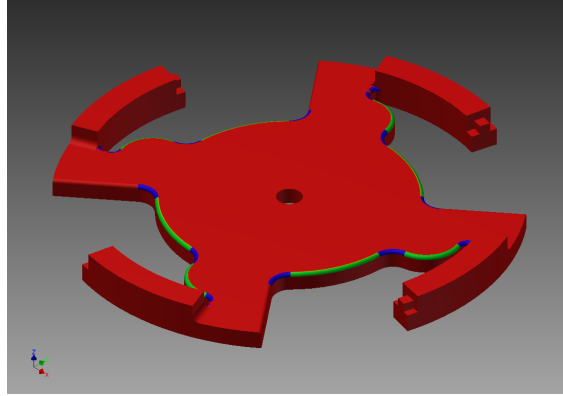


Figure 4.5: 3D Rotor Lock

these bearings the various components of the rotor assembly are positioned and are free to rotate, refer Figure 4.6. Two sets of 8mm ID x 16mm OD thrust bearings are used to contain the rotor head components vertically on the spindle and are held in place by the Rotor Lower and Upper Rest components. A number of radially positioned grub screws within the Rotor Upper Rest hold the whole rotor assembly together. This containment design is the weakest facet of the design as slippage of the grub screws would allow vertical separation of the rotor head assembly and resultant loss of RDS.

The RDS platform is to use a three point swashplate to control the roll, pitch, and collective through Servo Cyclic and Collective Pitch Mixing (CCPM), referred to as servo mixing. Servo mixing in software provides a mechanically simpler design without sacrificing accuracy. Three servo actuators are situated below the swashplate as per model helicopter designs. The CCPM swashplate design was chosen for simplest assembly and associated functional support within common Flight Management Unit open source software.

The sensor assembly is attached to the Rotor Upper Rest with the wiring loom routed through a center hole to match the spindle tube.

4.3 Electronics Design

4.3.1 System Architecture

Guidance, Navigation and Control of a model helicopter is currently under active development and support through the open source application APM:Copter. The APM:Copter

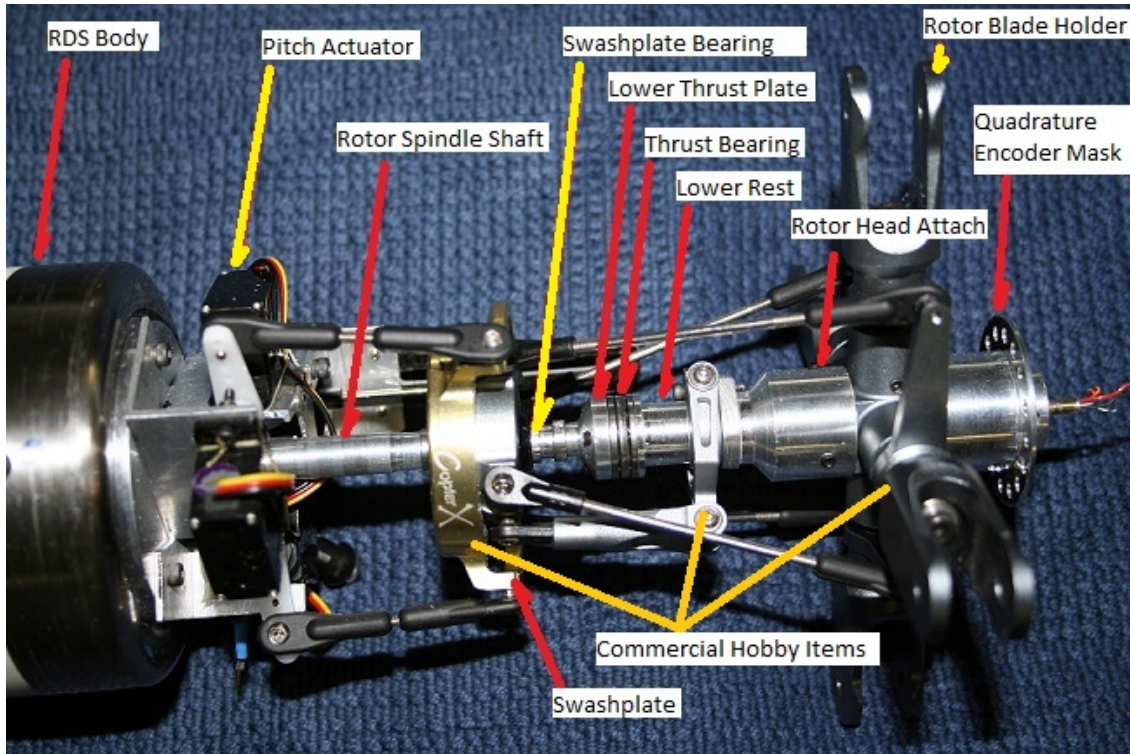


Figure 4.6: RDS Lift Guidance Mechanism Assembled

application has a ground based controller software package also open source, Mission Planner. The APM:Copter application is intended to be hosted on another open source hardware product; the PX4 FMU and daughter board PX4IO. For the sake of simplicity for the remainder of this report the combination of PX4FMU and PX4IO daughter board will be called FMU.

The DO-178B standard (States 1993) provides "guidance for determining, in a consistent manner and with an acceptable level of confidence, that the software aspects of airborne systems and equipment comply with airworthiness requirements". Of interest to this project, DO-178B identifies two techniques called Partitioning and Safety Monitoring. Inclusion of partitioning in the design accounts for the isolation of functionally independent software components so as to contain and/or isolate fault conditions. Safety Monitoring protects against specific failure conditions by directly monitoring a function for failures which would contribute to the failure condition. These two techniques were used within this project. Specifically, parallel independent initiation and detection of the readiness of the RDS for release, and monitoring of aspects of sensor or power supply data to ensure critical components are operating correctly prior to release.

Application of these two techniques led to the development of the System Architecture

and functional allocation as identified within Figure F.1. Readiness for Release and Built in Testing satisfies two System Safety Requirements. Readiness for Release is achievable through two functionally independent communication links. The first via serial communication between FMUs, whilst the second is via discrete signals between Sensor Manager and Release Controller. Without the Release Controller receiving confirmation through these two methods release is not initiated.

The constant review of Built in Test results from all processor boards provided assurance of correct operation prior to release.

Considering Figure F.1, the allocation of functions is as follows:

The Host FMU is responsible for flight control management of the host RPAS. The FMU interfaces to host systems and sensors for RPAS navigation, guidance and control of all host functions. The RDS system is to compare and align to the master host FMU navigation solution. The Host FMU has communication links to the ground station controller and provides status and command link for management of RDS release.

The RDS FMU is the master controller of the RDS. The RDS FMU interfaces to the Host RDS via Universal Asynchronous Receiver/Transmitter (UART) serial communication protocol. Similarly to the host FMU, once the RDS is released the RDS FMU uses information from RDS sensors (GPS / Compass / Accelerometers and rate gyros) for flight guidance, navigation and control.

Each FMU can communicate using UART and I2C serial communication protocols. The FMU is the master I2C node whilst it can be a Master or slave on the UART serial link.

The Sensor Manager is a slave on the RDS I2C serial network. The RDS Sensor Manager controls and receives status information from the Power Controller. The Sensor Manager also collates status information from the host interconnect and independently controls the Commit to Release discrete to the host. The RDS Sensor Manager presents status information to the RDS FMU for the following functions:

1. Host connection interlock and debug signals
2. RDS power
3. Commit to Release state.

4. Rotor head speed and direction.

The Host Release Controller is a slave on the Host I2C serial network. Similar to the RDS Sensor Manager, the Host Release Controller (RC) controls and receives status information from the host power controller, RDS interconnect and controls the Release, Lock/ Unlock and Extension servos. The Release Controller independently determines and notifies the Host FMU of RDS assertion of Commit to Release.

Both of the Sensor Manager and Release Controller utilise the same core Operating System and classes to achieve the assigned functions. Slight modification are introduced into each to support some unique functions.

4.3.2 Processor Selection

The FMU provides adequate memory and speed capacity to undertake this project. The FMU has on board 3 axis accelerometers, rate gyros and compass. The FMU also connects to external GPS, Compass and communication subsystems.

The space considered available to undertake the independent release readiness checks and communication required identification of a small processor footprint that had relatively high throughput, sufficient Input/Output (I/O) discrete, Pulse Width Modulation (PWM) capabilities and 5V power supply tolerance. The Arduino Pro Mini utilises the 16MHz ATmega328P processor at 5V and has 32K Flash, 1K EEPROM and 2K internal Flash memory. The circuit board footprint is 18x33mm due to elimination of IDE programming interface logic. The Pro Mini has 8 Analog and 14 Digital I/O pins, with each of the Digital I/O pins capable of setup as an interrupt. An additional advantage is that the board does not come pre-populated with connector pins, which is an important aspect allowing attachment to the RDS Sensor Assembly and FMU wiring loom through specific low profile DF13 connectors.

The Pro Mini Arduino is capable of running a small Real Time Operating System, NilRTOS. NilRTOS is available as an Arduino library and provides a minimum set of pre-emptive scheduling and synchronisation functions. NilRTOS is statically mapped at compilation time and does not allocate memory at runtime. NilRTOS does not utilise timer2 allowing real time operation whilst servo PWM is achieved through use of timer2. Nil-

RTOS is open source and is unproven in terms of correct verifiable operations, however, would provide the project with the ability to leverage off multiple independent functional threads to carry out the Safety Monitoring whilst supporting Quadrature Encoder interrupts and serving the I2C communications port. No specific testing was undertaken to verify NilRTOS operation beyond RDS functional testing.

4.3.3 Power Controller

Design requirements identified that the power controller needs to be powered from the internal battery at all times to enable control of the power output regardless of connection to the Host RPAS, as would be the case during descent. Control of external and internal supply by the power controller board would be commanded through digital 5V logic levels enable signals.

The following is a description of the Power board, Figure 4.7 operation and associated component involvement. Refer to Appendix E Figure E.1. Internal RDS battery supply is two LiFEP04 batteries in series configuration providing 6.4V DC with capacity of 1300mAh. LiFEP04 was selected for its high density, low cost, low toxicity and high thermal stability, (Dupr, Martin, Degryse, Fernandez, Soudan & Guyomard 2010). Passing through a commercial 5V regulator the internal power supply is connected to the Power Controller board via connector J3. External power is passed through the RDS wiring loom and connected to the Power Controller Board via connector J1. RDS Power is available at connector J2 and passes back through the wiring loom to the Electronics assembly.

The power controller circuit includes selection and status monitoring of external and internal power. Internal power monitoring includes monitoring of battery remaining charge capacity and voltage level. Selection of power supply source is achieved by two discrete input signals which enter the Power Controller board on a single six pin connector J4. J4 also includes status and monitoring signals presented to the wiring loom. A Hirose DF13 connector is used to provide security of cabling in the vibration environment whilst Hirose DF3 are used for the higher current power routing.

Power source for U1, U2 and U4 is the 5V regulated internal RDS battery power supply. U3 is powered from the applied voltage at either VIN1 or VIN2 which ever is available.

U3 controls two discrete N-channel MOSFETs emulating an ideal diode. U3 has Enable inputs EN1 and EN2 (Active Low) that control the availability of the corresponding power supply to the output connector J2. U1 provides for Opto-isolated translation of external power application into a sensed voltage level as an inverted logic signal to U2 input B. The External Power Supply enable signal, is applied to the other U2 input, input A from connector J4. When the External Power Supply enable signal is LOW, U2 passes the state of the external supply through to U3 EN1. Accordingly, when external supply is available and U2 Input A is LOW the U1 signal state is presented to turn on U3 supply 1 (External supply). In contrast, if the External Power Supply enable (U2 input A) is asserted HIGH, U3 turns off the associated MOSFET and high current external supply is disabled.

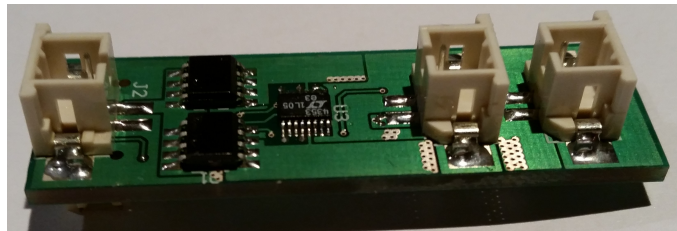


Figure 4.7: PowerControllerImage

U1, U2 and pull-down resistor R3 provides for automatic external power application during start-up of the Sensor Manager.

EN2 is connected to the Internal Power Supply enable signal and is defaulted to the enabled state (Active LOW) via the pull-down resistor R4. This condition is required should any interruption of internal power be realised during descent. When power is recovered this default LOW will immediately pass internal power supply to RDS circuits.

U4 provides for measuring of current and voltage of the internal battery. These status signals are presented to external use on connector J4.

4.3.4 Pro mini Connector and Quadrature Encoder Sensor boards

The Pro Mini Connector board provides for quick, yet secure, connection and disconnection of the Pro-Mini circuit board from the wiring loom. The associated circuit of, Figure 4.8, is disclosed in Appendix E Figure E.2. Connection between the board and pro mini is via soldered short multi-strand wires. The connectors are the standardised six pin

Hirose DF13 that have sufficient current capacity to carry the interface signals and single rotor lock servo signal line.

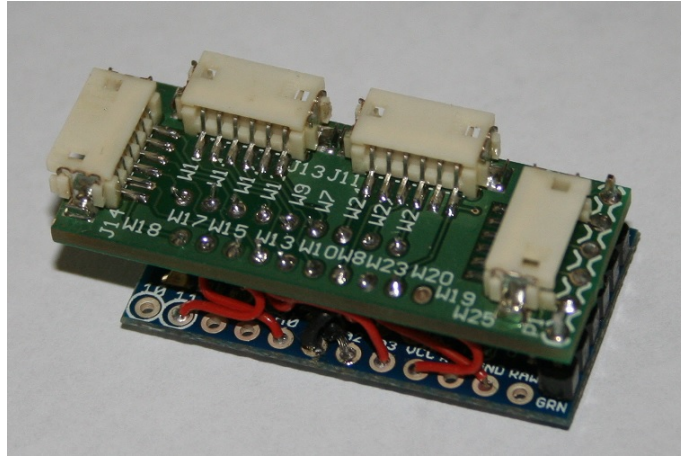


Figure 4.8: ProMiniwithConnectorBoard

The Quadrature Encoder Sensor board, Figure 4.9, is a small circuit with three components. Appendix E Figure E.2 contains the schematic. The two sensor boards receive their power and ground connections from the common Battery Controller supply output, nominally 5V DC. Resistors R6 and R7 establish the necessary forward bias current through the opto sensor. The sensor has a 3mm air gap between Infrared (IR) source and sensor diode within which the Quadrature Encoder mask spins. Two 3mm mounting holes attach each sensor to the underside of the Sensor Assembly and lock the relationship of sensor to mask consistently. The wiring loom connects to the sensor boards using Hirose DF13 connectors.

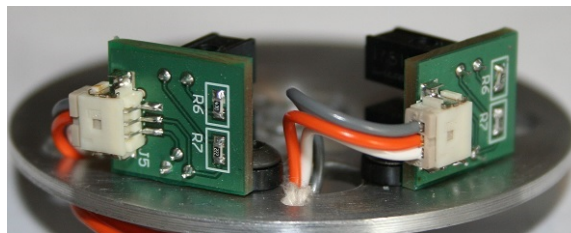


Figure 4.9: QuadratureSensor

The layout of the Sensor Assembly wiring loom routed through the Rotor spindle is shown at Appendix E Figure E.3.

4.4 Software

4.4.1 Interface Design

An Interface Control Document disclosed within Appendix I was written to define all possible communication between each of the sub systems shown in the System Architecture Diagram, Figure F.1. The Interface Control Document assigns the interconnect discretises and pin layout between the Host and RDS and the communication messages between Host and RDS FMUs, RDS FMU and Sensor Manager and Host FMU and Release Controller. The inter FMU communication uses the UART serial protocol, whilst FMU to Sensor Manager and FMU to Release Controller is via I2C, with the FMU in each later case as the I2C Master Node.

Identification of communication direction, regardless of UART or I2C protocol is in relation to the Slave Nodes. That is, Receive (Rx) messages are those expected to be received by the Slave from the Master, whilst Transmit (Tx) messages are requested from the Slave by the Master Node. In summary, six Rx messages are used to transmit commands from the RDS FMU to Sensor Manager. They are:

1. Msg 1: Used to notify the slave I2C node of the Message Identifier in the subsequent Tx Msg request. This allows the Slave Node to prepare the Tx buffers with the required data;
2. Msg 2: FMU Controlled State change request;
3. Msg 3: Weight off Wheels status;
4. Msg 4: Initialisation of Power Supply capacity. Intended to originate from the ground station and routed through the FMUs on power application;
5. Msg 5: FMU State Change Acknowledgment. Intended for use by the FMU to acknowledge Sensor Manager critical state changes; and
6. Msg 6: Debug command Override. Presently this is not implemented and debug code segments were instead spliced into the code during verification activities. Further details are provided within Chapter 5.

Six Tx messages are identified for transmitting Sensor Manager status information back to the RDS FMU. These are:

1. Msg 0: Sensor Manager State;
2. Msg 1: Quadrature Encoder Speed and Direction;
3. Msg 2: Internal Battery Current Usage;
4. Msg 3: Power Mode State. Identification of Selected Power Supply Source;
5. Msg 4: Built In Test Result; and
6. Msg 5: Interlock status. These include RDS Interlock Status, Debug and Commit to Release Interlock Status.

The Host Release Controller uses many of the same software functions as the Sensor Manager and in-turn many of the Rx and Tx messages have the same intent. Whilst the Sensor Manager manages the Power, Quadrature Encoder, Rotor lock actuator and the RDS side of the interlock connector, the Release Controller manages the same type of power controller, three servos for release, lock/unlock and connector extension and the Host side of the interlock connector.

The Host Release Controller has 5 each of Rx and Tx messages. In summary Rx messages are:

1. Msg 1: Used to notify the slave I2C node of the Msg Identifier in the subsequent Tx Msg request. This allows the Slave Node to prepare the Tx buffers with the required data;
2. Msg 2: FMU Controlled State change request;
3. Msg 3: Weight off Wheels status;
4. Msg 4: RDS Functional State; and
5. Msg 5: Debug command Override. Presently this is not implemented and debug code segments were instead spliced into the code during verification activities. Further details provided within Chapter 5.

Release Controller Tx messages are:

1. Msg 0: Sensor Manager State;
2. Msg 1: Host Battery Supply Amount;
3. Msg 2: Power Mode State. Identification of Selected Power Supply Source;
4. Msg 3: Built In Test Result; and
5. Msg 4: Interlock status. These include Host Interlock Status, Debug and Received Commit to Release Interlock Status.

Three sets of discrete signals are critical to operation, safe carriage and release, refer Appendix I Table 7. These are the Host Interlock and RDS Interlock discrete and the Commit to Release. All three are active LOW so that, should the InterConnector separate causing loss of connection, (which may cause signals to float), the state of each interlock would not be seen as valid active signals. The Host Interlock discrete pins are earthed within the RDS and vice versa the RDS Interlock discrete pins are earthed within the Host RPAS. Connection of the Host to RDS through the Interconnector, mates the two interlocks for their respective Sensor Manager and Release Controller function.

The Commit to Release discrete signal originates from the RDS Sensor Manager when Sensor Manager determines that the RDS FMU and itself are in the correct state for release. Should a request to Assert the Commit to Release discrete come from the RDS FMU before the correct timeframe, the Sensor Manager causes a BIT Fail event to occur and the Release Process is halted until recovery action is undertaken. The Release Controller senses the assertion of the Commit to Release and notifies the Host FMU of overall readiness for release. Similarly, should the Release Controller sense the assertion of the Commit to Release in the wrong sequence of Release, the Release process is halted and the system must re-initialise the Release States within all four processors.

4.4.2 Software Design

To achieve the system architecture of Figure F.1 the Software Design and Implementation conforms to the State Diagrams shown within Appendix F. Each of the four processes

have independent functional allocations, though are tightly coupled to achieve RDS safe carriage and release.

The following describes the functional elements that have been completed within this project so far. These elements relate to the Sensor Manager and Host Release Controller. Appendix H contains the source listings for both units. Those software classes that are the same in both units are not repeated within the Host Release Controller Listing.

The software is based upon the Arduino sketch and associated libraries. The main programs are SensorManager.ino and HostReleaseController.ino. These sketches include in the remaining C++ classes that complete the functionality. These sketches define the structure of NilRTOS threads, semaphores and Task Control Blocks and the rates and sequence of rate based functionality at 10Hz, 5Hz and 1Hz intervals. The sketches configure the usage of interface pins and establish the I2C message buffer system. Interrupts that manage the I2C communication are also first enabled with these sketches.

The Commander class inherits the functions of StateMachine, Power and Interface classes to effect overall functional control of the unit. The Commander class contains the functionality that is executed at the 10Hz, 5Hz and 1Hz rates and includes all sequence logic necessary to achieve the State Machine defined by Figure F.5.

The Commander class implements the Quadrature Encoder speed measurement algorithm, 4.1 presented within (Petrella & Tursini 2008). This iterative algorithm uses a mix of time and frequency measurements for accurate speed measurements.

$$\Omega = \frac{\Delta N}{\Sigma T_{sc,acc} + \Sigma T_{h-1} - \Sigma T_h} \cdot \frac{60}{N_p} \quad [\text{r/min}] \quad (4.1)$$

This algorithm allows the observation window to be extended to give accurate measurements at low speed, critical to the start-up sequence of the rotor head. Importantly the one algorithm can be used for low and high speed measurements.

The time based measurements required within this algorithm are achieved through the use of the RTOS. An observation window of 100msec was selected which matches the highest task rate. As the ATmega328P does not have a division instruction, fixed point calculations were undertaken within the implementation. When first implemented, the fixed point algorithm measurements showed a consistent 4.5

The StateMachine class contains all necessary functionality to change the current functional state, including validation that the requested target state is a valid transition from the current state. StateMachine functionality is called when State changes are attempted. The State change validation process uses a network graph adjacency list, which represents all of the valid and invalid StateMachine transitions. Should an invalid state change be attempted, StateMachine functionality causes a Built in Test fault to occur.

The Interface class contains attributes and methods that manage the discrete signals to and from the Host Interconnect.

The Power class manages the Power Controller. Power methods include power supply selection, internal battery capacity measurement and current fault detection.

The BIT class undertakes and records current BIT results, either at the time of an asynchronous event like the above Invalid State change or at 1Hz. BIT results are maintained and presented to the I2C interface when changed.

The I2CBuffer class receives and prepares communication for the I2C interface. All classes that have status information to be sent to the FMU call on methods within the I2CBuffer class to format and buffer the information for transmission.

The Rotor blade conformal lock / unlock is achieved through use of methods within the ServoTimer2 library. This is an open source library which was modified slightly to integrate with the RTOS, NilRTOS.

The Release Controller uses the same set of classes to achieve its requirements, though most are modified due to alternate state sequence or interface needs. The Commander includes a software debounce algorithm to assist the action of a ground operator to attach the RDS to the Host RPAS. Three servos are now managed by the Release Controller.

4.4.3 Software Operation

On power application all platforms initialise to the Initialise State. All variables and interfaces are established to known initial conditions. For example, the Release Controller ensures the RDS suspension hook is closed and locked and the Sensor Manager Commit to Release signal is not asserted. The Release Controller, fig:RDSSStateDiagramHostRC will

then immediately transition to the PackageCheck State. The Release Controller waits within the PackageCheck State until it detects Ground operator intervention through depression of the Load switch and confirmation that the Host RPAS is on the ground via receipt of Weight on Wheels by a Host FMU message. To load an RDS onto the Host RPAS, the Release Controller must first unlock then open the suspension hook. The Release Controller then confirms RDS connection through detection of the Active LOW Host Interlock discrete and initiates closure of the suspension hook and re-locking the safety actuator. The Host FMU meanwhile, Figure F.2, will stay in the Initialise State until the Release Controller reports its transition to the PackageCheck State. The Host FMU will then transition and wait in the HRCReady State until confirmation that the RDS is connected and suspension hooks closed and locked. The Release Controller enables Host DC supply through the Interconnect and reports an RDS is connected to the Host FMU. The Host FMU then transitions to the PackageConnected State and will wait therein until the RDS is to be tasked with a Ground Target location.

Before connection to the Host RPAS the RDS local power supply is turned on via external switch. The RDS FMU and Sensor Manager initialise to the Initialise State, refer Figures F.4 and F.5 respectively. Similar to the Host FMU, the RDS FMU will remain in the Initialise State until the Sensor Manager reports transition to the Prepared State. The Sensor Manager transitions to the Prepared State on detection of the RDS Interlock Active LOW due to connection to the Host RPAS. Any subsequent loss of this RDS Interlock signal outside of the ReadyConfirm or ReadyRelease States will cause the Sensor Manager to report a BIT Failure and transition to the BITFail State.

On notification that the Sensor Manager has transitioned to the Prepared State the RDS FMU will transition to the TransferReady State and awaits Ground Target tasking transfer to start from the Host FMU.

At the culmination of transferring and validating the tasking information, the RDS and Sensor Manager will be at the Tasked and ReadyConfirm States whilst Host FMU and Release Controller will be at Tasked and PackageConnected(PowerOn) States respectively.

Within the ReadyConfirm State the Sensor Manager commands and validates the swapping the source of power supply from External to Internal and ensures the Commit to Release is not asserted. Should either action fail, the Sensor Manager will cause a BIT Fail event and its State is transitioned to BITFail. All platforms would follow suit until

the BIT Fail reason is cleared.

On receipt of the external RDS Release Command from the Operator Ground Station the Host FMU dispatches to the RDS FMU a ReadyRequest command. The Host FMU transitions to the ReadyConfirm State and waits therein until the Release Controller reports that it has transitioned to the ReleaseConsent State.

Receiving the ReadyRequest command from the Host FMU causes the RDS FMU to command the Sensor Manager to the ReadyRelease State. Given no BIT Faults exist the Sensor Manager would transition to the ReadyRelease State wherein the Commit to Release discrete is asserted to the Host Release Controller. Whilst in the ReadyRelease State the Sensor Manager continues BIT Checks at 1Hz. Detection of BIT Failure would immediately cause the Sensor Manager to transition to the BITFail State and the invalidation of the Commit to Release signal.

Successful transition of the Sensor Manager to the ReadyRelease State allows the RDS FMU to transition to its ReadyRelease State which is reported to the Host FMU and in-turn to the Release Controller. It is the independent function of the Release Controller that now confirms concurrent assertion of the Commit to Release discrete and that the RDS FMU current State is ReadyRelease. When concurrent assertion occurs, the Release Controller State transitions to ReleaseConsent. Whilst in the ReleaseConsent State the lock/unlock safety is physically removed from the suspension hook actuator. The Host FMU now transitions finally to the ReadyRelease State and awaits there until the RPAS moves to the release point. It is during this final ReleaseConsent period that the hazard of inadvertant release is now exposed due to the suspension hook safety unlocked condition.

When finally at the release point, the Host FMU transitions to the ActionRelease State and commands the Release Controller to open the RDS suspension hooks. One second later the Release Controller determines if the RDS has actually parted from the Host and determines Gone or Hung status. A Hung status will be where the RDS has not fallen away for some reason. Following Hung/Gone determination the suspension hooks are again closed and locked.

At the ReadyRelease State the Sensor Manager is polling the RDS interlock discrete at 10Hz. Detection of transition to open circuit condition, transitions the signal to HIGH due to pull-up resistance to 5V DC. This transition signifies separation from the Host RPAS

and the Sensor Manager transitions to the Separation State. Within the Separation state the Sensor Manager suspends further BIT, enables the Quadrature Encoder interrupts and initiates a one second timer to allow for a period of separation before deploying the rotor blades. The RDS FMU is notified of release via the transition of the Sensor Manager to the Separation State. On timeout of the one second timer, the Sensor Manager transitions to the Deploy State and initiates deployment of the RDS rotor blades by actuating the rotor unlock servo. The Sensor Manager measures and reports the Rotor speed until the RDS FMU commands the transition to the PowerDown state.

On detection of Sensor Manager Deploy State transition, the RDS FMU transitions immediately to its Deploy state. Its role is to manipulate the collective pitch of the rotor to spin-up the rotor in the right direction until a nominal rotor head speed is confirmed. The RDS FMU transitions to MidCourse State and navigates to the required ground task point using a balance of the kinetic energy whilst retaining sufficient energy to undertake a final flare manoeuvre. When an altitude limit is breached the RDS FMU transitions to the Flare State and initiates minimisation of vertical velocity through the flare manoeuvre before ground impact. On detection of ground impact the RDS FMU commands the Sensor Manager and itself to PowerDown.

Abort conditions are supported by the design during pre-release phases. Abort transitions the Host and RDS FMU units back to TransferReady whilst the Release Controller and Sensor Manager return to PackageConnected(Power On) and Prepared States respectively.

4.5 Critical Design Analysis

3D CAD modeling provided significant opportunities to manipulate alternate design solutions, though the learning curve to achieve simple tasks was much more extensive than originally planned. So too was the translation of a 3D model to reality. The selection of wall thicknesses, even though utilising tool functionality to calculate proposed resultant weights, caused significant weight increases when reality did not match modeling techniques. Test runs could have been attempted in order to validate the modeling and manufacturing process before final designs completed.

A major lesson to remember is that co-ordination of external manufacturing agencies takes time. Availability of material and skilled personnel and the general sequencing

of prototype development caused significant impact to moving to the next stage in this project. An issue not lost on the author as alternate plans needed to be more thought through to enable work to continue on other facets whilst the delayed facet was resolved.

The decision to position the Sensor Package above the rotor blade may have simplified the measurement of Rotor head angular velocity but it introduced significant constraints and iterations of design. The nature of testing which is described in a later Chapter on verification identified a significant flaw in the Sensor Platform and Rotor top rest design. Further, the design idea of routing wires through the spindle may seem reasonable, however, the manufacturing capabilities required to bore a 3mm hole through a 5mm rod 150mm long is significant and not considered by the Author when making the original design decision. The alternate use of mixing 5mm OD x 3mm ID carbon fibre tubing with the base of the spindle succeeded in solving that problem however introduced the weakness in the Sensor Platform / Rotor Head design.

In regards to the electrical design another flaw detected during verification was the critical decision to use the TLC4353 Ideal diode as the power controller. Though expecting to simplify the method of control and parts count, a basic flaw in design was to not take into account pass through current due to the N-Channel MOSFET body diodes when the output was disabled. Though the Power Controller software interface expected power to be removed, the body diodes allowed power to remain present causing verification failures and the basic inability to turn off supply when commanded. A more appropriate alternate would be the use of P-Channel and N-Channel MOSFET load switches with minimal additional circuitry to provide the BIT status functionally the original design requires.

Developing and co-ordinating the manufacturing of the printed circuit board (pcb) designs provided the ability to tailor the circuit specifically to the need both in size and functionality. The process of pcb manufacturing can lead to implementation errors that are difficult to detect until the completed pcb is returned. The Author recognises that a detailed review of pcb design tool reports could have assisted detecting the common power rail anomalies that required circuit repair after the completion of stuffing the electronic components onto the board.

Achieving successful software design and development, within a finite agreed schedule, is a constant learning lesson for the Author. At the onset of this project, the considered

timing to achieve software development for all units was thought to be achievable. Life has many ways to intervene and the most appropriate action is to re-plan and progress with realistic goals. The two units, Sensor Manager and Release Controller, completed in this project's timeframe have in the most part been verified to achieve the established System Requirements. There is realistically a significant amount of work to complete the overall project though.

Within the detail of the I2CBuffer logic, although the implementation uses synchronisation flags to manage validity of data between the FMU and Sensor Manager or Release Controller, a late anomaly recognised during verification was the need for double buffering of this message data to eliminate race conditions. A new software class, MsgBuff, included within Appendix H was written using the double buffer algorithm identified within (Huang, Pillai & Shin 2002), however insufficient time was available to incorporate the class through inheritance within the I2CBuffer class.

4.6 Chapter Summary

A prototype Rotary Deceleration System has been designed and manufactured using plastic 3D printing and machined metal. The resultant design does not yet satisfy all the established System Requirements, though this chapter disclosed the software design of the four units (Host FMU, RDS FMU, Sensor Manager and Release Controller) that could satisfy safe carriage and release of the RDS at the required release point. Finally, a critical analysis was undertaken discussing the identified flaws in the design and implementation processes and for some, possible corrections.

Chapter 5

Verification

5.1 Chapter Overview

This chapter describes the development of a ground test facility useful to undertake dynamic lift and flare verification activities. It further describes the minimal set of system tests that verify a number of System Requirements. Finally a short critical analysis of the test facility design is undertaken.

5.2 Verification Test Facilities

A test facility has been designed, based upon (Slaymaker & Gray 1953) and built to provide for prototype verification activities. The design allows capability that can impart an adjustable angular velocity to the rotor head and to allow autorotation tests within the vertical dimension, being constrained laterally.

As defined within the preliminary design section the rig is to be attached to a building wall to provide overall rigidity.

As access to a wind tunnel of sufficient size was not feasible during this project, the overall intention was to utilise this test rig to determine lift versus pitch and flare characteristics in an informal manner. The design layout is seen within Figure 5.1 for the lift testing.

The higher flare tests were to extend the facility as in Figure 5.2. Figure 5.3 shows the

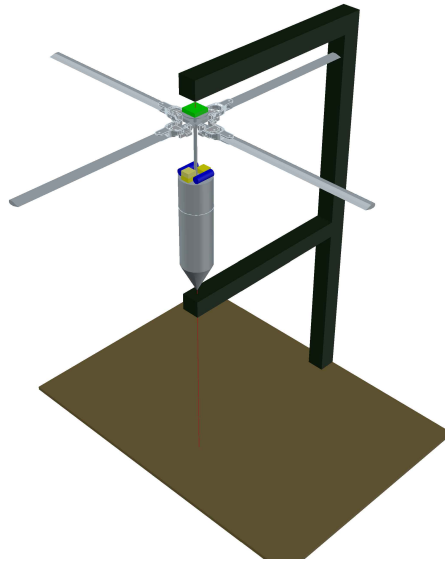


Figure 5.1: Ground Test Lift Analysis Rig

completed test facility during a test run.

The Test rig design includes:

1. A vertical spine, adjustable using three 2.1m segments,
2. A top fixed horizontal drive and guide wire attachment beam,
3. A lower horizontal support beam, adjustable in the vertical plane from the base up to the upper beam,
4. A base, and
5. Vertical guide wire between the upper support beam and base that passes through the RDS and lower support beam.

The vertical spine, upper and lower beams are made from tubular aluminium 50x50mm. The vertical spine is built up in three segments each 2.1m, whilst the two horizontal beams are of sufficient length to maintain the RDS at a distance allowing for complete rotation clearance of the RDS rotor blades. At the top of the upper segment, braces are included to attach the top horizontal support / drive beam and to provide for lateral stabilisation

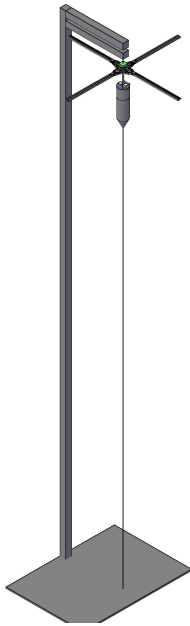


Figure 5.2: Ground Test Lift Analysis Rig

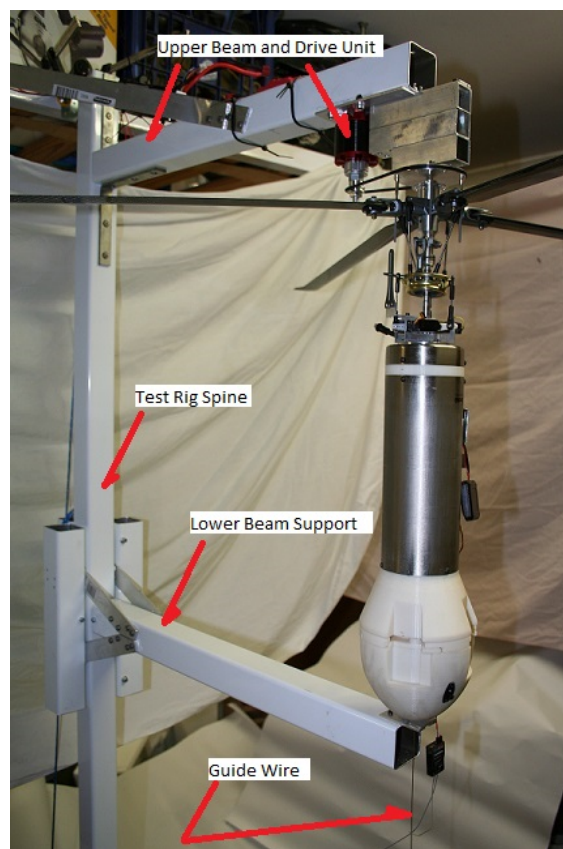


Figure 5.3: Completed Ground Test Lift Analysis Rig, Ian Saxby 2014

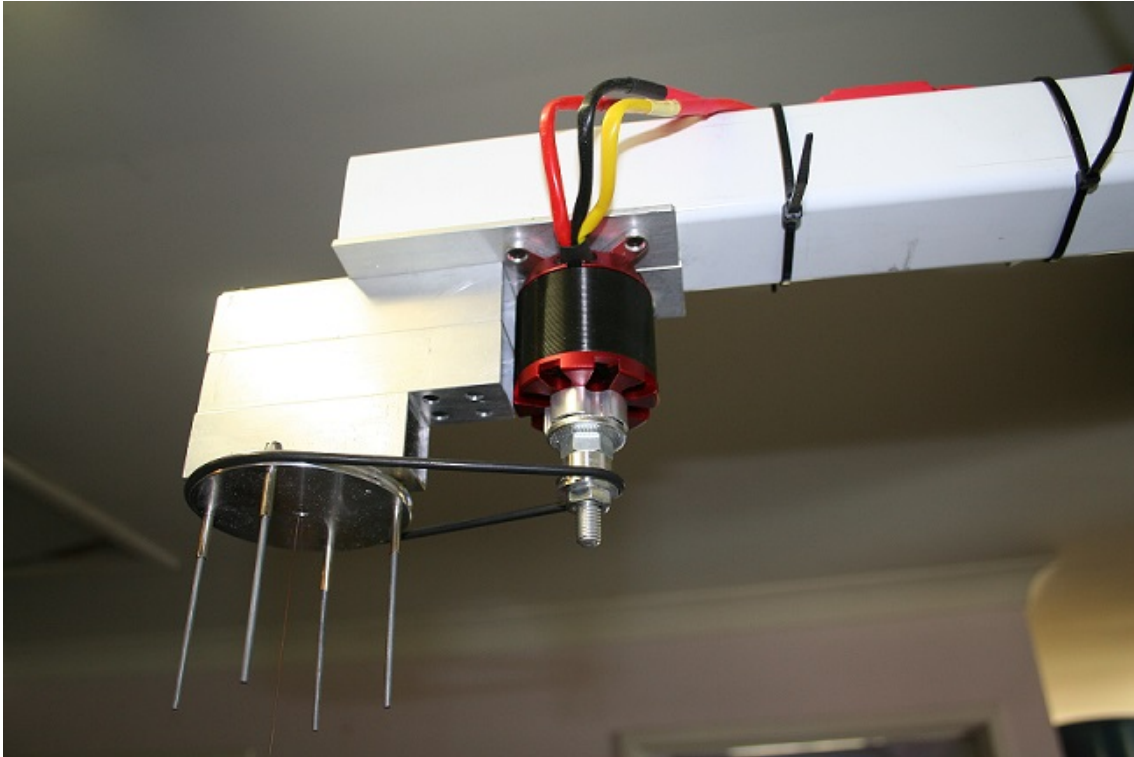


Figure 5.4: Upper Beam showing pre-load drive unit, Ian Saxby 2014

of the spine against a building wall. The adjustable height of the rig is setup depending on the use of the facility. The spine is designed to be supported at the base and upper end to allow the lower horizontal support beam to travel the full extent of the spine during the autorotation test cases. The extreme top of the rig spine is attached to guy ropes similar to radio mast fixtures to fix the rig vertically.

The top horizontal drive beam, includes the drive battery, Electronic Speed Controller (ESC), A Redback 91 size Brushless DC (BLDC) motor, pre-load rotary drive mechanism and Guide wire attachment point. When required the BLDC spins the pre-load rotary drive through a belted drive train at a ratio of 5:1. The pre-load rotary drive, as shown in Figure 5.4, uses a series of four metal wire fingers to impart the rotary motion onto the rotor head from the drive motor. The expectation is that the drive fingers allows for small vertical movements of the RDS during lift tests, and unimpeded downward movement during autorotation tests once the lower support beam is dropped away.

The Guide wire is passed up through the centre of the rotary drive mechanism and held centred to the rotary bearing, Figure 5.5.

The lower horizontal support beam holds the weight of the RDS during test activities,



Figure 5.5: Drive Unit Engagement fingers and Guide wire, Ian Saxby 2014

Figure 5.6. This support beam is restrained to the vertical spine by a system of wheels and bearings and is vertically adjustable via the use of rope and pulley system tied off at the base. A final expectation of the design was to include a load cell on the lower support beam to measure impacts of rotor head speed and collective pitch settings, however, this portion of the design was not completed.

Finally, the rig has a base that positions the vertical spine and includes a pulley and rope tie off cleat that holds the lower horizontal support beam at any desired height, Figure 5.3.

At the shortest length of 2.1m the test rig is used for lift and collective pitch adjustment testing. When using additional segments (giving heights 4.2 to 6.3m) the rig can be used to undertake Autorotation flare tests.

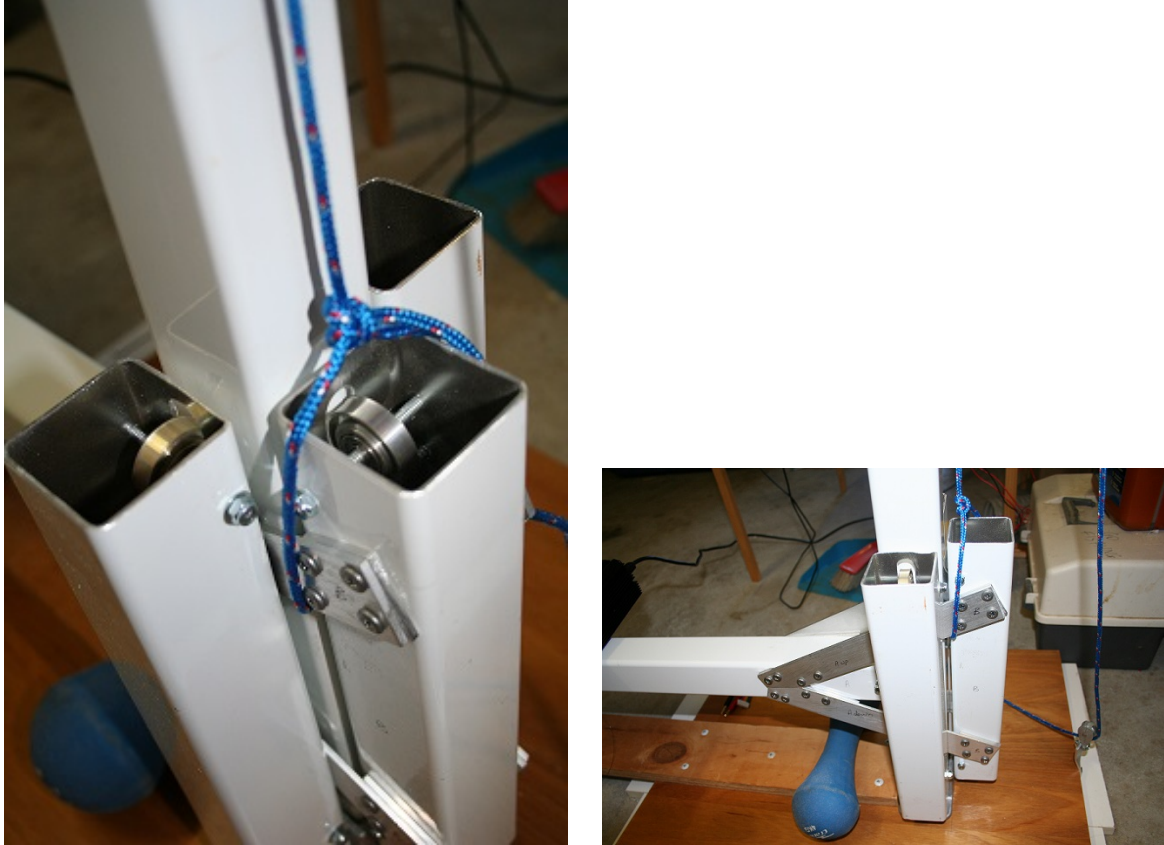


Figure 5.6: (a)Lower Beam Back View. (b)Lower Beam Side View.

5.3 Verification Activities

The verification of Release Controller and Sensor Manager software functionality was achieved through stub tests of functions. Integrated testing of the overall State Machine was achieved by splicing debug code into the 1Hz rate. This debug code initiated state transition sequence changes allowing verification of the interlock discrete signal transitions. This debug functionality allowed initiation of both normal and off-normal state sequence transitions at defined time steps. The time steps and target State is modified on multiplies of the 1Hz tick and provides verification evidence of a combination of timed automated or manual discrete input test inputs. Using this test sequence methodology allowed rotor angular velocity and rotor lock / unlock Servo control verification.

An example of the code sequence is shown in Listing 5.1. The sequence transitions through the Sensor Manager normal state machine sequence at the 4, 8 and 60 second marks. The gap between 8 and 60 seconds allows for test operator intervention to remove the RDS Interconnect discrete to confirm transition to the Deploy and subsequent states.

Listing 5.1: The Sensor Manager debug sequence.

```
if(RetrieveState() == PREPARED)
{
    if(bDebugFirstPass)
    {
        bDebugFirstPass = false;
        Count1Hz = 0;
    }
}
if(RetrieveState() >= PREPARED)
{
    Count1Hz++;
    if (Count1Hz == 4)
    {
        if(RetrieveState() == PREPARED)
        {
            SetState(READYCONFIRM);
            bDebugFirstPass = true;
        }
        else
            Count1Hz = 3;
    }
    if (Count1Hz == 8)
    {
        SetState(READYRELEASE);
        bDebugFirstPass = true;
    }
    if (Count1Hz == 60)
    {
        if(RetrieveState() == DEPLOY)
        {
            SetState(POWERDOWN);
            bDebugFirstPass = true;
        }
        else
        {
            Count1Hz = 10;
        }
    }
    if (Count1Hz > 61)
    {
        bDebugFirstPass = true;
        Count1Hz = 61;
    }
}
else
    Count1Hz = 0;
```


The wiring loom interconnectivity was verified by pin to pin resistance checks using a multimeter. Once all pins connections were verified power checks were undertaken without the power controller or FMU / Arduino boards connected. Next the power supply, power controller and Arduino boards were connected and the debug 1Hz sequence of State transitions executed to confirm internal and external power supply commands and resultant expected status information.

The Quadrature Encoder algorithm and associated quadrature sensors were tested using an combination of test input square wave and Salae Logic Analyser to confirm timing of input signal compared to output rotor angular velocity. A constant 4.5

5.4 Critical Analysis of Verification Facilities

In implementing the test rig a number of improvements are considered necessary. The first being to incorporate a method of measuring the drive mechanism angular velocity. This would eliminate the need for a hand held speed measuring device which necessitated the physical aiming of the speed detection device at a close proximity to the spinning rotor blades. Further, inclusion of automation would enable recording of velocities to allow comparative validation of the angular velocities detected by the RDS.

The second improvement relates to removing the bowing induced into the rig spine due to guide wire tension. For example, in using a minimum height 2.1m spine, when applying sufficient tension onto the guide wire the spine and top fixed horizontal drive beam bend enough so as the guide wire no longer passed central to the rotor drive bearing as shown in Figure 5.5. The effect of this is that the guide wire can be abraded or a lateral oscillation can be induced into the guide wire and therefore the RDS during tests. This lateral effect is quite significant and is further exacerbated by the third area for improvement below.

The third improvement focuses on the rotor pre-load drive mechanism. The design outcome included a 5mm separation between the inside of the drive fingers and the extremity of the RDS Sensor Assembly, refer Figure 5.4.

This gap was expected to be sufficient to allow minor vibration / lateral oscillations of the RDS during pre-load drive use. In undertaking testing it was noted that due to unbalanced rotor blades or the lateral movements caused by the bow discussed above,

the lateral space to the drive fingers is insufficient and the fingers impact the Sensor Assembly. This impact causes a dragging induced rotation force to the Sensor Assembly around the rotor axis. As experienced in the testing phase, this induced rotation force imparts excessive forces through the grub screws to the carbon fibre rotor spindle. This force was of such magnitude that the carbon fibre was damaged and the assembly was able to rotate. The immediate and serious impact due to this failure mode is that the Quadrature Encoder loom is twisted around the guide wire sheath to the point of failure in a matter of seconds. Additionally, the rotor head was then also free to travel upwards off the axle. There was insufficient time to re-loom to continue testing with the Quadrature Encoder so the platform was removed for further tests.

5.5 Chapter Summary

This chapter summaries the development and resultant manufacturing of a ground based test rig. The chapter identified the verification testing that has been undertaken within the project and also critical corrections necessary for further use.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

This project attempted to design and prototype a device that can control and decelerate a package using an active autorotation technique from the rotary helicopter environment. The project outcomes satisfied many of the challenging design issues though many remain.

The project provided the Author an insight into the 3D CAD modeling tools available and the experience of translating the resultant hardware physical design into reality through 3D printing and metal lathe work. The software for the two independent controller units is mature though additional work is still required to complete implementation and verification.

The project closes with a strong design of the physical components required to allow the deployment of conformally constrained rotor blades and the Software State Machine that satisfies many of causes that underpins the identified system safety hazard.

6.2 Achievement of Project Objectives

The following objectives have been fully or partially addressed:

1. Research rotorcraft theory of flight.

2. Research Navigational theory, including intersystem navigational alignment.
3. Identify the applicable Australian Aviation Regulatory regime implications and account for such within subsequent design.
4. Design and construct the mechanical, electronic and software components required to deliver the rescue package.
5. Design and construct a ground test capability to validate the proposed rescue package design.

Subordinate objectives achieved included undertaking the project development considering a safe development environment and production of a safe product.

The following objectives were either not addressed or not fully addressed:

1. Research rotorcraft mathematical modelling of flight mechanics.
2. Construct a mathematical model for autorotation considering the constrained rescue package flight path.
3. Execute ground based testing and evaluate the resultant test outcomes.

6.3 Further Work

There remains a significant amount of work in this project from completing the Flight Management aspects of the design State Machines through to correcting the design flaws in the physical, electrical and software areas. The flare modeling and translation to a core function within the RDS FMU code is also required. The complete source code developed so far is included within this dissertation should any further development be undertaken in the future.

References

- AIA (2003), *An Experimental analysis of chamber effects of a 6-bladed flapped autorotational aerodynamic decelerator*, number 2003 in ‘2143’, American Institute of Aeronautics and Astronautics.
- Aircraft, S.-., Dev, S. & Committee, S. A. (1996), *Guidelines and Methods for Conducting the Safety Assessment process on Civil Airborne Systems and Equipment*, SAE International.
- Alex (2007), ‘2.9 – hungarian notation’.
- CASA (2002), *CASR 101 - Unmanned aircraft and rocket operations*, Civil viation Safety Authority.
- CASA (2014), *NPRM 1309OS - Remotely Piloted Aircraft Systems*, Civil viation Safety Authority.
- de la Cierva, J. & Ray, J. G. (1931), *Wings of tomorrow: The Story of the Autogiro*, Brewer Warren and Putnam.
- Dupr, N., Martin, J.-F., Degryse, J., Fernandez, V., Soudan, P. & Guyomard, D. (2010), ‘Aging of the lifepo4 positive electrode interface in electrolyte’, *Journal of Power Sources* **195**(21), 7415 – 7425.
- Grewal, M., Weill, L., & Andrews, A. (2007), *Global Positioning Systems, Inertial Navigation, and Integration*, second edition edn, Wiley.
- Groves, P. D. (2003), ‘Optimising the transfer alignment of weapon ins’, *The journal of Navigation* **56**(02), 323–335.
- Huang, H., Pillai, P. & Shin, K. G. (2002), ‘Improving wait-free algorithms for interprocess communication in embedded real-time systems’, *Ann Arbor* **1001**, 48109–2122.

- Johnson, W. (1977), Helicopter optimal descent and landing after power loss, TM 73244, Ames Research Center, National Aeronautics and Space Administration.
- Lambermont, P. M. & Pirie, A. (1959), *Helicopters and Autogyros of the World*, Philosophical Library.
- Petrella, R. & Tursini, M. (2008), ‘An embedded system for position and speed measurement adopting incremental encoders’, *Industry Applications IEEE Transactions on* **44**(5), 1436–1444.
- Slaymaker, S. E. & Gray, R. B. (1953), Power-off flare-up tests of a model helicopter rotor in vertical autorotation, Technical Note 2870, National Advisory Committee for Aeronautics.
- States, U. (1993), *RTCA, Inc., Document RTCA/DO-178B [electronic resource]*, U.S. Dept. of Transportation, Federal Aviation Administration [Washington, D.C.].
- Wernicke, R. (1959), Preliminary tests of model spacecraft rotor landing system, Technical report, Bell Helicopter Corporation.

Appendix A

Project Specification

Project Specification

For: **IAN SAXBY**

Topic: REGULATING RESCUE PACKAGE DESCENT THROUGH CONTROLLED
AUTOROTATION

Supervisors: Mark Phythian

Sponsorship: Faculty of Health, Engineering & Sciences

Project Aim: To design, construct and demonstrate the safe carriage,
flight and delivery through rotorcraft autorotation of a res-
cue package from an air vehicle.

Program:

1. Research rotorcraft theory of flight.
2. Research rotorcraft mathematical modelling of flight mechanics.
3. Research Navigational theory, including intersystem navigational alignment.
4. Identify the applicable Australian Aviation Regulatory regime implications and account for such within subsequent design.
5. Construct a mathematical model for autorotation considering the constrained rescue package flight path.
6. Design and construct the mechanical, electronic and software components required to deliver the rescue package.
7. Design and construct a ground test capability to validate the proposed rescue package design.
8. Execute ground based testing and evaluate the resultant test outcomes.

As time and resources permit:

1. Demonstrate, through integration testing, the maturity of design through captive carriage testing.
2. Demonstrate safe delivery of the rescue package given a dynamic flight environment

Agreed:

Student Name: Ian Saxby

Date: 19 Mar 14

Supervisor Name: Mark Pythian

Date: 19 Mar 14

Appendix B

System Safety

B.1 Appendix Introduction

This Appendix provides a breakdown of Safety Releated Hazard Lists with associated Fault Tree Analyses and Safety Verification Matrix.

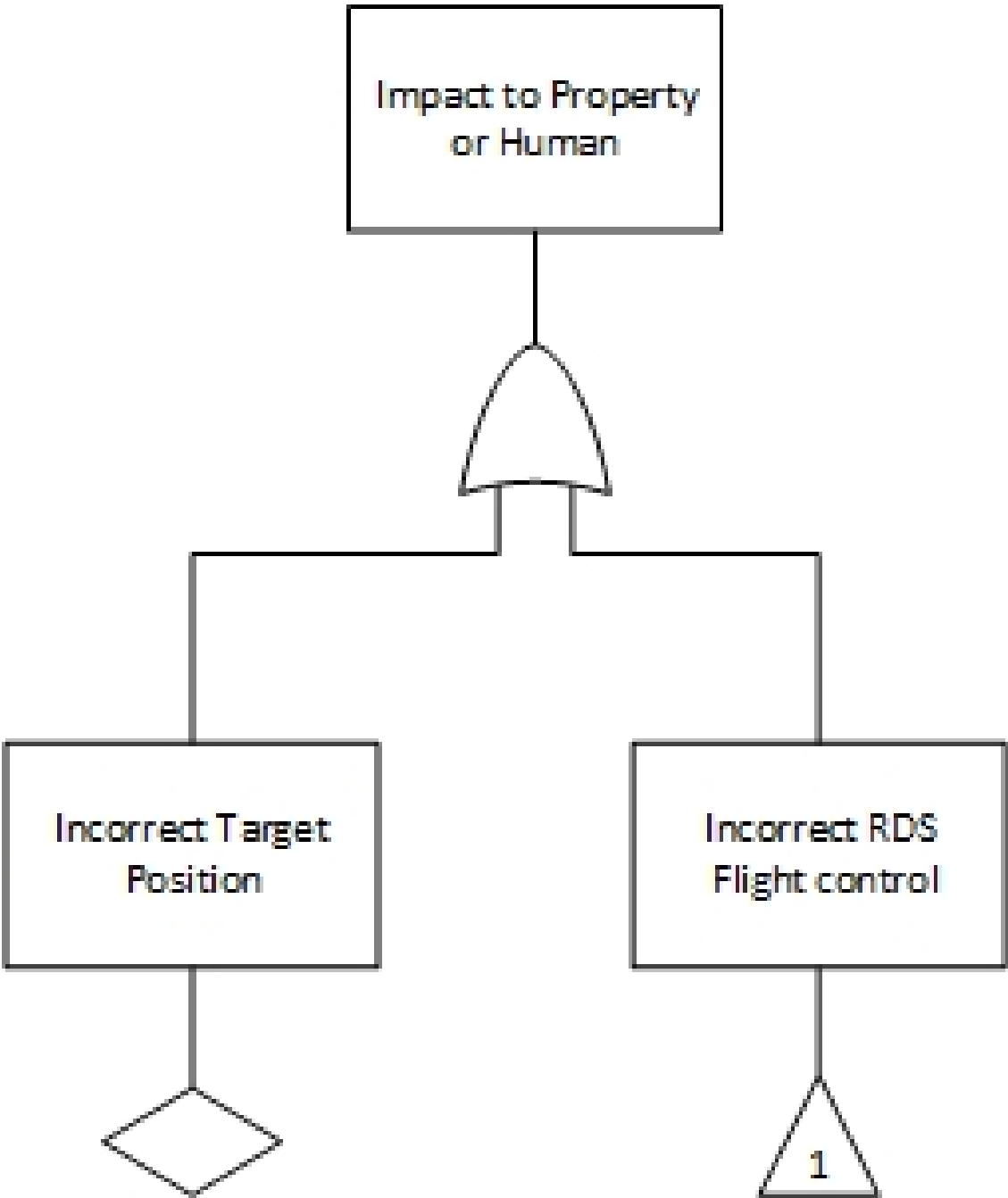


Figure B.1: Impact to Property or Human

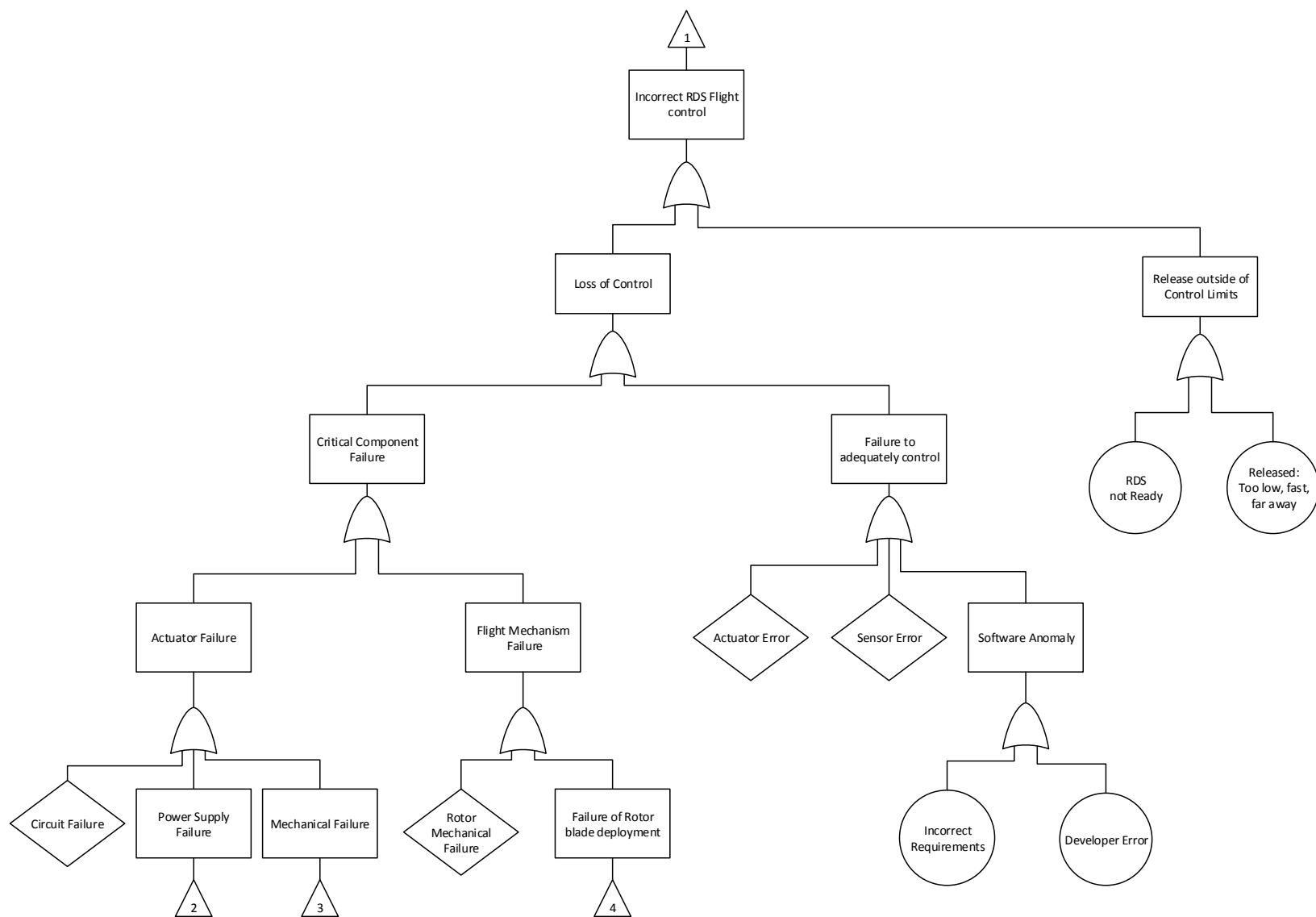


Figure B.2: Loss of Control

B.2 Safety Requirements Verification Matrix

Table B.1: System Safety Traceability Matrix

System Safety Re- quirement	Allocation	Design In- corporated	Implemented	Verified
SSR1	Host & RDS FMU	Yes	No	No
SSR2	Host & RDS FMU	Yes	No	No
SSR3	Host RC & RDS SM	Yes	Yes	Yes
SSR4	RDS SM	Yes	Yes	No
SSR5	RDS Rotor Assembly	Yes	Yes	Partial
SSR6	RDS FMU & SM	Yes	FMU open- source SM Yes	FMU No SM Yes
SSR7	RDS SM	Yes	Invalid State Yes Comm Error No	Invalid State -Yes

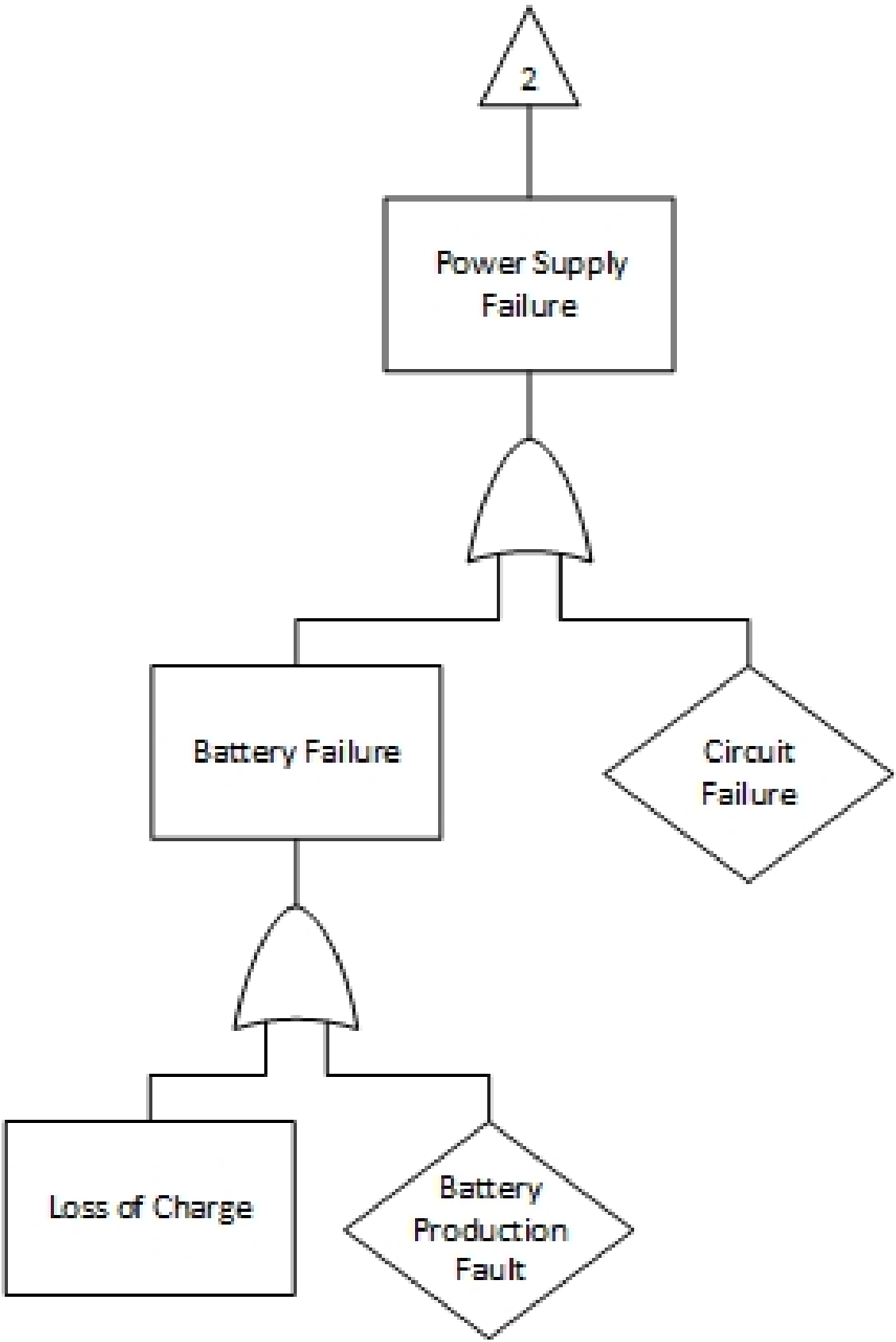


Figure B.3: Power Failure

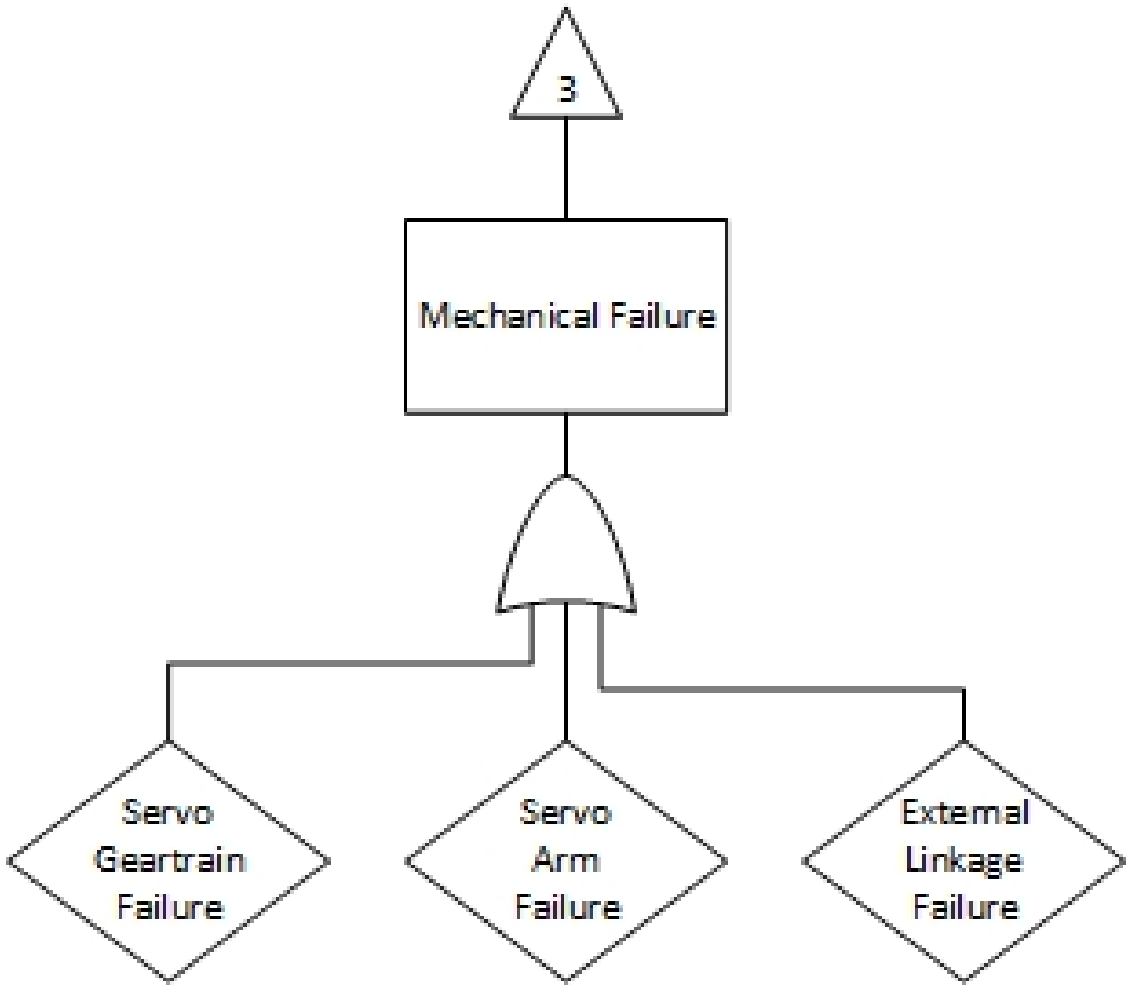


Figure B.4: Mechanical Failure

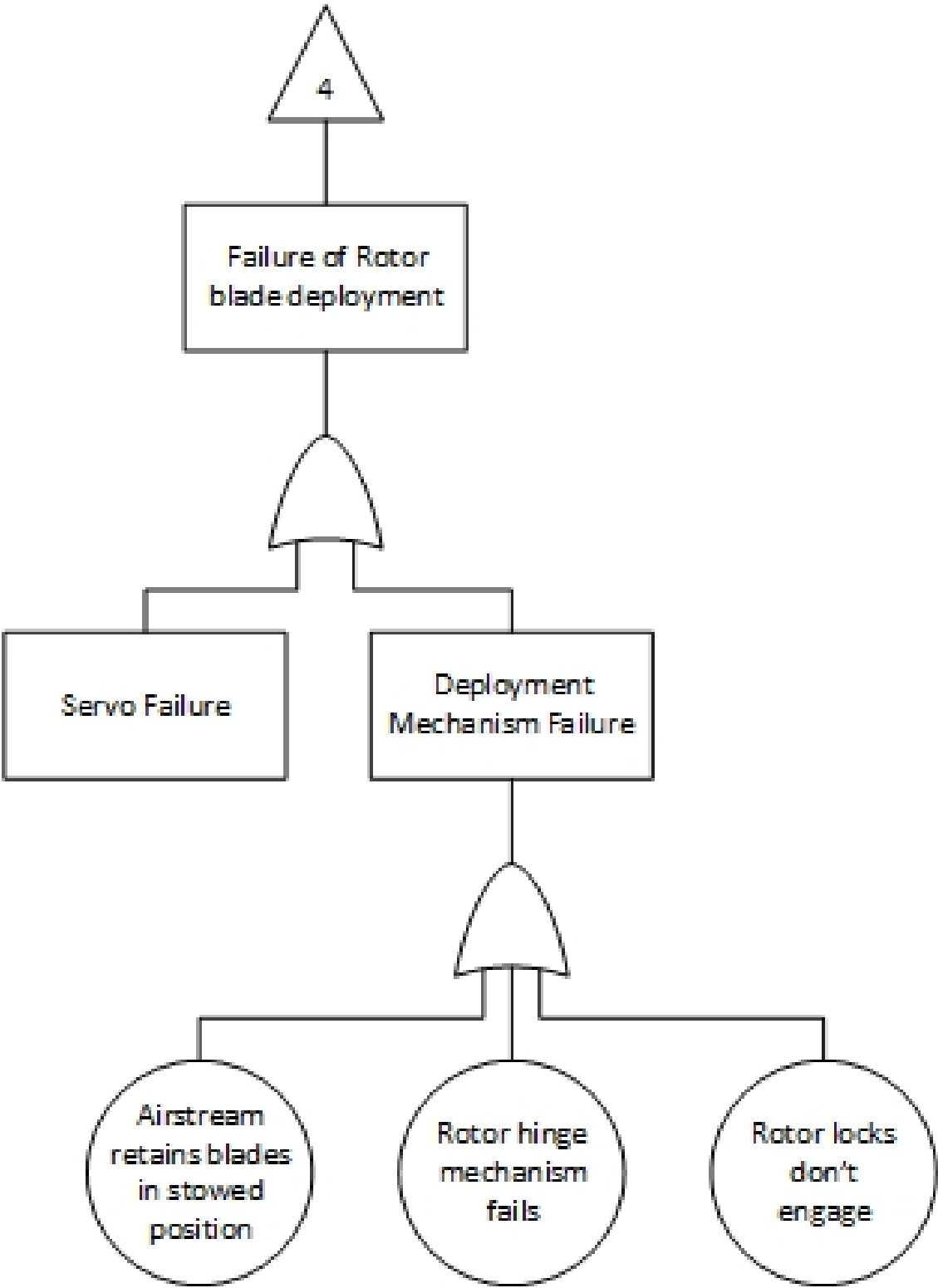


Figure B.5: Deployment Failure

Appendix C

System Requirements and Architecture

C.1 Appendix Introduction

This Appendix provides the overall System Requirements established for the prototype development. Each System Requirement is identified with a Unique Identifier starting with an R. concatenated with an increasing sequence number, (e.g. R.1). Where the System Requirement is safety related the Safety Requirement Number is also appended against the System Requirement Unique Identifier, e.g. R.3(SSR1).

C.2 System Requirements

C.2.1 Navigation

R.1 The System shall be capable of detection or calculation of Roll, Pitch and Yaw angular position, velocity and acceleration.

R.2 The System shall be capable of commanded flight / navigation to a given ground position.

R.3(SSR1) The System shall validate the specified target co-ordinates have been correctly passed to the RDS prior to release.

R.4(SSR2) The System shall ensure the RDS release conditions are within defined limits prior to release.

R.5(SSR3) The System shall ensure that the RDS is in a functional state appropriate for release.

C.2.2 Power Supply

R.6 The System shall include a controllable internal power supply.

R.7 The system shall be capable of receiving power from either external or internal sources when connected to the host Unmanned Aerial System (UAS).

R.8 The System powercontroller shall be capable of selecting between internal and external

power sources.

R.9(SSR4) The System shall verify RDS power supply availability and remaining capacity are within correct boundaries prior to authorising release.

C.2.3 Physical

R.10 The System shall incorporate commercially available model helicopter components, swashplate, head design and rotor blades.

R.11 The System shall include the capability to detect the direction of rotor head rotation.

R.12 The System shall include the capability to detect rotor head angular velocity within the range of 0 to 2500 rev/min.

R.13 The System shall incorporate a folding mechanism to allow conformal stowage of rotor blades pre deployment.

R.14 The System shall incorporate a rotor blade lock mechanism to retain the rotor blades conformal to the sub-system body until deployed during the descent phase.

R.15 The System shall be capable of flight with a 500ml water container as payload.

R.16 The System shall incorporate a release mechanism on the host UAS to secure the deployable sub-system.

R.17 The host UAS release mechanism shall incorporate a secondary lock mechanism that must be disengaged to allow release to occur.

R.18(SSR5) The Rotor Deployment mechanism shall force the rotor blades into the airstream following release.

C.2.4 Interface

R.19 The System shall be capable of detecting connection or loss of connection between the host UAS and deployable sub-system.

R.20 The System shall communicate using UART TX/RX between the Host UAS and deployable sub-system Flight Management Systems during carriage.

R.21 The System shall incorporate an independent discrete response indicating deployable sub-system readiness for release.

C.2.5 Built In Test

R.22(SSR6) The System shall carry out periodic Built in Test (BIT) functions on guidance, control and navigation sensors prior to authorising release.

1. State Sensors; GPS, Rate Gyros, Accelerometers,
2. Incorrect Power Supply states,
3. Power Supply current limit violations, and
4. Inadvertent release.

R.23(SSR7) The system shall include failure-detection logic and self-check software to confirm correct operation prior to authorising release.

1. Invalid Software State changes, and
2. Communication errors.

C.2.6 Ground Test Facility

R.24 The System shall include a Ground Test Subsystem.

R.25 The Ground Test Subsystem shall be capable imparting an adjustable angular velocity to the rotor head and to allow autorotation tests within the vertical dimension, whilst constrained laterally.

Appendix D

RDS Mechanical Drawings

RDS mechanical drawings are included within this Appendix. Where the items were 3D printed the dimensioning of the associated mechanical drawings are reduced in complexity or removed for brevity.

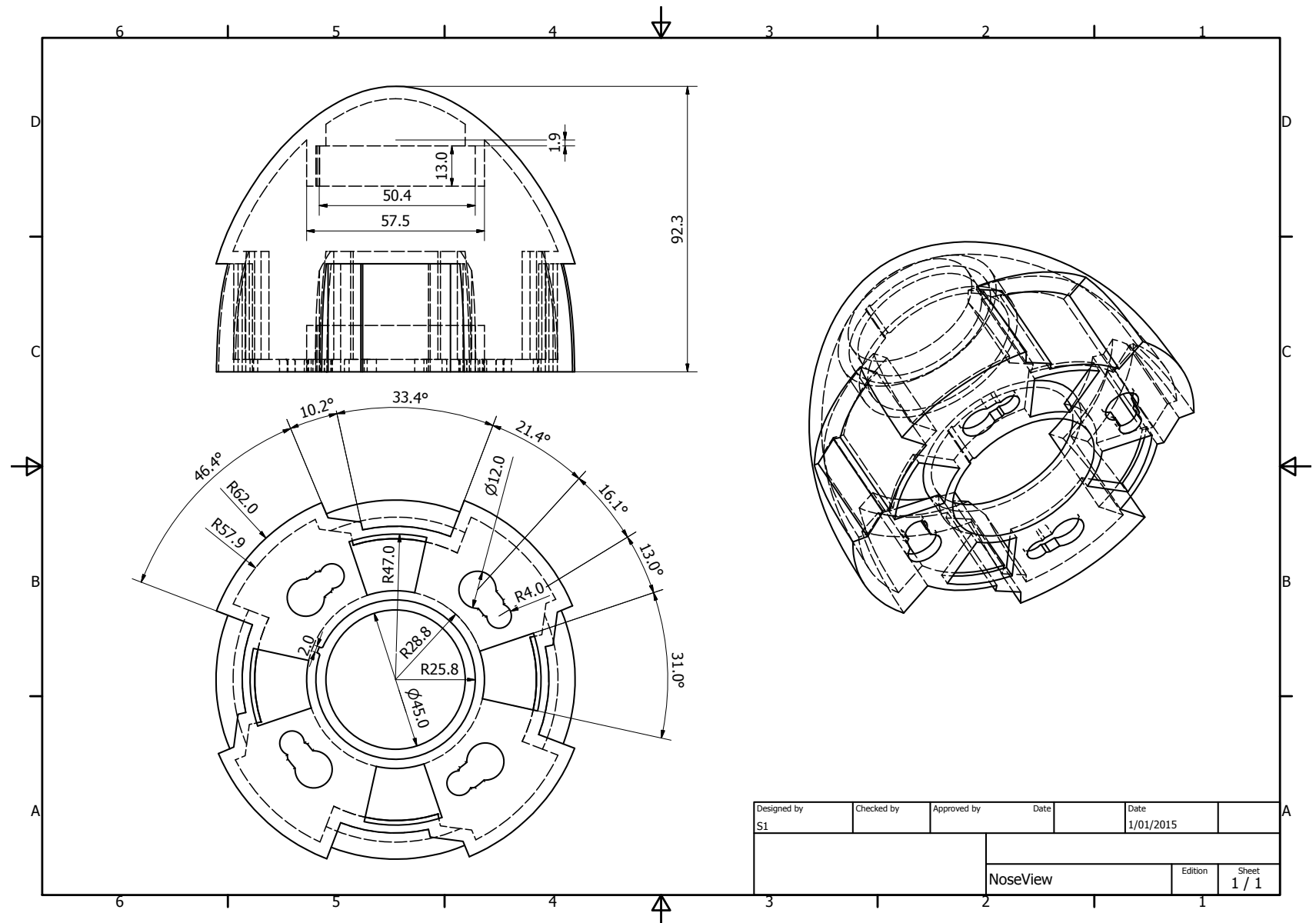


Figure D.1: Nose - 3D Printed

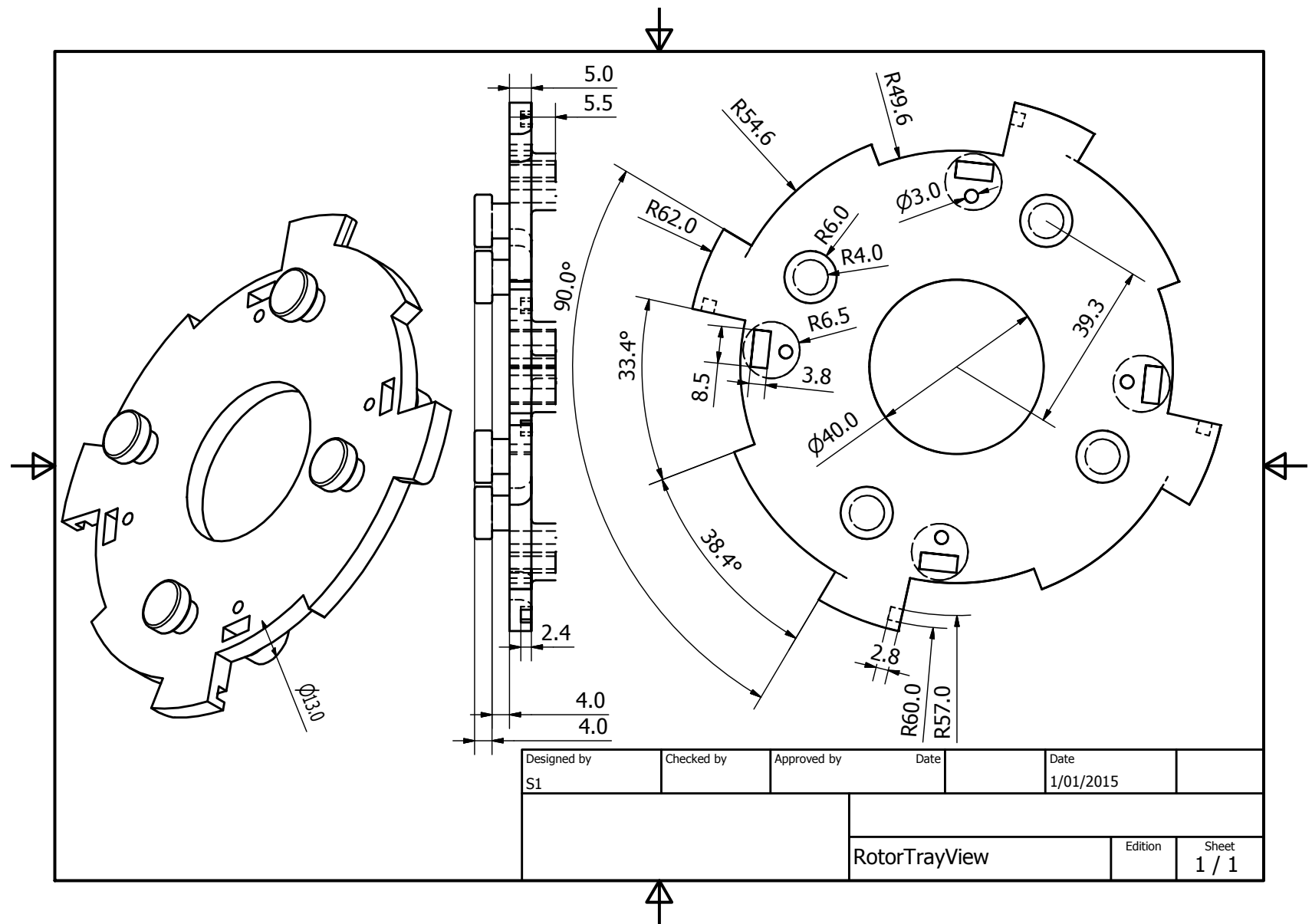


Figure D.2: Rotor Lock Tray - 3D Printed

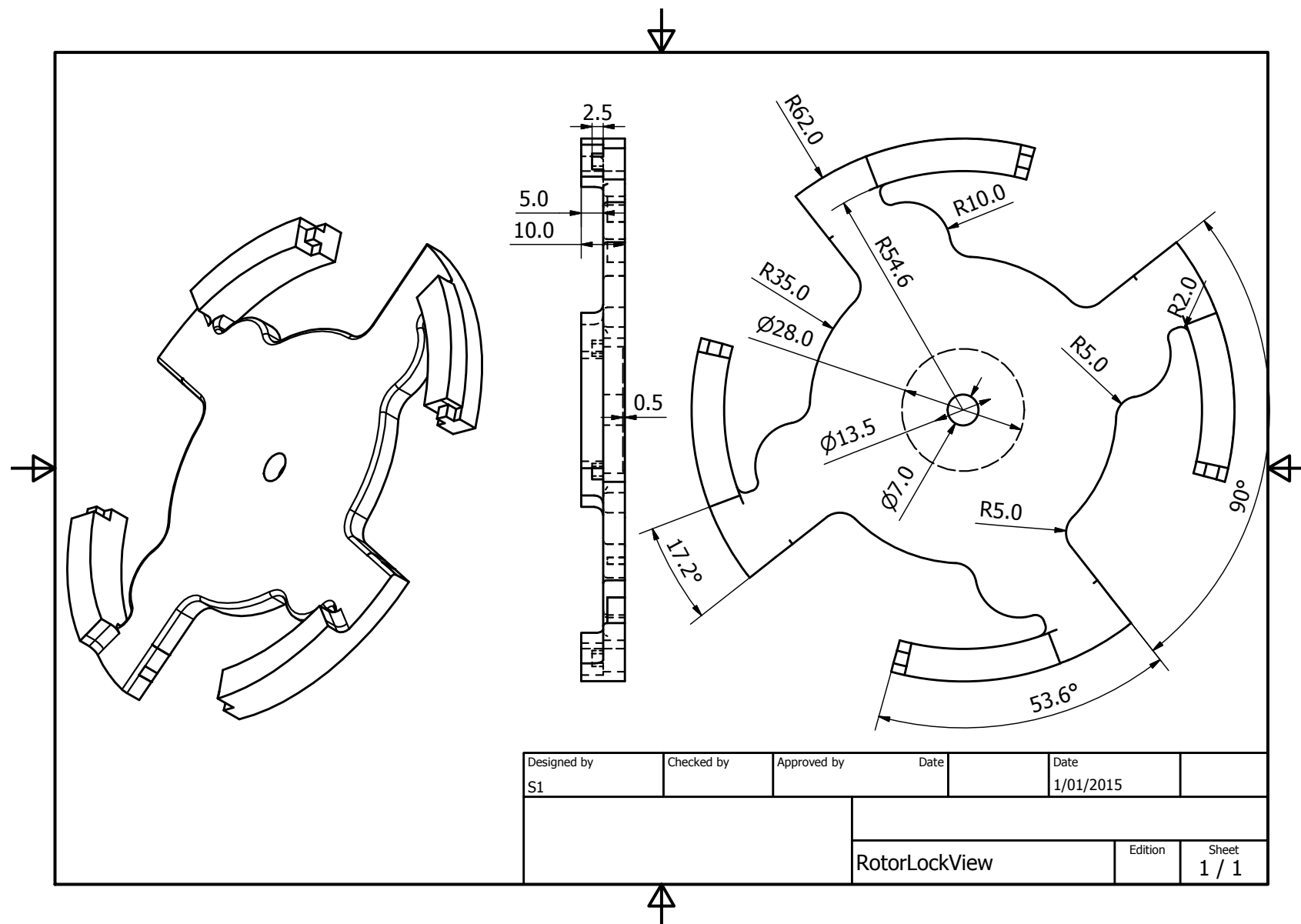


Figure D.3: Rotor Lock - 3D Printed

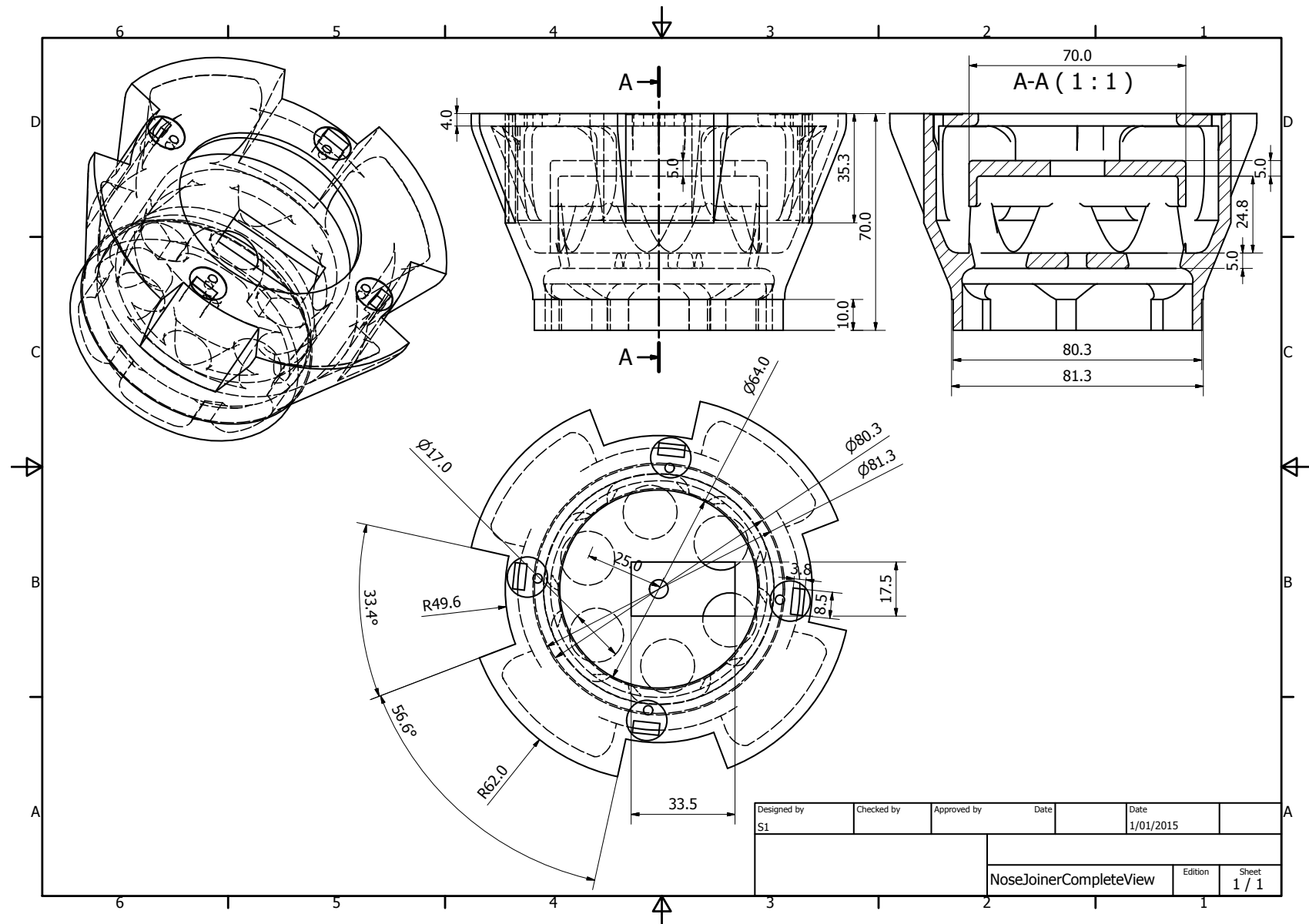


Figure D.4: Nose Joiner - 3D Printed

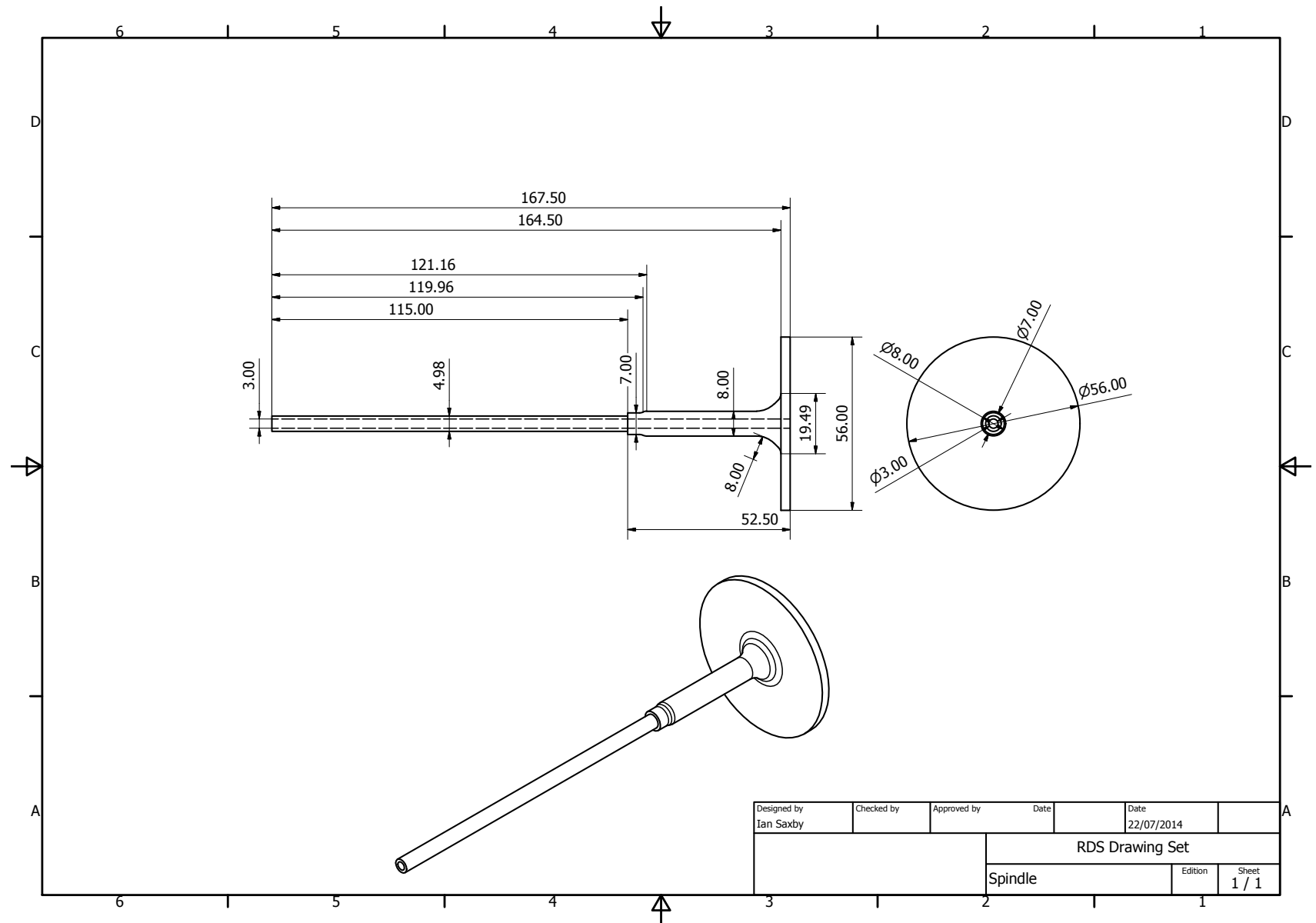


Figure D.5: RDS Spindle

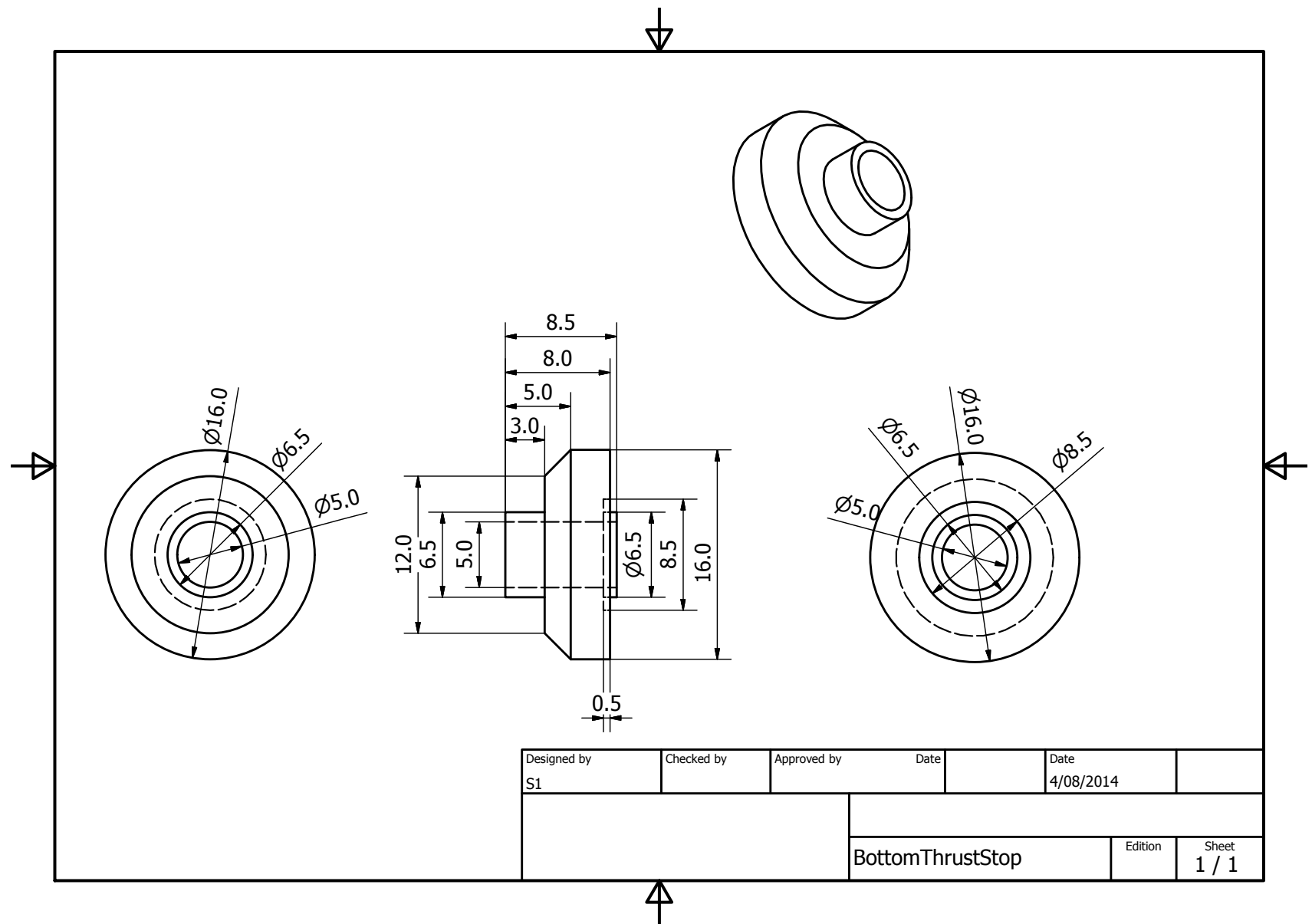


Figure D.6: Bottom Thrust Stop

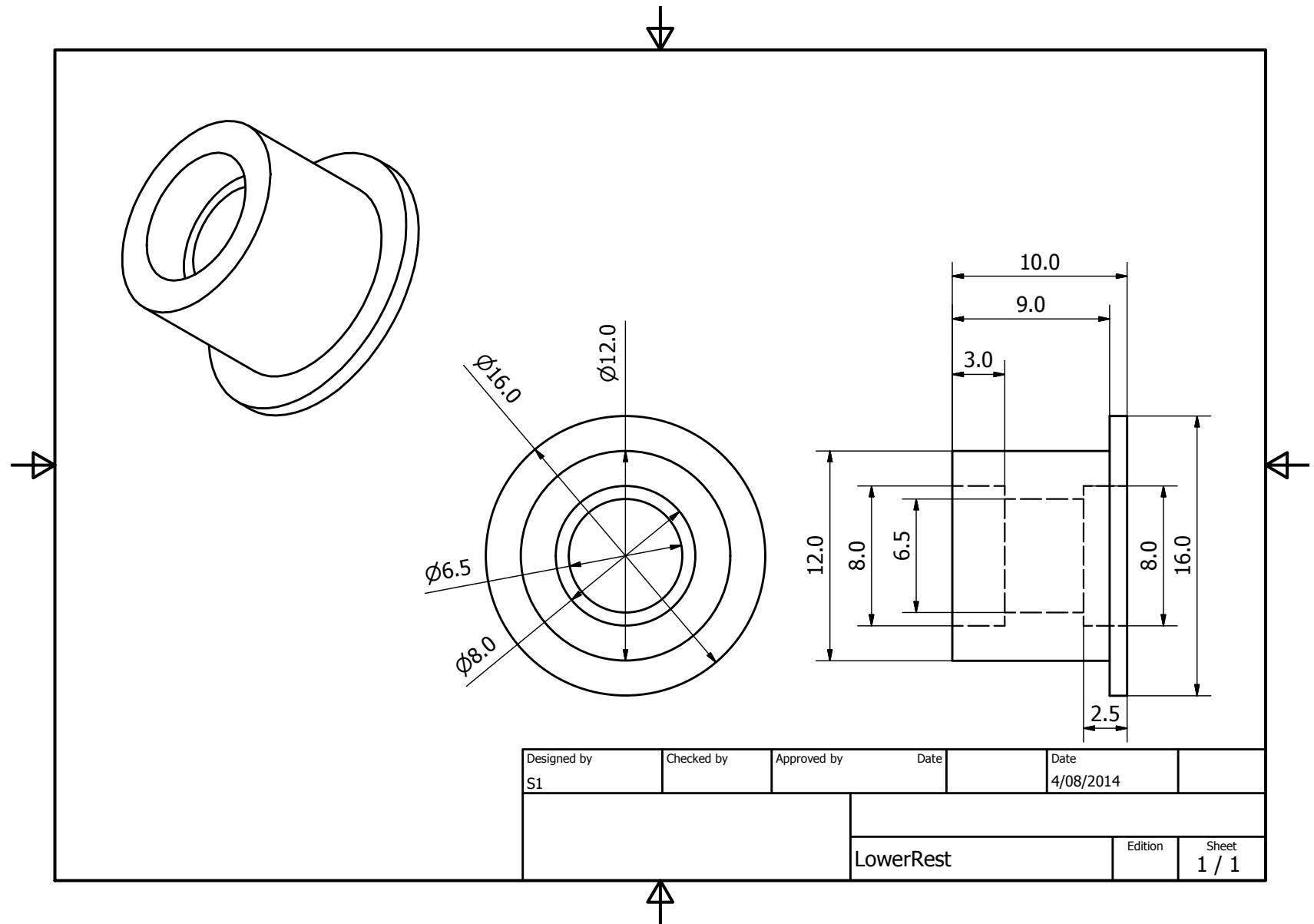


Figure D.7: Lower Rest

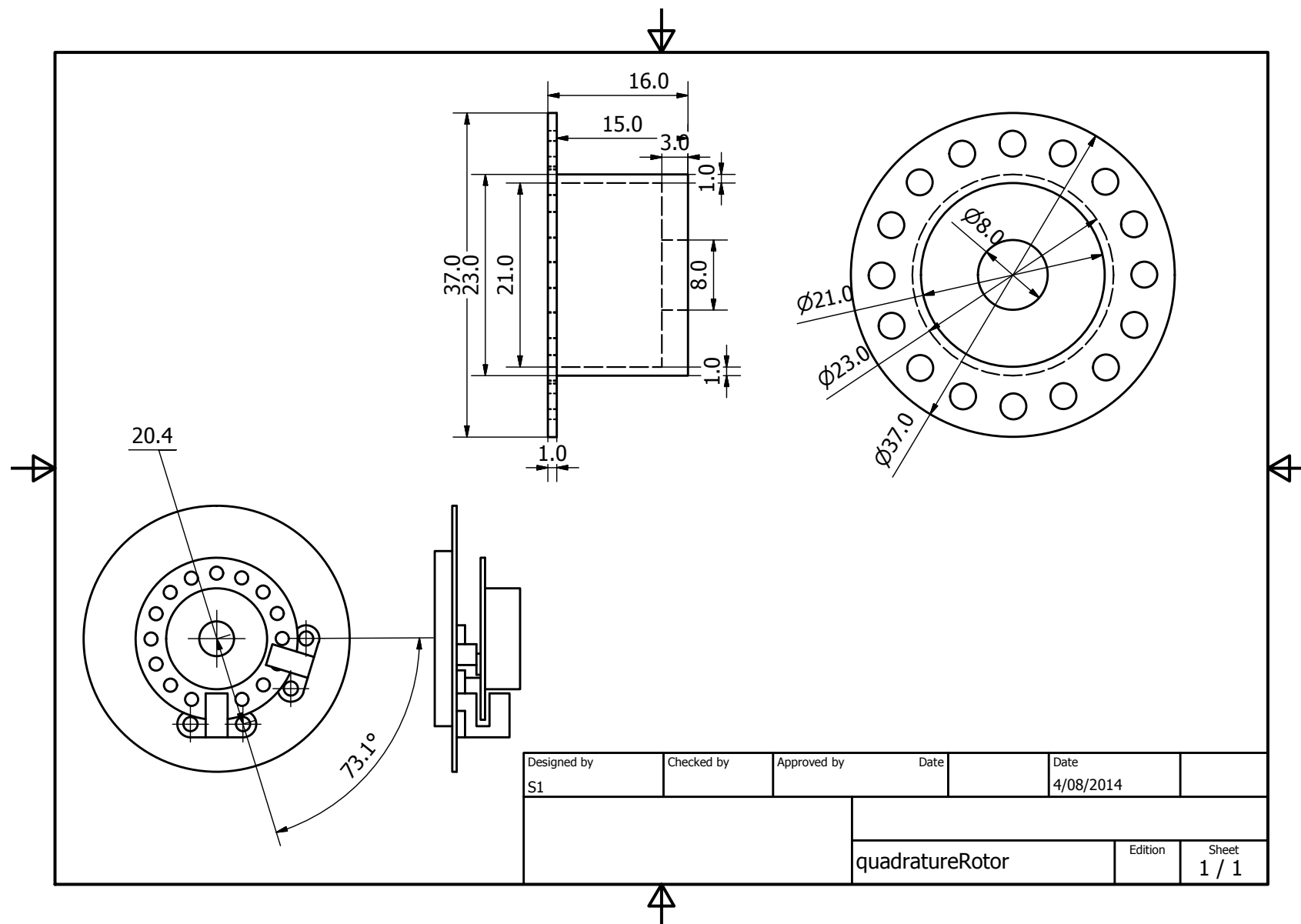


Figure D.8: Quadrature Encoder Mask

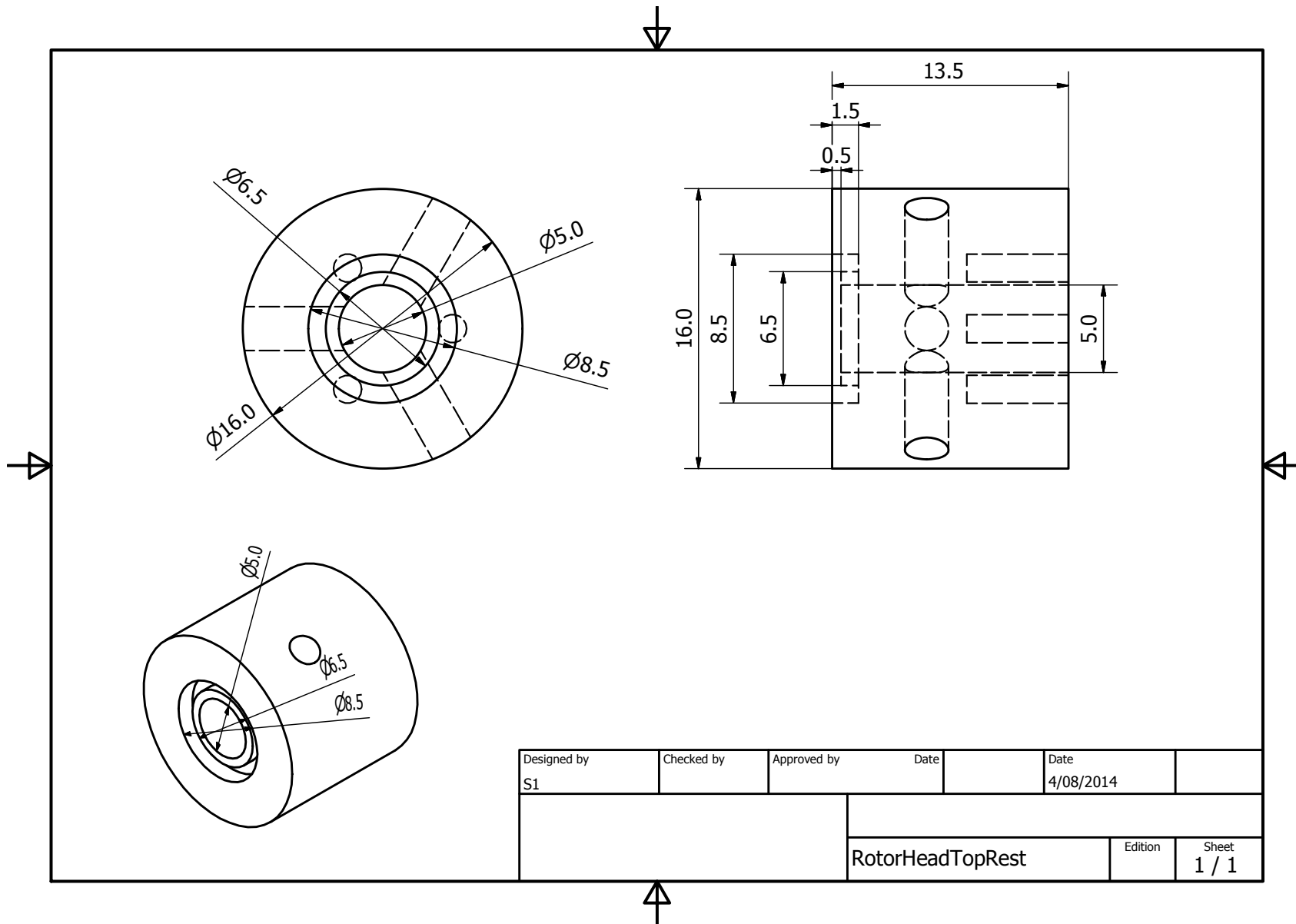


Figure D.9: Rotor Head Top Rest

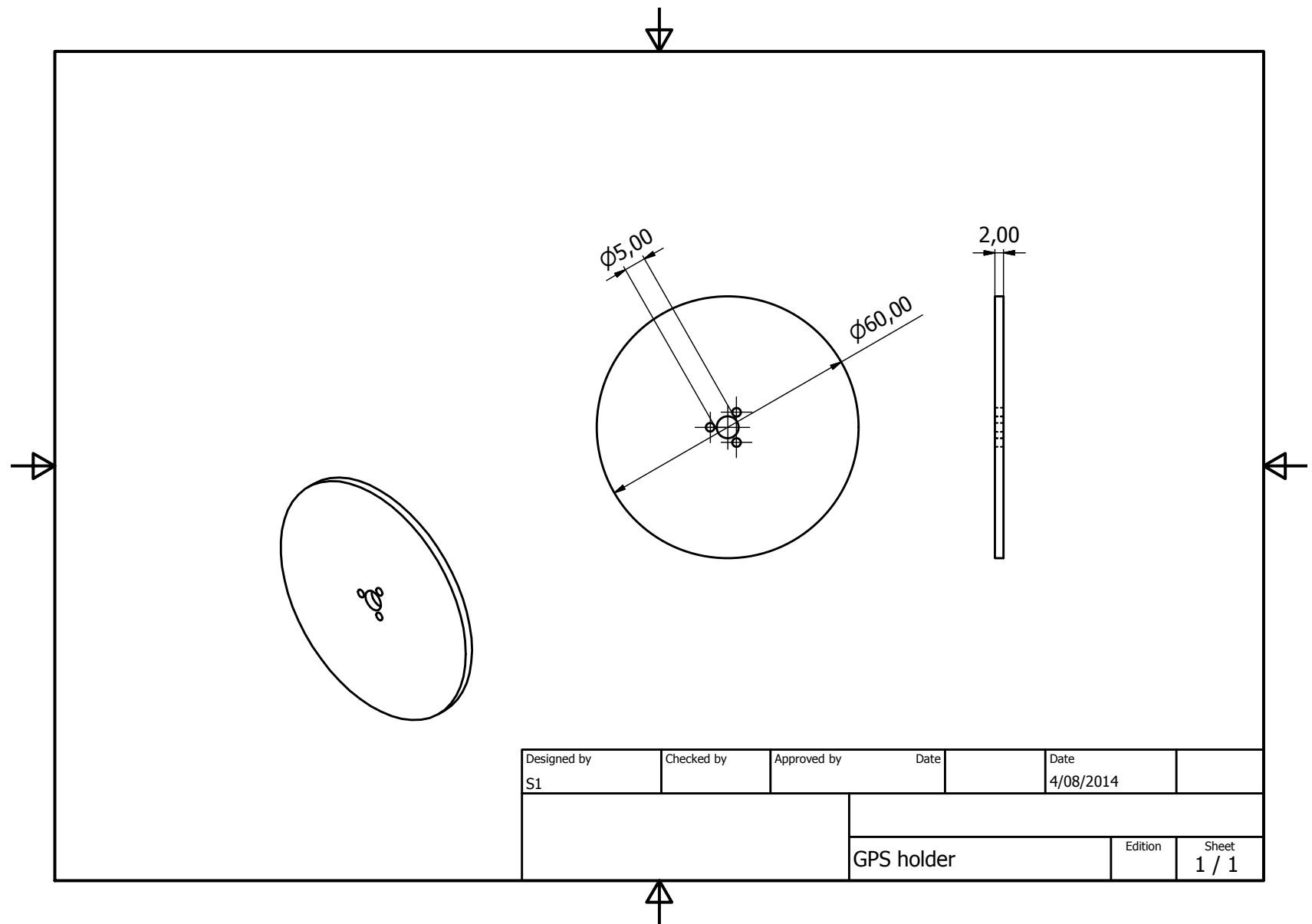


Figure D.10: Sensor Assembly Platform

Appendix E

Electrical Schematics

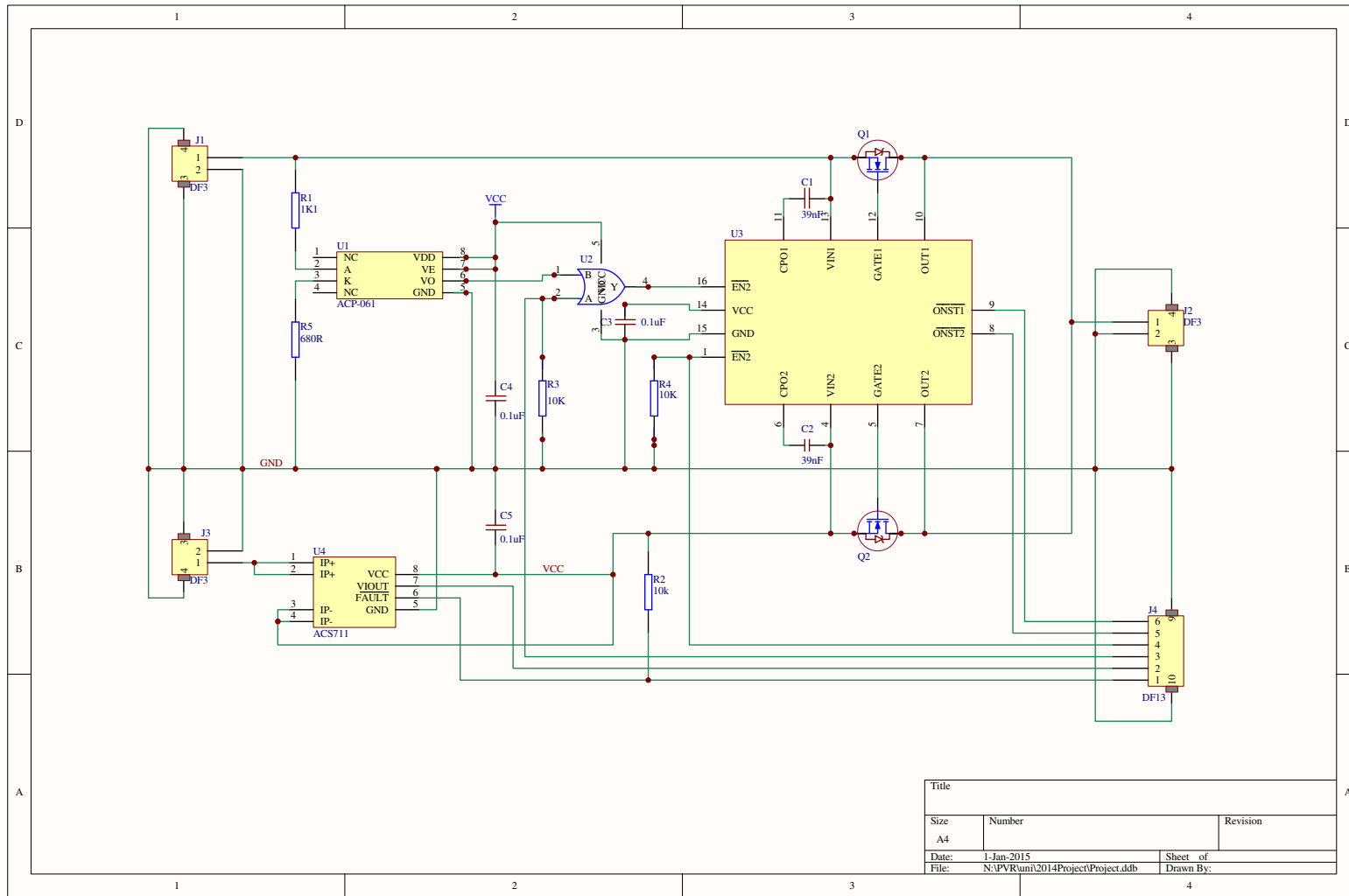


Figure E.1: Schematic Battery Controller

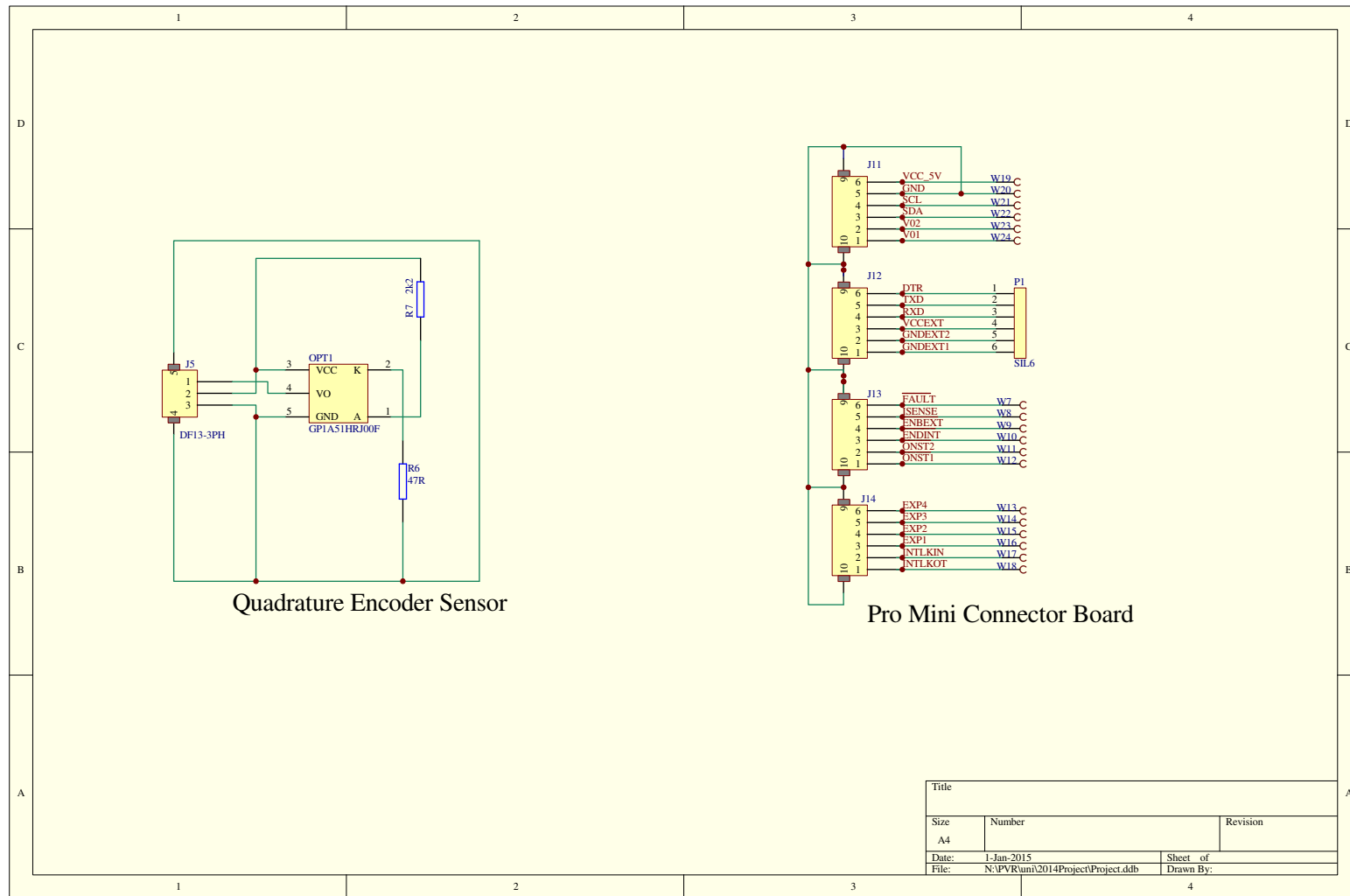


Figure E.2: Schematic of Quadrature Encoder and Pro Mini Connector Boards

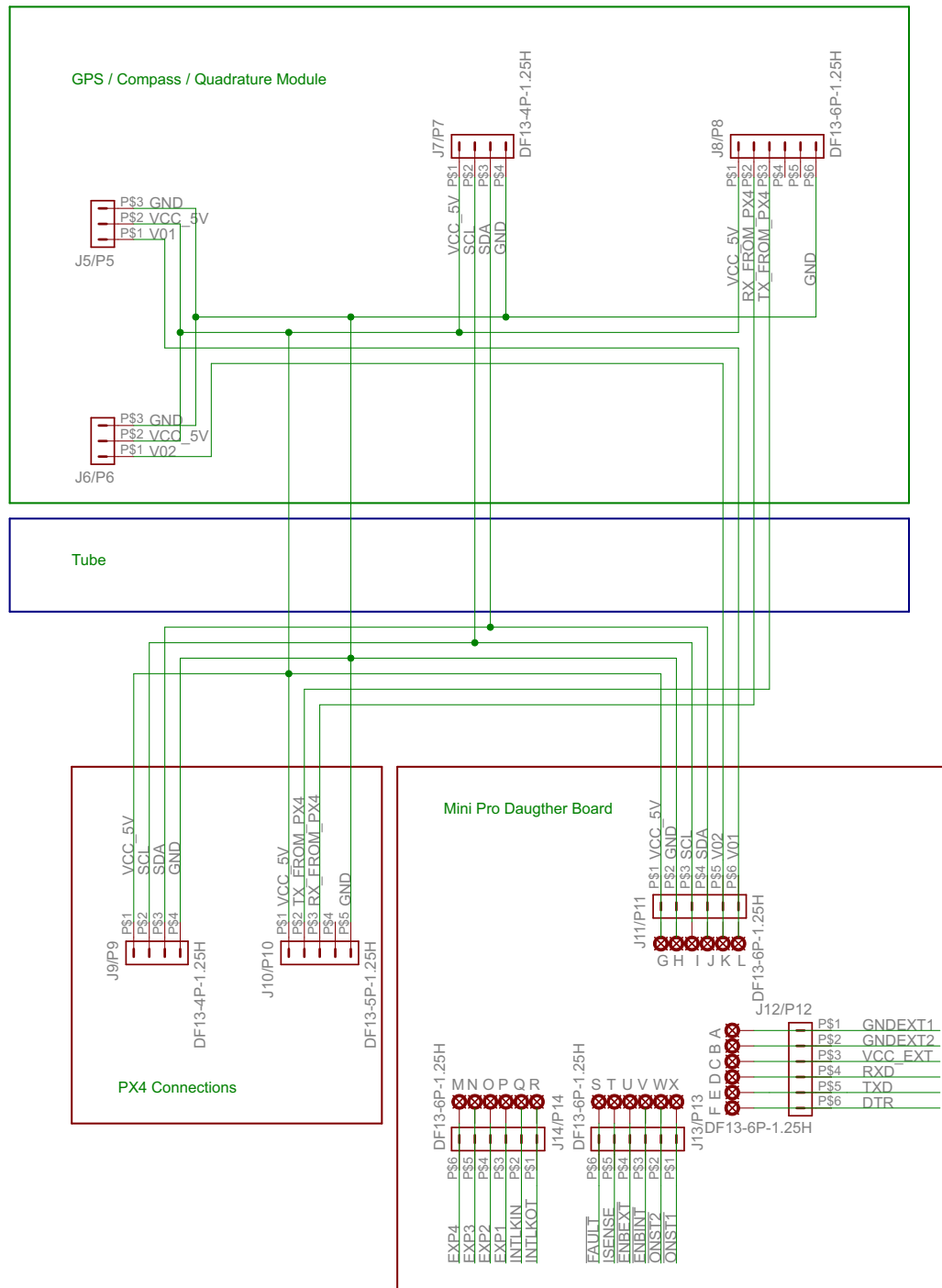


Figure E.3: Sensor Assembly Harness

Appendix F

RDS Software Design

This appendix contains Software Design Disclosure for the all processors involved in satisfying the System Requirements allocated to the Host FMU, Host Release Controller, RDS FMU and RDS Sensor Manager. This disclosure is focused on the Safe Carriage and Release of the RDS from the Host RPAS and does not cover the Navigation or Guidance.

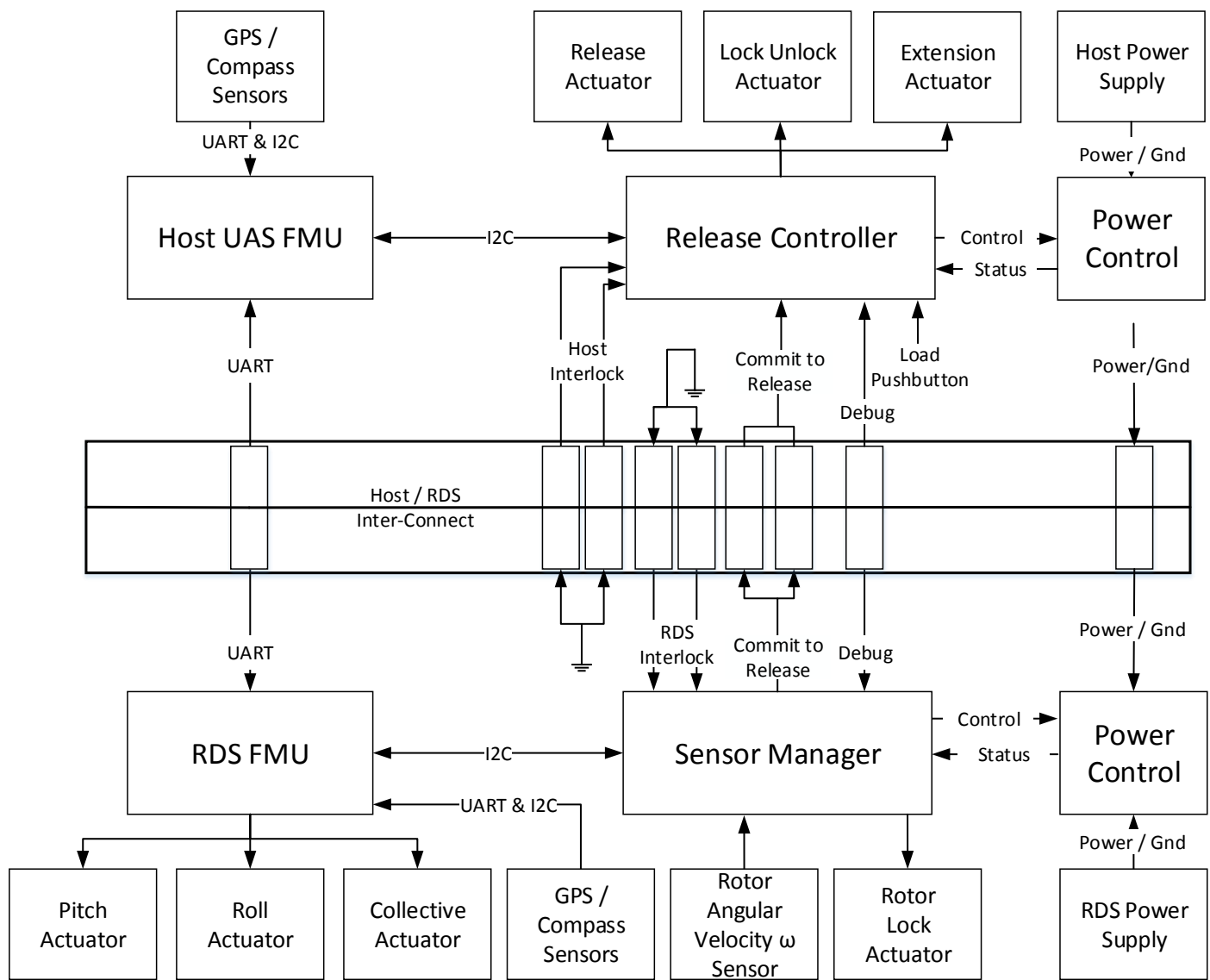


Figure F.1: High Level Interconnect System Diagram

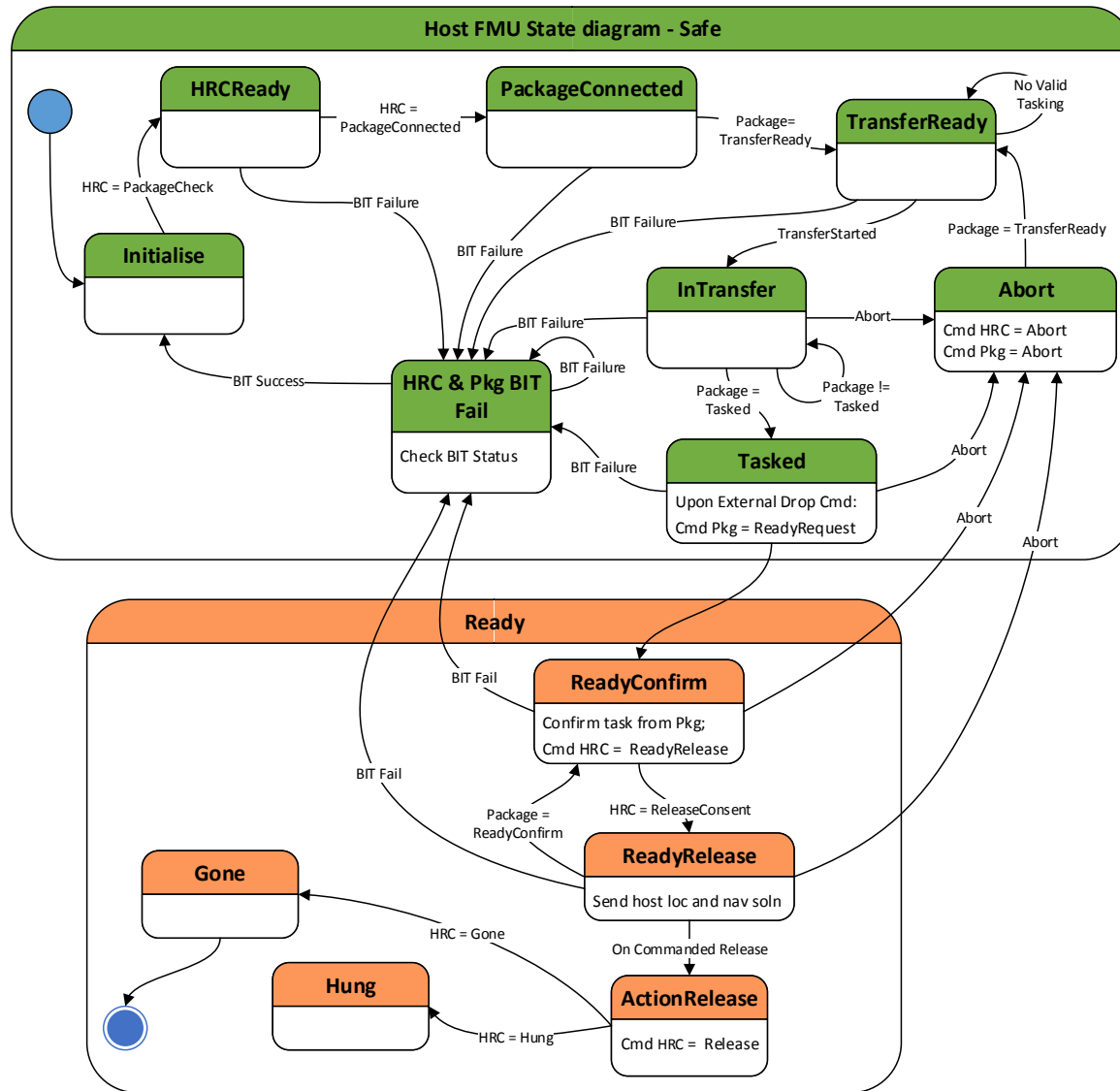


Figure F.2: Host FMU State Diagram

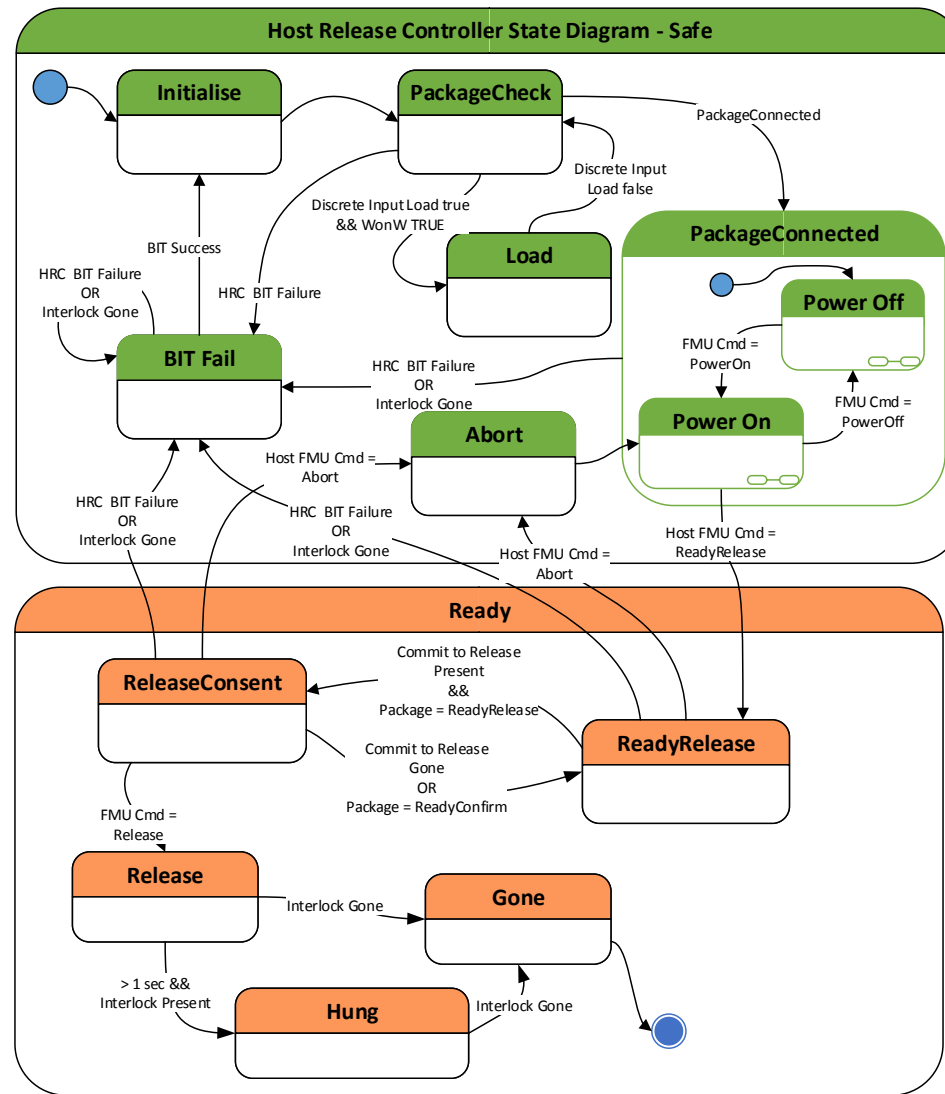


Figure F.3: Host Release Controller State Diagram

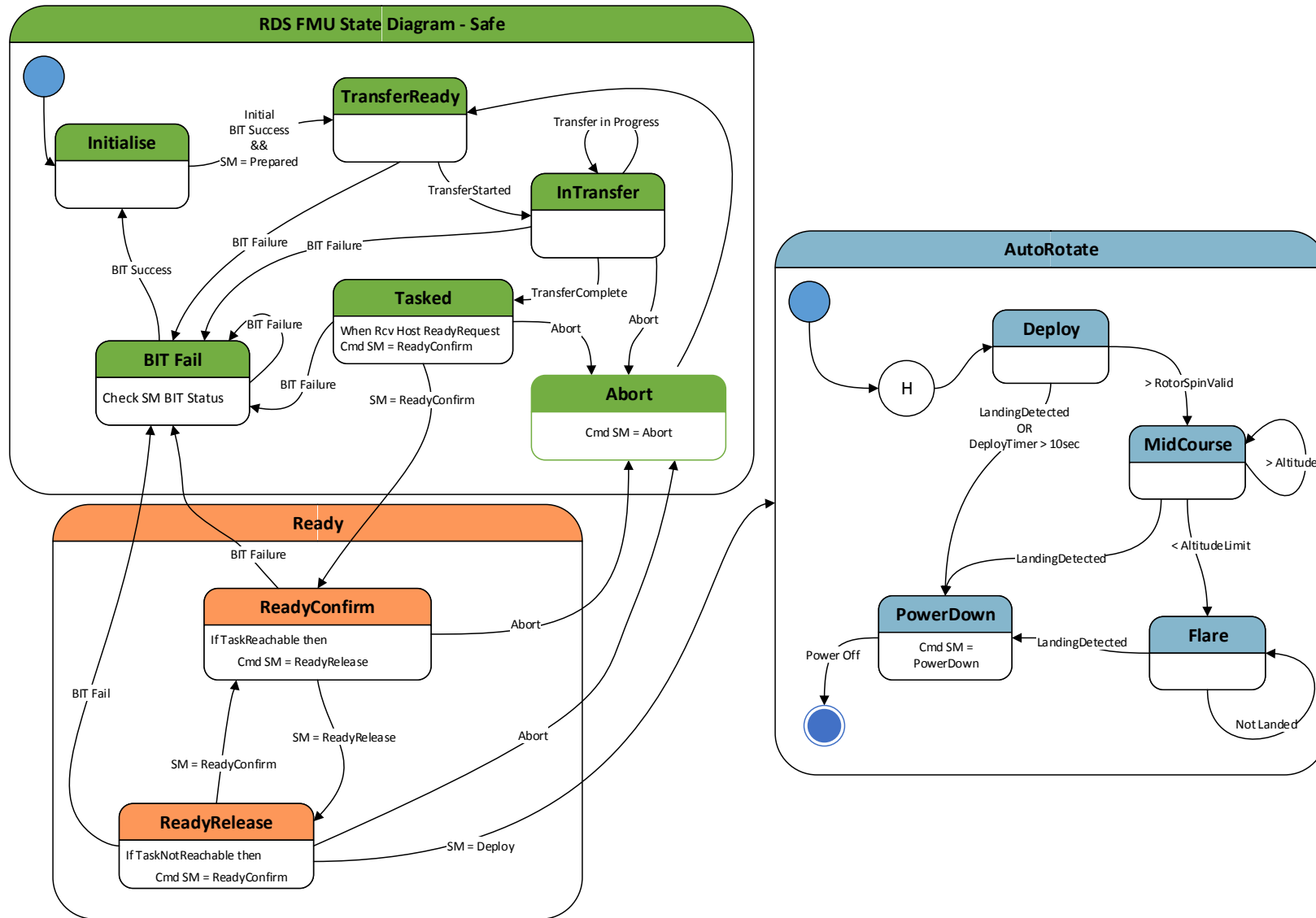


Figure F.4: RDS FMU State Diagram

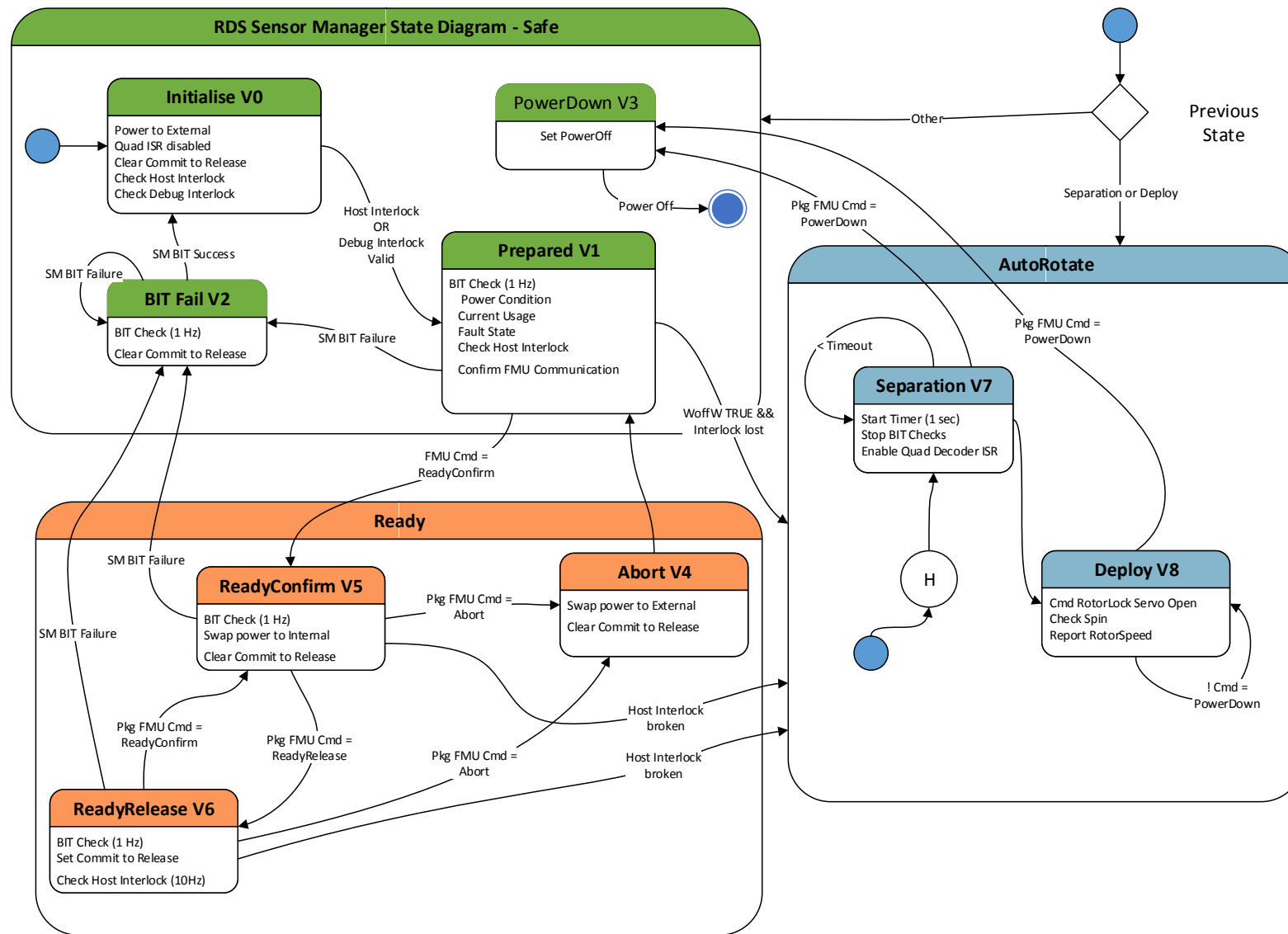


Figure F.5: RDS Sensor Manager State Diagram

Appendix G

Risk Analysis

Description of Hazard		People At risk	Number At risk	Parts of Body	Risk Level	
Moving rotor blades or parts impact with operator (this is applicable to both the ground lift and flare test rigs)		Operator	1	eyes, face, hands	Marginal Consequences - Remote Likelihood	
Categories	Short Term Controls		Long Term Controls		Completion Details	
Design	Provide visible distance marker to identify rotor diameter footprint		Nil		Completed	
Safety Devices	Include wire cage encapsulating rotating blades at eye height		Nil		Not effective	
Warning Devices						
Procedures and Training						
P.P.E.						

Table G.1: Risk Management Chart for Ground Test Apparatus

Description of Hazard		People At risk	Number At risk	Parts of Body	Risk Level	
Personnel fall from heights during installation procedure		Operator	1	Body	Critical Consequences - Remote Likelihood	
Categories	Short Term Controls		Long Term Controls		Completion Details	
Design	Design apparatus to be constructed at ground level and then raised Roof Workers Safety Harness to be used Use a ladder to access majority of work vice rooftop.		Nil		Design Complete	
Safety Devices			Nil		Safety Harness available	
Warning Devices						
Procedures and Training P.P.E.	Training in use of Safety harness		Nil		Training completed	

Table G.2: Risk Management Chart for Working at Heights

Description of Hazard		People At risk	Number At risk	Parts of Body	Risk Level
Injury sustained to hands, feet or eyes during machine work		Operator	1	Body	Marginal Consequences - Remote Likelihood
Categories	Short Term Controls		Long Term Controls		Completion Details
Design Safety Devices Warning Devices Procedures and Training P.P.E.	Ensure availability of eye protection Ensure availability of footwear (steelcapped boots) Ensure use of appropriate tools)				Available Available Available and Training Received

Table G.3: Risk Management Chart for Work Place Safety

Description of Hazard		People At risk	Number At risk	Parts of Body	Risk Level
Injury sustained following impact of package onto structure or personnel during descent		Operator	1	Body	Critical Consequences - Remote Likelihood
Categories	Short Term Controls	Long Term Controls		Completion Details	
Design	Include safety signal authorisation prior to release Incorporate readiness checks prior to release Incorporate redundant power supplies Complete testing of control algorithm with Hardware in the Loop			Not Required Not Required Implemented Implemented	
Safety Devices					
Warning Devices	Include high visibility Package colour scheme			Not Required	
Procedures and Training	Establish procedures for Flight Test Execution Establish cleared boundary around target			Not Required Not Required	
P.P.E.					

Table G.4: Risk Management Chart for Impact of package onto structure or personnel during descent

Appendix H

Source Listings

H.1 Nameing Conventions

The following tables from, (Alex 2007) , were used withn this project to describe the common variable and naming conventions used within the software listings.

Table H.1: Type prefix

Type prefix	Meaning	Example
b	boolean	bool bHasEffect;
c (or none*)	class	Creature cMonster;
ch	char (used as a char)	char chLetterGrade;
d	double, long double	double dPi;
e	enum	Color eColor;
f	float	float fPercent;
n	short, int, long	int nValue;
	char used as an integer	
s	struct	Rectangle sRect;
str	C++ string	std::string strName;
sz	Null-terminated string	char szName[20];

The following type modifiers are placed before the prefix if they apply:

Table H.2: Type modifier

Type modifier	Meaning	Example
a	array on stack	int anValue[10];
p	pointer	int* pnValue;
pa	dynamic array	int* panValue = new int[10];
r	reference	int rnValue;
u	unsigned	unsigned int unValue;

The following scope modifiers are placed before the type modifier if they apply:

Table H.3: Scope modifier

Scope modifier	Meaning	Example
g-	global variable	int g_nGlobalValue;
m_	member of class	int m_nMemberValue;
s_	static member of class	int s_nValue;

H.2 Sensor Manager Listings

H.3 The SensorManager.ino Code

Listing H.1: The main Sensor Manager Sketch.

```

/*
 * SensorManager.ino
 *
 * Created on: 27 Aug 2014
 * Author: Ian Saxby
 */

#include <Arduino.h>

// #include <Wire.h> // I2C library
#include <NilRTOS.h>
#include <DigitalIO.h>
// Use tiny unbuffered NilRTOS NilSerial library.
#include <NilSerial.h>

// Macro to redefine Serial as NilSerial to save RAM.
// Remove definition to use standard Arduino Serial.
#define Serial NilSerial

#include <Wire.h>
#include <NilFIFO.h>
#include <NilAnalog.h>
#include <NilTimer1.h>
#include "ServoTimer2.h"

#include "PinoutConfigSM.h"
#include "QuadEncoder.h"
#include "Power.h"
#include "BIT.h"
#include "I2CBuffer.h"
#include "StateMachine.h"
#include "Commander.h"

// FIFO of received Command Messages

NilFIFO<I2CMsgRx, FIFO_DEPTH>* pcFIFO;

// i2c settings
#define SLAVE_ADDRESS 0x33

// Task Tick Counter, initialised to 0. Range 0 to 9
uint8_t nTaskTick = 0;

// Declare global pointers to the various controller objects
// These are instantiated during setup
Commander* pcCmdr;
BIT* pcBIT;

```

```

// I2C Buffer Arrangements
// Establish a I2C Slave Transmission Message Buffer Class pointer
I2CBuffer *g_cI2C_MsgTx;

// Setup a buffer for storing I2C Tx data for transferring to
// I2C Slave Transmission Message Buffer
I2CMsgTx g_uI2CUpdate_MsgTx;

// Setup a global buffer for initial I2C receipt before storing in
// ICP FIFO
I2CMsgRx g_uI2CTemp_MsgRx;

// Setup a global buffer for I2C dispatch for I2C to Transmit from
I2CMsgTx g_uI2CTemp_MsgTx;

// declare servoTimer2 object ptr to control the Rotor lock servo
ServoTimer2* pcRotorServo;

// Global weight off wheels variable
uint8_t g_eWheelOffWheels;

// Circular buffer for Omega average
uint16_t g_nOmegaCirBuf[CIRCBUFSIZE];
// Direction of rotation, initialised to reverse.
bool g_bDirCW = false;
// Direction of rotation, initialised to forward. forces two cycles
// before initiating first detection
bool g_bLastDirCW = true;

// Declare and initialize a semaphore for limiting access to a region.
SEMAPHORE_DECL(cSem1Hz, 0);
SEMAPHORE_DECL(cSem5Hz, 0);
SEMAPHORE_DECL(cSem10Hz, 0);
SEMAPHORE_DECL(cSemDebug, 0);

NIL_WORKING_AREA(waThread1, 96);

// Declare thread function for thread 1.
NIL_THREAD(Thread1, arg)
{
    for (uint8_t i = 0; i < NUMDECBUF; i++)
    {
        g_DecBuf[i].u_nDeltaN.n32DeltaN = 0;
    }

    // Wait for Ready State to be entered before initialising Quad
    // Decoder Interrupts
    nilTimer1Start(TSC_PERIOD);

    while (true)

```

```

{

    // Execute 10Hz Commander related functionality
    pcCmdr->Task10Hz();

    // Increment the Task Tick counter
    nTaskTick++;

    // Now determine if slower Tasks require to be run in this
    // slot
    switch (nTaskTick)
    {
        case 4: // every 10th tick (at time slot 4) = 1 Hz
        {
            // Signal 1Hz Task to run
            // Release the Semaphore but not Reschedule RTOS
            // until end

            nilSemSignalI(&cSem1Hz);
            break;
        }
        case 2:// every 5th tick (at time slots 2 and 7) = 5 Hz
        case 7:
        {
            // Signal 5Hz Task to run
            // Release the Semaphore but not Reschedule RTOS
            // until end
            nilSemSignalI(&cSem5Hz);
            break;
        }
        case 10: // on the 10th tick reset TaskTick count to 0
        {
            nTaskTick = 0;
            break;
        }
    }
}

//DEBUG Code
if (pcCmdr->RetrieveState() == DEPLOY)
{
    // Enable the Debug Code
    nilSemSignalI(&cSemDebug);
}

// Sleep so lower priority threads can execute.
nilTimer1Wait();
}

}

//-----
// Declare a stack with 64 bytes beyond context switch and
// interrupt needs
NIL_WORKING_AREA(waThread3, 96);

```

```

// Declare thread function for thread 2.
NIL_THREAD(Thread3, arg)
{

    // 5 Hz Task

    while (TRUE)
    {
        // Wait for notification to run task
        nilSemWaitTimeout(&cSem5Hz, TIME_INFINITE);

        // Execute 5Hz Commander related functionality
        pcCmdr->Task5Hz();
    }
}

//-----
// Declare a stack with 64 bytes beyond context switch and
// interrupt needs
NIL_WORKING_AREA(waThread4, 96);

// Declare thread function for thread 4.
NIL_THREAD(Thread4, arg)
{

    // *****
    // 1 Hz Task
    // *****

    while (TRUE)
    {
        // Wait for notification to run task
        nilSemWaitTimeout(&cSem1Hz, TIME_INFINITE);

        // Execute 1Hz Commander related functionality
        pcCmdr->Task1Hz();

        //pcBIT->BITCheck();
    }
}

//-----
// Declare a stack with 64 bytes beyond context switch and
// interrupt needs
NIL_WORKING_AREA(waThread5, 64);

// Declare thread function for thread 4.
NIL_THREAD(Thread5, arg)
{
    // Wait for notification to run task
    nilSemWaitTimeout(&cSemDebug, TIME_INFINITE);

    systime_t waketime = nilTimeNow();

```



```

uint16_t nAvgOmega = 0;

while (TRUE)
{

    waketime += MS2ST(1000);
    nilThdSleepUntil(waketime);

    if(IsDebugMode())
    {
        nAvgOmega = g_nOmegaCirBuf[0];

        for (uint8_t i = 1; i < CIRCBUF_SIZE; i++)
        {
            nAvgOmega += g_nOmegaCirBuf[i];
        }
        nAvgOmega /= CIRCBUF_SIZE;

        Serial.print(F("␣AvgW:␣"));
        Serial.print(nAvgOmega);
        Serial.print(F("␣Dir:␣"));
        if (g_bDirCW)
        {
            Serial.print(F("␣CW␣"));
        }
        else
        {
            Serial.print(F("␣CCW␣"));
        }
        Serial.print(waketime);
        Serial.println();
    }
}

/*
 * Threads static table, one entry per thread. A thread's priority is
 * determined by its position in the table with highest priority first.
 *
 * These threads start with a null argument. A thread's name is also
 * null to save RAM since the name is currently not used.
 */
NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY(NULL, Thread1, NULL, waThread1, sizeof(waThread1))
NIL_THREADS_TABLE_ENTRY(NULL, Thread3, NULL, waThread3, sizeof(waThread3))
NIL_THREADS_TABLE_ENTRY(NULL, Thread4, NULL, waThread4, sizeof(waThread4))
NIL_THREADS_TABLE_ENTRY(NULL, Thread5, NULL, waThread5, sizeof(waThread5))

NIL_THREADS_TABLE_END()
//

```

```

void PinOutInitialisation(void);

void setup()
{
    // Instantiate the I2C Slave Transmission Message Buffers
    g_cI2C_MsgTx = new I2CBuffer();

    // Instantiate the Servo Controller Object
    pcRotorServo = new ServoTimer2;

    // Initialise uC MiniPro Input Output Pin configuration
    PinOutInitialisation();

    // Instantiate the FIFO buffer
    pcFIFO = new NilFIFO<I2CMsgRx, FIFO_DEPTH>;

    // Setup Serial Communication
    Serial.begin(9600);

    Serial.println("");
    Serial.println("_RESTART_");

    // initialize i2c as slave
    Wire.begin(SLAVE_ADDRESS);

    // define callbacks for i2c communication
    Wire.onReceive(receiveData);
    Wire.onRequest(sendData);

    // Instantiate the Commander
    pcCmdr = new Commander;

    // start kernel
    nilSysBegin();
}

//-----
// Loop is the idle thread. The idle thread must not invoke any
// kernel primitive able to change its state to not runnable.
void loop()
{
}

//I2C callbacks

// callback for received command
void receiveData(int byteCount)
{
    int indx = 0;

```

```

uint8_t nOverflow;
I2CMsgRx* psFIFOSlot;

while(Wire.available())
{
    // Ensure no Msg Received is longer than largest expected
    if(indx > MAX_MSG_RX_LENGTH)
    {
        // Create failure condition here
        // Set the Receive Buffer Overflow Attempted BIT flag
        pcBIT->BITFlagUpdate(RXBUFF_OVRFLW, true);
        // Break out of Receive process
        break;
    }
    else
    {
        // initial fast receipt of I2C Command Data
        g_uI2CTemp_MsgRx.g_nRxAllData[indx] = Wire.read();
        indx++;
    }
}

if ((indx > 0) && (indx <= MAX_MSG_RX_LENGTH))
{
    if (g_uI2CTemp_MsgRx.g_nType1RxMsgNo == SETUP_I2C_SEND_CMD)
    {
        // Retrieve the requested message in readiness for subsequent
        // SendData() activity
        g_cI2C_MsgTx->getTxMessage(\
            (uint8_t)g_uI2CTemp_MsgRx.g_nType1RxEnumValue,
            &g_uI2CTemp_MsgTx);
    }
    else
    {
        // Store Received I2C Command into FIFO

        // Get a free FIFO slot.
        psFIFOSlot = pcFIFO->waitFree(TIME_IMMEDIATE);

        // Only store if free space. else set Bit Fail condition
        if (psFIFOSlot != 0)
        {
            // Store message into FIFO.
            (*psFIFOSlot) = g_uI2CTemp_MsgRx;

            // Signal thread data is available.
            pcFIFO->signalData();
        }
        else
        {

```

```

        // Set the Receive Buffer Overflow Attempted BIT flag
        pcBIT->BITFlagUpdate(FIFO_OVRFLW, true);

    }
}
}

// callback for sending data via I2C
void sendData()
{
    Wire.write(g_uI2CTemp_MsgTx.g_nTxAllData, \
               g_uI2CTemp_MsgTx.nWdCount);
}

void PinOutInitialisation()
{
    // Initialisation for Quadrature Decoding
    // GP1A51HRJ00F has internal pull-up resistor
    pinMode(PIN_CHANNEL_A, INPUT);
    pinMode(PIN_CHANNEL_B, INPUT_PULLUP);

    // External Power Enable Output Pin 4
    pinMode(PIN_POWERENB_EXT, OUTPUT);
    // Internal Power Enable Output Pin 5
    pinMode(PIN_POWERENB_INT, OUTPUT);

    // Attach the PWM servo class to the Rotor Lock Output pin 6
    // attach the given pin to the next free channel, sets
    // pinMode, returns channel number or 0 if failure
    pcRotorServo->attach(PIN_ROTORLOCK_PWM);

    // Close the Rotor Lock to Lockin the Rotor blades
    pcRotorServo->write(ROTOR_CLOSE);

    // alternate to establish servo movement range
    // pcRotorServo->attach(ROTORLOCK_PWM, int min, int max);

    // Initialisation for Interlock and Discrete Input / Outputs
    pinMode(PIN_PKG_INTERLOCK_OUT, OUTPUT);

    // Host connected interlock Pin Input, requires pull-up resistor
    // setting Host disconnected is Active High and no cycle
    pinMode(PIN_PKG_INTERLOCK_IN, INPUT_PULLUP);

    pinMode(PIN_COMMIT_RELEASE, OUTPUT);

    // Debug Pin Input, requires pull-up resistor setting
    // Debug mode is Active Low
    pinMode(PIN_PKG_DEBUG, INPUT_PULLUP);
    // Current Sensor Fault Input Pin, circuit has pull-up
    // resistor

```

```
pinMode(PIN_CURRENTSENSORFAULT, INPUT);
// External Power Status flag, LTC4353 has internal pull-up
// resistor
pinMode(PIN_PWRSTATUS_EXT_ONST1, INPUT);
// Internall Power Status Flag, LTC4353 has internal pull-up
// resistor
pinMode(PIN_PWRSTATUS_INT_ONST2, INPUT);

// Debug
// Set ChA Low
// digitalWrite(CHANNEL_A, LOW);
// Set ChB Low
//digitalWrite(PIN_CHANNEL_B, LOW);
}
```

H.4 The Commander.h Code

Listing H.2: The Commander header file.

```

/*
 * Commander.h
 *
 * Created on: 3 Sep 2014
 * Author: 0050083462
 */

#ifndef COMMANDER_H
#define COMMANDER_H

#include <Arduino.h>
#include <stdint.h>
#include "I2CBuffer.h"
#include "StateMachine.h"
#include "Power.h"
#include "Interface.h"

class Commander: public StateMachine, public Power, public Interface
{
public:
    Commander();
    void Task10Hz(void);
    void Task5Hz(void);
    void Task1Hz(void);
    void ExecuteCmd(I2CMsgRx* sTempRxMsgBuffer);
    // void PowerCheck(void);
    virtual ~Commander(){}

private:
    // State based control variables
    uint8_t nSeparationCounter;

    I2CMsgRx sTempRxMsgBuffer;
};

#endif /* COMMANDER_H */

```

H.5 The Commander.cpp Code

Listing H.3: The Commander source file.

```

/*
 * Commander.cpp
 *
 * Created on: 3 Sep 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include <NilRTOS.h>
#include <NilFIFO.h>
#include <NilSerial.h>
#include <util/atomic.h>

#include "ServoTimer2.h"
#include "Commander.h"
#include "PinoutConfigSM.h"
#include "StateMachine.h"
#include "BIT.h"
#include "QuadEncoder.h"
#include "Interface.h"
#include "I2CBuffer.h"

extern ServoTimer2* pcRotorServo;
extern BIT* pcBIT;
extern NilFIFO<I2CMsgRx, FIFO_DEPTH>* pcFIFO;
extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

extern uint16_t g_nOmegaCirBuf[CIRCBUFSIZE];
// Direction of rotation, initialised to reverse.
extern bool g_bDirCW;
// Direction of rotation, initialised to forward. forces
// two cycles before initiating first detection
extern bool g_bLastDirCW;

ISR(ChannelA_vect)
{
    uint32_t tISR;

    // Determine time of interrupt
    tISR = micros();

    // Determine direction of rotation
    // As this is INT0 isr on RISING INT0 will be high
    digitalRead(PIN_CHANNEL_B) ? g_bDirCW = false : g_bDirCW = true;

```

```

    if(g_bDirCW == g_bLastDirCW)
    {
        // Store away the timestamp of this Quadrature interrupt
        g_DecBuf[g_nBufferIndx].DecoderISRTime = tISR;
        // Increment the count of Decoder interrupts
        g_DecBuf[g_nBufferIndx].u_nDeltaN.n16DeltaN++;
    }
    else
    {
        g_bLastDirCW = g_bDirCW;
        // Clear the count of Decoder interrupts
        g_DecBuf[g_nBufferIndx].u_nDeltaN.n16DeltaN = 0;
    }
}
/* 2nd interrupt routine If required for higher
 * accuracy
ISR(ChannelB_vect)
{
    uint32_t tISR;

    // Determine time of interrupt
    tISR = micros();

    // Determine direction of rotation
    // As this is INT1 isr on RISING INT1 will be high
    digitalRead(PIN_CHANNEL_A) ? g_bDirCW = true : g_bDirCW = false;

    if(g_bDirCW == g_bLastDirCW)
    {
        // Store away the timestamp of this Quadrature interrupt
        g_DecBuf[g_nBufferIndx].DecoderISRTime = tISR;
        // Increment the count of Decoder interrupts
        g_DecBuf[g_nBufferIndx].u_nDeltaN.n16DeltaN++;
    }
    else
    {
        g_bLastDirCW = g_bDirCW;
        // Clear the count of Decoder interrupts
        g_DecBuf[g_nBufferIndx].u_nDeltaN.n16DeltaN = 0;
    }
}
*/
// Commander Constructor
Commander::Commander()
{
    // Instantiate the BIT Object
    pcBIT = new BIT;

    // Confirm that the device is not already in an Autorotate State
    // If so set power to internal supply else external
    if(RetrieveState() > READYRELEASE)

```



```

{
    // Already in autorotate state so cmd internal power source
    if (!SwitchPowerSupply(PWRSOURCEINT))
    {
        // Failure to transition to INT power so set BIT fail
        SetState(BITFAIL);
    }
    // and attach the ISR vector to the INT0 for Encoder ChA
    attachInterrupt(0, ChannelA_vect, RISING);
    // attachInterrupt(1, ChannelB_vect, RISING);

    // and stop the BIT Checks
    pcBIT->ModifyBITCheckFlag(false);
}
else
{
    // Not in an Autorotate state therefore default initialise to
    // use External power source
    if (!SwitchPowerSupply(PWRSOURCEEXT))
    {
        // Failure to transition to EXT power so set BIT fail
        SetState(BITFAIL);
    }
    // Setup the Host Interlock wrap around output
    OutputHostInterlockSignal();

    // Initialise the Separation Timer
    nSeparationCounter = SEPARATIONCOUNT;
}
}

void Commander::Task10Hz(void)
{
    // Quadrature Encoder variables
    uint32_t temp1;
    uint32_t temp2;
    uint32_t temp3;
    uint32_t tnow;
    // Delta Th
    uint32_t nDeltaTh;
    //Copy of current global ISR DoubleBuffer Index
    uint16_t nThisIndex;
    // Tsc,acc Extended Observation Window (32bit word,
    // using fixed point division)
    c_UTscacc u_nSumTscacc;

    // Ensure spare words are initialised to 0
    u_nSumTscacc.spare = 0;

    // Angular rotation variable [Should be direct to I2C repository
    // though]
    uint16_t nOmega ;

```

```

// Internal counter for Abort phase
static uint8_t nExtPowerDelay = 0;

volatile bool bPulseDetected;
static bool bPowerDownFirstPass = true;

switch(RetrieveState()) // In priority order
{
    case DEPLOY:
    {
        // Quadrature Decoding Section
        // Increment the Tsc counter to calculate extended Tsc
        // period
        g_nTscTick++;

        // Create atomic sequence of instructions
        nilSysLock();

        // Determine current time in usec
        tnow = micros();

        // Calculate Omega only if Encoder pulses have been
        // detected
        // Check using current Buffer index
        if (g_DecBuf[g_nBufferIndx].u_nDeltaN.n16DeltaN > 0)
        {
            bPulseDetected = true;
            // record this buffer pointer
            nThisIndex = g_nBufferIndx;
            // Switch over the buffer index
            g_nBufferIndx ^= 1;
        }
        else
        {
            bPulseDetected = false;
        }
        // Release the atomic instruction set
        nilSysUnlock();

        if(bPulseDetected)
        {
            // Calculate Delta Th
            nDeltaTh =
                tnow - g_DecBuf[nThisIndex].DecoderISRTime;

            // Calculate SUM Tsc, acc = g_nTscTick * TSC_PERIOD;
            // scaled n16SumTscacc 15
            u_nSumTscacc.n16SumTscacc = g_nTscTick * 3;
            // Now Tsc, acc scaled into 32 bit word (scale 0)
            temp1 = u_nSumTscacc.n32SumTscacc >> 1;
        }
    }
}

```

```

// Calculate Omega =  $\frac{DeltaN}{(SumTsc, acc + DeltaT(h-1) - DeltaT(h))} * \frac{60}{Np}$ 
//
// temp2 scale (16)
temp2 =
(uint32_t)(g_DecBuf[nThisIndex].u_nDeltaN.n32DeltaN*60);
// temp3 scale (20-4)
temp3 =
(uint32_t)(temp1 + g_DecBuf[nThisIndex].nDeltaThm1 -
nDeltaTh)
    >> 4;
nOmega = (uint32_t)(temp2/temp3) >> NP;
// Negate an approx constant ~4.5% from result due to
// over reading above calculation
nOmega -= ((nOmega >> 6) * 3);
// Limit the Omega output
if(nOmega > MAXLIMIT.OMEGA)
    nOmega = MAXLIMIT.OMEGA;

g_nOmegaCirBuf[g_nOCirIndx] = nOmega;

// Now update circular buffer index for next pass
g_nOCirIndx = (g_nOCirIndx + 1) % CIRCBUFSIZE;

// *****
// Save away Omega and Direction into I2C message buffer
// area

// *****
// DIRECTION      g_bDirCW
// Angular Speed nOmega
// First Invalidate current message area
g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG1);

if (g_bDirCW)
{
    // Rotor travelling CW
    g_uI2CUpdate_MsgTx.g_nType2IntValue =
        (int16_t)nOmega;
}
else
{
    // Rotor travelling CCW
    g_uI2CUpdate_MsgTx.g_nType2IntValue =
        (int16_t)(~nOmega + 1);
}
// Now save into I2C Tx Message Common Area
g_cI2C_MsgTx->putMessage(MSG1, &g_uI2CUpdate_MsgTx);

// Finally Validate Msg area
g_cI2C_MsgTx->I2CValidateTxMsg(MSG1);

```

```

        // Save Delta T(h) as Delta T(h-1)
        g_DecBuf[nThisIndex].nDeltaThm1 = nDeltaTh;

        // Clear Extended Observation Window counter
        g_nTscTick = 0;

        // Clear the count of Decoder interrupts for next time
        // use of this buffer
        g_DecBuf[nThisIndex].u_nDeltaN.n16DeltaN = 0;
    }
    else
    {
        // If extended observation window is greater than
        // "Constant" ticks then set angular speed = 0
        if(g_nTscTick > MAXEXTOWSWIN)
        {
            g_nTscTick = 0;
            g_nOmegaCirBuf[g_nOCirIndx] = 0;
            g_nOCirIndx = (g_nOCirIndx + 1) % CIRCBUFSIZE;
        }
    }
    break;
}
case SEPARATION:
{
    // The majority of SEPARATION initialisation has been
    // completed within the ReadyRelease to Separation
    // changeover portion of the ReadyRelease Case.
    // *****
    // The following assumes no power loss following release
    // need to change this as other functions do account for
    // such
    // *****
    // If Low means Interlock disconnected, the package is
    // falling!
    // Therefore:
    // a. Decrement the Separation timer

    // Decrement the Separation timer, each decrement
    // ~= 100msec
    nSeparationCounter--;

    // If Separation Timer is completed then switch State
    // to DEPLOY for the remainder of this pass and execute
    // the entry actions of the DEPLOY state
    if (nSeparationCounter < 1)
    {
        SetState(DEPLOY);
        // Open the Rotor Lock to Release the Rotor blades
        pcRotorServo->write(ROTOROPEN);
    }
}

```

```

        break;
    }
    case READYRELEASE:
    {
        if (!IsHostInterlockPresent())
        {
            // *****
            // The following assumes no power loss following
            // release need to change this as other functions
            // do account for such
            // *****
            // If Low means Interlock disconnected, the package
            // is falling!
            // Therefore:
            // a. change state to SEPARATION,
            // b. count down the Separation timer,
            // c. Enable the Quadrature Decoder ISRs,
            // d. stop the BIT Checks.

            SetState(SEPARATION);

            // Decrement the Separation timer,
            // each decrement ~= 100msec
            nSeparationCounter--;

            // Attach the ISR vector to the INT0 for Encoder ChA
            attachInterrupt(0, ChannelA_vect, RISING);
            // attachInterrupt(0, ChannelA_vect, CHANGE);
            // attachInterrupt(1, ChannelB_vect, RISING);
            // attachInterrupt(1, ChannelB_vect, CHANGE);

            // Stop the BIT Checks
            pcBIT->ModifyBITCheckFlag(false);
            break;
            // Do not need to undertake any further
            // READYRELEASE functionality
        }
        // Check if Commit to Release is not currently asserted
        // If not assert
        if(IsReleaseProhibited())
        {
            // Assert the Commit To Release Interlock
            CommitToRelease();
        }
        break;
    }
    case ABORT:
    {
        // Un-Assert the Commit To Release Interlock
        ProhibitRelease();
        // Reselect External Power Supply and determine if it is
        // available to supply power to the package as it may be

```

```

        // turned off by the host
        if (!SwitchPowerSupply(PWRSOURCE_EXT))
        {
            // Wait within the ABORT State for a maximum of
            // period before External power should become
            // available from Host
            if (nExtPowerDelay > MAXEXTSUPPLYWAIT)
            {
                // Timeout!
                nExtPowerDelay = 0;
                // Failure to transition to EXT power so set BIT
                // fail
                SetState(BIT_FAIL);
            }
            else
            {
                nExtPowerDelay++;
            }
        }
        else
        {
            // Reset the Supply delay switch over counter
            nExtPowerDelay = 0;
            // A successful switch to External supply was
            // achieved so transition to PREPARED State
            SetState(PREPARED);
        }
        break;
    }
case POWERDOWN:
{
    if (bPowerDownFirstPass)
    {
        detachInterrupt(0);

        // Report zero angular velocity
        for (uint8_t i = 0; i < CIRCBUF_SIZE; i++)
        {
            g_nOmegaCirBuf[i] = 0;
        }
        bPowerDownFirstPass = false;
    }
    // constantly switch to external to turn off Package.
    SwitchPowerSupply(PWRSOURCE_EXT);
    break;
}
default:
    break;
}
}

void Commander::Task5Hz(void)

```

```

{

    I2CMsgRx* psTempRxMsgBuffer;

    // Check for msg within FIFO
    // Use TIME_IMMEDIATE to prevent sleeping in this thread.
    psTempRxMsgBuffer = pcFIFO->waitData(TIME_IMMEDIATE);

    // Act on any received Message
    if (psTempRxMsgBuffer)
    {
        // Yes, one is available so fetch Message from the FIFO.
        sTempRxMsgBuffer = *psTempRxMsgBuffer;

        // Signal FIFO slot is free.
        pcFIFO->signalFree();

        // Act on the message
        ExecuteCmd(&sTempRxMsgBuffer);
    }

    switch(RetrieveState()) // In priority order
    {
        case READYCONFIRM:
        {
            // Un-Assert the Commit To Release Interlock
            ProhibitRelease();

            // Check if already switched to internal power supply
            if(ReadRecordedSupply() != PWRSOURCEINT)
            {
                // Request swap to Internal Power Supply, confirm
                // it is supplying power to the package
                if (!SwitchPowerSupply(PWRSOURCEINT))
                {
                    // Failure to transition to INT power
                    // so set BIT fail
                    SetState(BITFAIL);
                }
            }
            break;
        }
        case INITIALISE:
        {
            // Un-Assert the Commit To Release Interlock
            ProhibitRelease();

            // Allow the BIT Checks
            pcBIT->ModifyBITCheckFlag(true);

            // Should this be a full power on transition the
            // Commander Constructor has already undertaken the

```

```

        // following actions as part of initialisation
        // a. Power to External, and
        // b. Quadrature Decoder ISR(s) detached.
        if(IsDebugMode() || IsHostInterlockPresent())
        {
            // transition to PREPARED state
            SetState(PREPARED);
        }
        break;
    }
    case BITFAIL:
    {
        // Check if Commit to Release is currently asserted
        // If it is unassert it, Prohibit any release
        if(!IsReleaseProhibited())
        {
            // Un-Assert the Commit To Release Interlock
            ProhibitRelease();
        }
        break;
    }
    default:
        break;
}
}

void Commander::Task1Hz(void)
{

    static uint8_t Count1Hz;
    static bool bDebugFirstPass = true;

    if(RetrieveState() == PREPARED)
    {
        if(bDebugFirstPass)
        {
            bDebugFirstPass = false;
            Count1Hz = 0;
        }
    }
    if(RetrieveState() >= PREPARED)
    {
        Count1Hz++;
        if (Count1Hz == 4)
        {
            if(RetrieveState() == PREPARED)
            {
                SetState(READYCONFIRM);
                bDebugFirstPass = true;
            }
            else
                Count1Hz = 3;
        }
    }
}

```



```

    }
    if (Count1Hz == 8)
    {
        SetState(READYRELEASE);
        bDebugFirstPass = true;
    }
    if (Count1Hz == 60)
    {
        if(RetrieveState() == DEPLOY)
        {
            SetState(POWERDOWN);
            bDebugFirstPass = true;
        }
        else
        {
            Count1Hz = 10;
        }
    }
    if (Count1Hz > 61)
    {
        bDebugFirstPass = true;
        Count1Hz = 61;
    }
}
else
    Count1Hz = 0;

if(pcBIT->RetrieveBITCheckFlag())
{
    pcBIT->BITCheck();
}
UpdateDebugInterLockState();
}

void Commander::ExecuteCmd(I2CMsgRx* psRcvMsgBuffer)
{
    // Decode and react to Message No 2 through 6
    switch(psRcvMsgBuffer->g_nType1RxMsgNo)
    {
        case MSG2:
        {
            // Commanded State Change
            // Single data byte
            SetState((eSTATE)psRcvMsgBuffer->g_nType1RxEnumValue);
            break;
        }
        case MSG3:
        {
            // Save notified Weight Off Wheels Status
            UpdateWeightOffWheelsState(psRcvMsgBuffer->g_nType2RxEnumValue);
            break;
        }
    }
}

```

```
    }  
    case MSG4:  
    {  
        // Initialise the Internal Battery Capacity  
        SetupCapacityAhValue(psRcvMsgBuffer->g_nType4RxIntValue);  
        break;  
    }  
    case MSG5:  
    {  
        // Todo  
        break;  
    }  
    case MSG6:  
    {  
        // Debug Interface Messages  
        // Todo  
        break;  
    }  
    }  
}
```

H.6 The BIT.h Code

Listing H.4: The BIT header file.

```

/*
 * BIT.h
 *
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef BIT_H_
#define BIT_H_

enum eBITFLAG {
    SUPPLY_INT,           // BIT Pos 0
    SUPPLY_EXT,           // BIT Pos 1
    SUPPLY_OFF,           // BIT Pos 2
    CURRENT_FAULT,        // BIT Pos 3
    PACKAGE_INTERLOCK_GONE, // BIT Pos 4
    BATCAP_BELOW_WARN,    // BIT Pos 5
    FMU_NOT_COMM,         // BIT Pos 6
    INVALID_STATE_CHG,    // BIT Pos 7
    SUPPLY_NOT_CMD,        // BIT Pos 8
    RXBUFF_OVRFLW,        // BIT Pos 9
    FIFO_OVRFLW,          // BIT Pos 10
    HOOK_LOCKED,          // BIT Pos 11
    SPARE1,               // BIT Pos 12
    SPARE2,               // BIT Pos 13
    SPARE3,               // BIT Pos 14
    BIT_FLAG_OVRFLW       // BIT Pos 15
};

class BIT
{
public:
    BIT();
    void BITFlagUpdate(eBITFLAG nBITFlagPosition, bool bCondition);
    uint16_t ReadBITFlags() const;
    bool RetrieveBITCheckFlag(void) const;
    void ModifyBITCheckFlag(bool bFlag);
    bool BITCheck(void);

private:
    uint16_t nBITCondition;
    // Authority to undertake BIT checks
    bool bBITCheckAuthFlag;
};

#endif /* BIT_H_ */

```

H.7 The BIT.cpp Code

Listing H.5: The BIT source file.

```

/*
 * BIT.cpp
 *
 * Created on: 7 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include "BIT.h"
#include "Commander.h"
#include "PinoutConfigSM.h"

extern Commander* pcCmdr;
extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

// BIT Constructor
BIT::BIT()
{
    nBITCondition = 0;

    // Initialise Allow BIT Checking
    bBITCheckAuthFlag = true;
}

void BIT::BITFlagUpdate(eBITFLAG nBITFlagPosition, bool bCondition)
{
    // the flag to be shifted into BIT Flag position
    uint16_t nFlag = 1;

    // DEBUG print statements
    // Uncomment if required
    //Serial.print("BIT Pos ");
    //Serial.print(nBITFlagPosition);
    //Serial.print(" bCondition ");
    //Serial.println(bCondition);

    // Assert that nBITFlagPosition < 16 positions
    if (nBITFlagPosition > 15)
    {
        BITFlagUpdate(BIT_FLAG_OVRFLW, true);
        //pcCmdr->SetState(BITFAIL);
        return;
    }

    // Shift and Set or clear the BIT Flag position

```

```

    if(bCondition)
    {
        // The update is to record a BIT FAIL
        // Need to shift and OR Mask into place
        nBITCondition |= (nFlag << nBITFlagPosition);
        // Set State to BITFAIL
        pcCmdr->SetState(BITFAIL);
    }
    else
    {
        // The update is to record a BIT PASS
        // need to shift, Invert and AND Mask into place
        nBITCondition &= (~(nFlag << nBITFlagPosition));

        // Check if all BIT Flag positions are false
        if(!(nBITCondition & 0xFFFF))
        {
            // DEBUG print statements
            // Uncomment if required
            //Serial.print("All BIT clear ");

            // Now check if the system is already in BITFAIL State
            // If so then this allows the system to transition out
            // of BITFAIL
            if(pcCmdr->RetrieveState() == BITFAIL)
            {
                pcCmdr->SetState(INITIALISE);
            }
        }
    }

    // Update the I2C Tx Msg Buffer
    // First Invalidate current message area
    g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG4);

    g_uI2CUpdate_MsgTx.g_nType3UIntValue = nBITCondition;

    // DEBUG print statements
    // Uncomment if required
    //Serial.print("nBITCondition ");
    //Serial.println(g_uI2CUpdate_MsgTx.g_nType3UIntValue);

    // Store it into the I2C Message Area
    g_cI2C_MsgTx->putMessage(MSG4, &g_uI2CUpdate_MsgTx);

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG4);
}

uint16_t BIT::ReadBITFlags() const
{
    return nBITCondition;
}

```

```

}

bool BIT::RetrieveBITCheckFlag(void) const
{
    return bBITCheckAuthFlag;
}

void BIT::ModifyBITCheckFlag(bool bFlag)
{
    bBITCheckAuthFlag = bFlag;
}

bool BIT::BITCheck(void)
{
    // Execute BIT

    if(pcCmdr->RetrieveState() == BITFAIL)
    {
        // Un-Assert the Commit To Release Interlock
        pcCmdr->ProhibitRelease();

        // Switch to External power source
        if (pcCmdr->SwitchPowerSupply(PWRSOURCEEXT))
        {
            // Clear the External Power Transition BITFail Flag
            BITFlagUpdate(SUPPLYEXT, false);
        }

        // IF the power supply switch fails BIT will handle setting
        // the bit flag and already within BITFAIL state so no need
        // to change State. If the switch is successful and that
        // cleared the only bit Flag then BIT will transition
        // state to INITIALISE

        // Detach the Interrupts, necessary if transiting to BITFAIL
        // post ReadyRelease 10Hz Task
        detachInterrupt(0);
    //    detachInterrupt(1);

    }

    // Do other BIT checks

    // Check that the Current Sensor has not set the Over Current
    // FAULT flag. Current Sensor FAULT condition is Active LOW
    if(pcCmdr->ReadSensorFAULT())
    {

        Serial.println("CFAULT");

        // Fault exists so set the Current Fault BIT flag
        BITFlagUpdate(CURRENTFAULT, true);
    }
}

```

```

    }
    else
    {
        // No problem here so clear the Power BIT flag
        BITFlagUpdate(CURRENT_FAULT, false);
    }

    // Now check that the power supply outputs are as expected
    // Check if Internal power supply is turned on
    if (pcCmdr->ReadSupplyOutputState(PWRSOURCE_INT))
    {
        // Yes it is so Confirm the External is turned off
        if (!pcCmdr->ReadSupplyOutputState(PWRSOURCE_EXT))
        {
            // It is not on so Clear the External power BIT flag
            BITFlagUpdate(SUPPLY_EXT, false);
        }
        else
        {
            Serial.println("PWREXT_FAULT");
            // It is on so (somehow a dual power supply condition)
            // set the External power BIT flag
            BITFlagUpdate(SUPPLY_EXT, true);
        }
    }
    // else that means External must be turned on otherwise there
    // should be no supply and this can't be happening

    // Confirm the recorded power supply = commanded power supply
    if (pcCmdr->ReadRecordedSupply() != pcCmdr->ReadCommandedSupply())
    {
        Serial.println("PWR_RECORD_DIFF");
        // They don't equal so set the Power BIT flag
        BITFlagUpdate(SUPPLY_NOT_CMD, true);
    }
    else
    {
        // Clear the Supply Not as Commanded BIT flag
        BITFlagUpdate(SUPPLY_NOT_CMD, false);
    }

    // How to recover from FIFO Overflow and others

    // If any BIT flag set change state to BITFAIL
    if (nBITFaultFlags)
    {
        // Change SensorManager state to BITFAIL
        pcManagerState->SetState(BITFAIL);
    }
}

```

H.8 The I2CBuffer.h Code

Listing H.6: The I2CBuffer header file.

```

/*
 * I2CBuffer.h
 *
 * Created on: 31 Aug 2014
 * Author: Ian Saxby
 */

#ifndef I2CBUFFER_H_
#define I2CBUFFER_H_

#include <Arduino.h>
#include <stdint.h>
#include "StateMachine.h"
#include "Interface.h"

#define MAX_MSG_TX_LENGTH 5 // including wordcount as last element
#define MAX_MSG_RX_LENGTH 2
#define MAX_TX_BUFFERS 6
#define FIFO_DEPTH 2

#define SETUP_I2C_SEND_CMD 1

enum eMSG {MSG0, MSG1, MSG2, MSG3, MSG4, MSG5, MSG6};

typedef struct I2CMsgTx
{
    struct {
        bool bInvalid;
        uint8_t g_nMsgNo;
    };
    union
    {
        struct {
            uint8_t g_nType1EnumValue;
            uint8_t g_nType1Rem[MAX_MSG_TX_LENGTH - 1];
        };
        struct {
            int16_t g_nType2IntValue;
            uint8_t g_nType2Rem[MAX_MSG_TX_LENGTH - 2];
        };
        struct {
            uint16_t g_nType3UIntValue;
            uint8_t g_nType3Rem[MAX_MSG_TX_LENGTH - 2];
        };
        struct {
            eSTATE g_nType4EnumValue;
            uint8_t g_nType4Rem[MAX_MSG_TX_LENGTH - 1];
        };
    };
};

```



```

    };
    struct {
        bool      g_nType5BoolValue1;
        bool      g_nType5BoolValue2;
        bool      g_nType5BoolValue3;
        uint8_t   g_nType5EnumValue[MAX_MSG_TX_LENGTH - 3];
    };
    uint8_t g_nTxAllData [MAX_MSG_TX_LENGTH];
};
uint8_t nWdCount;
} I2CMsgTx;

typedef struct I2CMsgRx
{
    union {
        struct {
            uint8_t g_nType1RxMsgNo;
            uint8_t g_nType1RxEnumValue;
            uint8_t g_nType1RxRem;
        };
        struct {
            uint8_t g_nType2RxMsgNo;
            eWOFFW g_nType2RxEnumValue;
            uint8_t g_nType2RxRem;
        };
        struct {
            uint8_t g_nType3RxMsgNo;
            uint8_t g_nType3RxEnumValue;
            uint8_t g_nType3RxRem;
        };
        struct {
            uint8_t g_nType4RxMsgNo;
            uint16_t g_nType4RxIntValue;
        };
        uint8_t g_nRxAllData [MAX_MSG_RX_LENGTH];
    };
} I2CMsgRx;

class I2CBuffer
{
public:
    // Constructor
    I2CBuffer ();

    void I2CInvalidateTxMsg (eMSG eMsg);
    void I2CValidateTxMsg (eMSG eMsg);
    // Store Message into identified Message Buffer
    void putMessage (eMSG eMsg, I2CMsgTx *pMsgData);

    // Retrieve Message from identified Message Buffer
    void getTxMessage (uint8_t MsgNoTx, I2CMsgTx *pMsgOut);

```

```
private:

    I2CMsgTx sMsgTxArray[MAX_TX_BUFFERS];

};
#endif /* I2CBUFFER_H */
```

H.9 The I2CBuffer.cpp Code

Listing H.7: The I2CBuffer source file.

```

/*
 * I2CBuffer.cpp
 *
 * Created on: 31 Aug 2014
 * Author: Ian Saxby
 */

#include <stdio.h>
#include "I2CBuffer.h"

I2CBuffer::I2CBuffer()
{
    // Initialise the Output (TX) Message buffer area
    for (int MsgNo = 0; MsgNo < MAX_TX_BUFFERS; MsgNo++)
    {
        for (int indy = 0; indy < MAX_MSG_TX_LENGTH; indy++)
        {
            sMsgTxArray[MsgNo].g_nTxAllData[indy] = 0;
        }
        // Invalidate each message
        sMsgTxArray[MsgNo].bInvalid = true;

        // Initialise the Word count for each message

        switch (MsgNo)
        {
            case MSG0: // Sensor Manager State
            {
                sMsgTxArray[MSG0].nWdCount = 2;
                break;
            }
            case MSG1: // Quadrature Data
            {
                sMsgTxArray[MSG1].nWdCount = 3;
                break;
            }
            case MSG2: // Internal Battery Remaining Capacity
            {
                sMsgTxArray[MSG2].nWdCount = 3;
                break;
            }
            case MSG3: // Supply Source
            {
                sMsgTxArray[MSG3].nWdCount = 2;
                break;
            }
            case MSG4: // Built In Test Results

```

```

        {
            sMsgTxArray[MSG4].nWdCount = 3;
            break;
        }
        case MSG5:
        {
            sMsgTxArray[MSG5].nWdCount = 4;
            break;
        }
    }
}

void I2CBuffer::I2CInvalidateTxMsg(eMSG eMsg)
{
    sMsgTxArray[eMsg].bInvalid = true;
}

void I2CBuffer::I2CValidateTxMsg(eMSG eMsg)
{
    sMsgTxArray[eMsg].bInvalid = false;
}

// Store Message into identified Message Buffer
void I2CBuffer::putMessage(eMSG eMsg, I2CMsgTx *pMsgIn)
{
    for(uint8_t indx = 1; indx < MAXMSG_TXLENGTH; indx++)
    {
        sMsgTxArray[eMsg].g_nTxAllData[indx] =
            (*pMsgIn).g_nTxAllData[indx];
    }
}

// Retrieve Message from identified Message Buffer into local
// buffer for I2C
void I2CBuffer::getTxMessage(uint8_t MsgNoTx, I2CMsgTx *pMsgOut)
{
    (*pMsgOut) = sMsgTxArray[MsgNoTx];
}

```

H.10 The Interface.h Code

Listing H.8: The Interface header file.

```

/*
 * Interface.h
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef INTERFACE_H_
#define INTERFACE_H_

// Separation counter for 1 second at 10 * Task10Hz
#define SEPARATIONCOUNT 10
// Rotor Servo Open angle in degrees
#define ROTOROPEN 2200
// Rotor Servo Close angle in degrees
#define ROTORCLOSE 1200
// Define Abort delay in 10Hz iterations
#define MAXEXTSUPPLYWAIT 20 // = 2 seconds

enum eWOFFW {WONW, WOFFW};

class Interface
{
public:

    Interface();
    bool IsDebugMode(void) const;
    // Detect the Debug Interlock state (Active LOW) and update
    // Mode flag
    bool UpdateDebugInterLockState(void);
    // Output the Host Interlock wrap signal at state (Active LOW)
    // and update flag
    void OutputHostInterlockSignal(void);

    // Check if Host Interlock is present (still LOW)
    bool IsHostInterlockPresent(void);
    void CommitToRelease(void);
    bool IsReleaseProhibited() const;
    void ProhibitRelease(void);
    void UpdateWeightOffWheelsState(eWOFFW bWoffW);
    bool IsWeightOffWheels(void) const;
    virtual ~Interface() {};

private:

    void UpdateI2CMsg(void);

```

```
eWOFFW eWOffWStatus;  
  
bool bDebugMode;  
  
bool bReleaseProhibited;  
  
bool bHostInterLockPresent;  
  
};  
  
#endif /* INTERFACE_H */
```

H.11 The Interface.cpp Code

Listing H.9: The Interface source file.

```

/*
 * Interface.cpp
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#include <Arduino.h>
#include "Interface.h"
#include "PinoutConfigSM.h"
#include "I2CBuffer.h"

extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

Interface::Interface()
{
    eWOffWStatus = WQW;

    UpdateDebugInterLockState();

    bReleaseProhibited = true;

    IsHostInterlockPresent();
}

bool Interface::IsDebugMode(void) const
{
    return bDebugMode;
}

bool Interface::UpdateDebugInterLockState(void)
{
    // Return the Debug Input Interlock state
    // This pin is active LOW using pull-up resistors
    if(digitalRead(PIN_PKG_DEBUG))
    {
        // Input is HIGH so Debug mode IS NOT enabled
        bDebugMode = false;
    }
    else
    {
        // Input is LOW so Debug mode IS enabled
        bDebugMode = true;
    }
}

```

```
    UpdateI2CMsg();
    return bDebugMode;
}

void Interface::OutputHostInterlockSignal(void)
{
    // This pin is active LOW as is using pull-up resistors
    digitalWrite(PIN_PKG_INTERLOCK_OUT, LOW);
}

bool Interface::IsHostInterlockPresent(void)
{
    // This pin is active LOW as is using pull-up resistors
    if(digitalRead(PIN_PKG_INTERLOCK_IN))
    {
        bHostInterLockPresent = false;
    }
    else
    {
        // Input is LOW so Host Interlock IS present
        bHostInterLockPresent = true;
    }

    UpdateI2CMsg();

    return bHostInterLockPresent;
}

void Interface::CommitToRelease(void)
{
    // Set Active Low Commit To Release Interlock Output
    digitalWrite(PIN_COMMIT_RELEASE, LOW);

    // Clear Release Prohibited flag to indicate interlock
    // IS asserted
    bReleaseProhibited = false;

    UpdateI2CMsg();
}

bool Interface::IsReleaseProhibited(void) const
{
    return bReleaseProhibited;
}

void Interface::ProhibitRelease(void)
{
    // Clear Active Low Commit To Release Interlock Output
    digitalWrite(PIN_COMMIT_RELEASE, HIGH);

    // Set Release Prohibited flag to indicate interlock NOT
    // asserted
    bReleaseProhibited = true;
}
```



```

        UpdateI2CMsg();
    }

    void Interface::UpdateWeightOffWheelsState(eWOFFW bWoffW)
    {
        eWOffWStatus = bWoffW;
    }

    bool Interface::IsWeightOffWheels(void) const
    {
        if (eWOffWStatus == WOFFW)
            return true;
        else
            return false;
    }

    void Interface::UpdateI2CMsg(void)
    {
        // Update the I2C Tx Msg Buffer
        // First Invalidate current message area
        g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG5);

        g_uI2CUpdate_MsgTx.g_nType5BoolValue1 = bHostInterLockPresent;
        // Transmit the Debug state
        g_uI2CUpdate_MsgTx.g_nType5BoolValue2 = bDebugMode;

        // Transmit the Commit To Release Status (inverted Release
        // Prohibited status)
        g_uI2CUpdate_MsgTx.g_nType5BoolValue3 = !bReleaseProhibited;

        // Store it into the I2C Message Area
        g_cI2C_MsgTx->putMessage(MSG5, &g_uI2CUpdate_MsgTx);

        // Finally Validate Msg area
        g_cI2C_MsgTx->I2CValidateTxMsg(MSG5);
    }

```

H.12 The PinoutConfigSM.h Code

Listing H.10: The PinoutConfigSM header file.

```

/*
 * PinoutConfigSM.h
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef PINOUTCONFIGSM_H
#define PINOUTCONFIGSM_H

// Definition of Sensor Manager Pro Mini Pinout Assignments
// Rotor Lock Release
#define PIN_ROTORLOCKPWM 6

// Channel A is INT0
#define PIN_CHANNEL_A 2
// Channel B is INT1
#define PIN_CHANNEL_B 3

// External Power Enable
#define PIN_POWERENB_EXT 4
// Internal Power Enable
#define PIN_POWERENB_INT 5

// Host Interconnect wrap Output
#define PIN_PKG_INTERLOCK_OUT 7
// Commit to Release
#define PIN_COMMIT_RELEASE 8
// Host Interconnect wrap Input
#define PIN_PKG_INTERLOCK_IN 11
// Debug Input
#define PIN_PKG_DEBUG 14

// Current Sensor Fault Input
#define PIN_CURRENTSENSORFAULT 15
// External Power State Status ONST1 Input
#define PIN_PWRSTATUS_EXT_ONST1 16
// Internal Power State Status ONST2 Input
#define PIN_PWRSTATUS_INT_ONST2 17

// Current Sensor Value input analog
#define PIN_VIOUTSENSE A7
// Voltage Supply Sense input analog
#define PIN_VOLTAGESENSE A6

#endif /* PINOUTCONFIGSM_H */

```

H.13 The Power.h Code

Listing H.11: The Power header file.

```

/*
 * Power.h
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef POWER_H
#define POWER_H

#include <Arduino.h>
#include <stdint.h>
#include "QuadEncoder.h"

// Power Controller is Active Low
#define POWERON LOW
#define POWEROFF HIGH

// Constants defining Power Cmds for External and Internal supply
// selection
enum ePWRSUPPLY {PWRSOURCEEXT, PWRSOURCEINT, PWRSOURCEOFF};

class Power
{
public:
    Power();
    void SetupCapacityAhValue(uint16_t nValue);
    void UpdateInstantCurrent(void);
    void UpdateInstantVoltage(void);
    bool SwitchPowerSupply(ePWRSUPPLY eRequestedSupply);
    uint16_t ReadCapacityRemain(void) const;
    uint16_t ReadInstantVoltage(void) const;
    bool ReadSupplyOutputState(ePWRSUPPLY eSupply) const;
    ePWRSUPPLY ReadCommandedSupply(void) const;
    ePWRSUPPLY ReadRecordedSupply(void) const;
    bool ReadSensorFAULT(void) const;
    virtual ~Power(){}

private:

    // Initial Current Rating
    uint16_t nSetupCapacityAhRating;

    // Power Statistics
    uint16_t nRemainingAhCapacity;
    uint16_t nInstantCurrent;

```

```
uint16_t  nInstantVoltage;
uint32_t  tPrevTimeSense;

// Next two variables relate to Power Source
// enumerated variable contents
ePWRSUPPLY ePowerCommand;
// ePowerSource reflects actual Power Source
ePWRSUPPLY ePowerSource;

// ACS711 Current Sensor Over Current Fault input
// bool bSensorFAULT;

};

#endif  /* POWER_H */
```

H.14 The Power.cpp Code

Listing H.12: The Power source file.

```

/*
 * Power.cpp
 *
 * Created on: 7 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include <NilAnalog.h>
#include "Power.h"
#include "PinoutConfigSM.h"
#include "BIT.h"
#include "I2CBuffer.h"

extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

// Power Constructor
Power::Power()
{
    nSetupCapacityAhRating = 0;

    // Power Statistics
    nRemainingAhCapacity = 0;
    nInstantVoltage = 0;

    // Do not care about response just request internal power supply on
    SwitchPowerSupply(PWRSOURCEINT);
}

void Power::SetupCapacityAhValue(uint16_t nValue)
{
    nSetupCapacityAhRating = nValue;
}

void Power::UpdateInstantCurrent(void)
{
    // This is coded to match the calling timing of 1Hz
    // This functionality needs considerable verification before
    // utilisation

    uint32_t tCurrentSenseNow;
    uint16_t nVIOUT;

    // Determine time of Sense
    tCurrentSenseNow = micros();

```

```

    // Will assume that the sensed current is the same for the
    // preceding delta time period. Don't need to average it.
    nVIOUT = (uint16_t)nilAnalogRead((char)PIN_VIOUTSENSE);

    // Determine Amp usage

    // using ACS 711 equation the mV / Amp calculation is
    //  $V_{IOUT} = (0.11 * i - (V_{cc}/2)) * V_{cc} * 3.3V$ 
    // with Vcc at 5V
    nInstantCurrent = (nVIOUT * 0.0726) - 0.275;

    // this is missing the * dt * portion of calculation

    // Now calculate Capacity Remaining
    nRemainingAhCapacity = nRemainingAhCapacity - (nInstantCurrent *
        (tCurrentSenseNow - tPrevTimeSense)) >> 12;

    tPrevTimeSense = tCurrentSenseNow;
}

void Power::UpdateInstantVoltage(void)
{
    nInstantVoltage = (uint16_t)nilAnalogRead((char)PIN_VOLTAGESENSE);
}

bool Power::SwitchPowerSupply(ePWRSUPPLY eRequestedSupply)
{
    bool bSwitchSuccess;

    // Record the Power Command for BIT purposes
    ePowerCommand = eRequestedSupply;

    switch(ePowerCommand)
    {
        case PWRSOURCEEXT: // IF EXTERNAL Power Commanded
        {
            // Set the External Power Enable output
            digitalWrite(PIN_POWERENB_EXT, POWERON);

            // Confirm requested power source is now on
            bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEEXT);

            // If Requested source is supplying response should be true
            // then turn off alternate
            if (bSwitchSuccess)
            {
                // Turn off Internal Power Supply
                digitalWrite(PIN_POWERENB_INT, POWEROFF);

                // Confirm Internal Supply is turned off
                bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEINT);
            }
        }
    }
}

```

```

        // Alternate supply should be off so response should be
        // false
        if (!bSwitchSuccess)
        {
            // Record the Power Command for BIT purposes
            ePowerCommand = eRequestedSupply;
            ePowerSource = ePowerCommand;
            bSwitchSuccess = true;
            //return true;
        }
        else
        {
            bSwitchSuccess = false;
        }
        // Debug line
        //bSwitchSuccess = true;
    }
    else
    {
        bSwitchSuccess = false;
    }
    // Debug line
    //bSwitchSuccess = true;
}
break;
}
case PWRSOURCEINT:    // INTERNAL Power Commanded
{
    // Output the Internal Power Enable (Active Low)
    digitalWrite(PIN_POWERENB_INT, POWERON);

    //Confirm requested power source is now on
    bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEINT);

    // If Requested source is supplying response should be true
    // then turn off alternate
    if (bSwitchSuccess)
    {
        // Turn off External Power Supply
        digitalWrite(PIN_POWERENB_EXT, POWEROFF);

        // Confirm External Supply is turned off
        bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEEXT);

        // Alternate supply should be off so response should be
        // false
        if (!bSwitchSuccess)
        {
            // Record the Power Command for BIT purposes
            ePowerCommand = eRequestedSupply;
            ePowerSource = ePowerCommand;
            bSwitchSuccess = true;
            //return true;
        }
    }
}

```

```

        }
        else
        {
            bSwitchSuccess = false;
// Debug line
//bSwitchSuccess = true;
        }
    }
    else
    {
        bSwitchSuccess = false;

// Debug line
//bSwitchSuccess = true;
    }
    break;
}
case PWRSOURCEOFF:
{
    // Turn off Measured Supply
    // Output the Internal Power Enable (Active Low)
    digitalWrite(PIN.POWERENB.INT, POWEROFF);

    // Output the External Power Enable (Active Low)
    digitalWrite(PIN.POWERENB.EXT, POWEROFF);

    //Confirm power sources are both now off
    // Active LOW, so should be HIGH input
    bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEOFF);

    if(bSwitchSuccess)
    {
        // Record the Power Command for BIT purposes
        ePowerCommand = PWRSOURCEOFF;
        ePowerSource = ePowerCommand;
    }

    break;
}
}

// Update the I2C Tx Message Buffer with current Power State
// First Invalidate current message area
g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG3);

g_uI2CUpdate_MsgTx.g_nType1EnumValue = (uint8_t)ePowerSource;

// Now save into I2C Tx Message Common Area
g_cI2C_MsgTx->putMessage(MSG3, &g_uI2CUpdate_MsgTx);

// Finally Validate Msg area

```



```

    g_cI2C_MsgTx->I2CValidateTxMsg(MSG3);

    return bSwitchSuccess;
}

uint16_t Power::ReadCapacityRemain(void) const
{
    return nRemainingAhCapacity;
}

uint16_t Power::ReadInstantVoltage(void) const
{
    return nInstantVoltage;
}

bool Power::ReadSupplyOutputState(ePWRSUPPLY eSupply) const
{
    bool bDigitalState = false;

    switch (eSupply)
    {
        case PWRSOURCE_EXT:
        {
            // Power is ON when Active LOW output on ONST1
            if(digitalRead(PIN_PWRSTATUS_EXT_ONST1) == 0)
                bDigitalState = true;
            break;
        }
        case PWRSOURCE_INT:
        {
            // Power is ON when Active LOW output on ONST2
            if(digitalRead(PIN_PWRSTATUS_INT_ONST2) == 0)
                bDigitalState = true;
            break;
        }
        case PWRSOURCE_OFF:
        {
            // Power is ON when Active LOW output either ONST1 or ONST2
            if(digitalRead(PIN_PWRSTATUS_EXT_ONST1) == 0)
                bDigitalState = false;
            else if(digitalRead(PIN_PWRSTATUS_INT_ONST2) == 0)
                bDigitalState = false;
            else
                // Both are turned off so response success
                bDigitalState = true;

            break;
        }
    }
    return bDigitalState;
}

```

```
bool Power::ReadSensorFAULT(void) const
{
    // Current Sensor FAULT condition is Active LOW
    if(!digitalRead(PIN_CURRENTSENSORFAULT))
    {
        // Result is LOW so fault exists, set response = true
        return true;
    }
    else
    {
        // Result is HIGH so NO fault exists, set response = false
        return false;
    }
}

ePWRSUPPLY Power::ReadCommandedSupply(void) const
{
    return ePowerCommand;
}

ePWRSUPPLY Power::ReadRecordedSupply(void) const
{
    return ePowerSource;
}
```

H.15 The QuadEncoder.h Code

Listing H.13: The QuadEncoder header file.

```

/*
 * QuadEncoder.h
 *
 * Created on: 28 Aug 2014
 * Author: Ian Saxby
 */

#ifndef QUADENCODER_H
#define QUADENCODER_H

// Encoder Tsc period constant (usec)
#define TSC_PERIOD 98304 // closest to 100msec using scaling 3 << 15
// Define Np divisor 32 => 5 (by number of shift right 2^5)
// #define NP 5
// Define Np divisor 16 => 4 (by number of shift right 2^4)
#define NP 4
// Number of Quadrature Decoder Buffers, used for concurrent w calcs
// and isr execution
#define NUMDECBUF 2
// Circular Buffer Size for averaging filter of Quadrature Encoder
// Angular rate output
#define CIRCBUF_SIZE 2
// Define the maximum duration of the extended observation window
// before angular rate is marked as zero rpm
#define MAXEXTOWSWIN 15 // which is about 1.5 seconds

// Maximum Quadrature Rotor output velocity
#define MAXLIMIT_OMEGA 10000

// Quadrature Decoder Timer record structure definition
typedef union
{
    struct {
        uint16_t spare;
        uint16_t n16DeltaN;
    };
    uint32_t n32DeltaN;
} c_UDelta;

typedef struct
{
    c_UDelta u_nDeltaN;
    uint32_t DecoderISRTime;
    uint32_t nDeltaThml = 0;
} DecoderBuffer;

typedef union

```

```
{
    struct {
        uint16_t spare;
        uint16_t n16SumTscacc;
    };
    uint32_t n32SumTscacc;
} c_UTscacc;

// Quadrature Decoder Tsc,acc global count variable
static uint16_t g_nTscTick = 0;

// 2 element Quadrature Decoder Timer Buffer declaration
static DecoderBuffer g_DecBuf[NUMDECBUF];
// Index into above buffer initialised to 0
static uint8_t g_nBufferIndx = 0;
// Circular buffer for Omega average
//static uint16_t g_nOmegaCirBuf[CIRCBUFSIZE];
// Circular Buffer index
static uint8_t g_nOCirIndx = 0;

#endif /* QUADENCODER_H */
```

H.16 The BIT.cpp Code

Listing H.14: The BIT source file.

```

/*
 * BIT.cpp
 *
 * Created on: 7 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include "BIT.h"
#include "Commander.h"
#include "PinoutConfigSM.h"

extern Commander* pcCmdr;
extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

// BIT Constructor
BIT::BIT()
{
    nBITCondition = 0;

    // Initialise Allow BIT Checking
    bBITCheckAuthFlag = true;
}

void BIT::BITFlagUpdate(eBITFLAG nBITFlagPosition, bool bCondition)
{
    // the flag to be shifted into BIT Flag position
    uint16_t nFlag = 1;

    // DEBUG print statements
    // Uncomment if required
    //Serial.print("BIT Pos ");
    //Serial.print(nBITFlagPosition);
    //Serial.print(" bCondition ");
    //Serial.println(bCondition);

    // Assert that nBITFlagPosition < 16 positions
    if (nBITFlagPosition > 15)
    {
        BITFlagUpdate(BIT_FLAG_OVRFLW, true);
        //pcCmdr->SetState(BIT_FAIL);
        return;
    }

    // Shift and Set or clear the BIT Flag position

```

```

    if(bCondition)
    {
        // The update is to record a BIT FAIL
        // Need to shift and OR Mask into place
        nBITCondition |= (nFlag << nBITFlagPosition);
        // Set State to BITFAIL
        pcCmdr->SetState(BITFAIL);
    }
    else
    {
        // The update is to record a BIT PASS
        // need to shift, Invert and AND Mask into place
        nBITCondition &= (~(nFlag << nBITFlagPosition));

        // Check if all BIT Flag positions are false
        if(!(nBITCondition & 0xFFFF))
        {
            // DEBUG print statements
            // Uncomment if required
            //Serial.print("All BIT clear ");

            // Now check if the system is already in BITFAIL State
            // If so then this allows the system to transition out
            // of BITFAIL
            if(pcCmdr->RetrieveState() == BITFAIL)
            {
                pcCmdr->SetState(INITIALISE);
            }
        }
    }

    // Update the I2C Tx Msg Buffer
    // First Invalidate current message area
    g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG4);

    g_uI2CUpdate_MsgTx.g_nType3UIntValue = nBITCondition;

    // DEBUG print statements
    // Uncomment if required
    //Serial.print("nBITCondition ");
    //Serial.println(g_uI2CUpdate_MsgTx.g_nType3UIntValue);

    // Store it into the I2C Message Area
    g_cI2C_MsgTx->putMessage(MSG4, &g_uI2CUpdate_MsgTx);

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG4);
}

uint16_t BIT::ReadBITFlags() const
{
    return nBITCondition;
}

```

```

}

bool BIT::RetrieveBITCheckFlag(void) const
{
    return bBITCheckAuthFlag;
}

void BIT::ModifyBITCheckFlag(bool bFlag)
{
    bBITCheckAuthFlag = bFlag;
}

bool BIT::BITCheck(void)
{
    // Execute BIT

    if(pcCmdr->RetrieveState() == BITFAIL)
    {
        // Un-Assert the Commit To Release Interlock
        pcCmdr->ProhibitRelease();

        // Switch to External power source
        if (pcCmdr->SwitchPowerSupply(PWRSOURCEEXT))
        {
            // Clear the External Power Transition BITFail Flag
            BITFlagUpdate(SUPPLYEXT, false);
        }

        // IF the power supply switch fails BIT will handle setting
        // the bit flag and already within BITFAIL state so no need
        // to change State. If the switch is successful and that
        // cleared the only bit Flag then BIT will transition
        // state to INITIALISE

        // Detach the Interrupts, necessary if transiting to BITFAIL
        // post ReadyRelease 10Hz Task
        detachInterrupt(0);
    //    detachInterrupt(1);

    }

    // Do other BIT checks

    // Check that the Current Sensor has not set the Over Current
    // FAULT flag. Current Sensor FAULT condition is Active LOW
    if(pcCmdr->ReadSensorFAULT())
    {

        Serial.println("CFAULT");

        // Fault exists so set the Current Fault BIT flag
        BITFlagUpdate(CURRENTFAULT, true);
    }
}

```

```

    }
    else
    {
        // No problem here so clear the Power BIT flag
        BITFlagUpdate(CURRENT_FAULT, false);
    }

    // Now check that the power supply outputs are as expected
    // Check if Internal power supply is turned on
    if (pcCmdr->ReadSupplyOutputState(PWR_SOURCE_INT))
    {
        // Yes it is so Confirm the External is turned off
        if (!pcCmdr->ReadSupplyOutputState(PWR_SOURCE_EXT))
        {
            // It is not on so Clear the External power BIT flag
            BITFlagUpdate(SUPPLY_EXT, false);
        }
        else
        {
            Serial.println("PWREXT_FAULT");
            // It is on so (somehow a dual power supply condition)
            // set the External power BIT flag
            BITFlagUpdate(SUPPLY_EXT, true);
        }
    }
    // else that means External must be turned on otherwise there
    // should be no supply and this can't be happening

    // Confirm the recorded power supply = commanded power supply
    if (pcCmdr->ReadRecordedSupply() != pcCmdr->ReadCommandedSupply())
    {
        Serial.println("PWR_RECORD_DIFF");
        // They don't equal so set the Power BIT flag
        BITFlagUpdate(SUPPLY_NOT_CMD, true);
    }
    else
    {
        // Clear the Supply Not as Commanded BIT flag
        BITFlagUpdate(SUPPLY_NOT_CMD, false);
    }

    // How to recover from FIFO Overflow and others

    // If any BIT flag set change state to BITFAIL
    if (nBITFaultFlags)
    {
        // Change SensorManager state to BITFAIL
        pcManagerState->SetState(BITFAIL);
    }
}

```


H.17 The ServoTimer2.h Code

The function `ServoTimer2.h` is included to show how to include C source code. Again, you might need an explanation of *what* it does and *how* it does it here.

Listing H.15: The ServoTimer2 header file.

```

/*
  ServoTimer2.h – Interrupt driven Servo library for Arduino using
  Timer2– Version 0.1
  Copyright (c) 2008 Michael Margolis. All right reserved.
  Modified by Ian Saxby to work with NiRTOS

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301
  USA
  */

/*
  This library uses Timer2 to drive up to (8) now 4 servos using
  interrupts so no refresh activity is required from within the sketch.
  The usage and method naming is similar to the Arduino software servo
  library
  http://www.arduino.cc/playground/ComponentLib/Servo
  except that pulse widths can be in microseconds or degrees.
  write() treats parameters of 180 or less as degrees, otherwise values
  are milliseconds.

  A servo is activated by creating an instance of the Servo class passing
  the desired pin to the attach() method. The servo is pulsed in the
  background to the value most recently written using the write() method

  Note that analogWrite of PWM on pins 3 and 11 is disabled when the first
  servo is attached

  The methods are:

```

ServoTimer2 – Class for manipulating servo motors connected to Arduino pins.

attach(pin) – Attaches a servo motor to an i/o pin.

attach(pin, min, max) – Attaches to a pin setting min and max values in microseconds

default min is 544, max is 2400

write() – Sets the servo pulse width in microseconds.

read() – Gets the last written servo pulse width in microseconds.

attached() – Returns true if there is a servo attached.

detach() – Stops an attached servos from pulsing its i/o pin.

The library takes about 824 bytes of program memory and 32+(2*servos) bytes of SRAM.

The pulse width timing is accurate to within 1%

*/

// ensure this library description is only included once

#ifndef ServoTimer2.h

#define ServoTimer2.h

#include <Arduino.h>

#include <inttypes.h>

// the shortest pulse sent to a servo

#define MIN_PULSE_WIDTH 750

// the longest pulse sent to a servo

#define MAX_PULSE_WIDTH 2250

// default pulse width when servo is attached

#define DEFAULT_PULSE_WIDTH 1500

// the maximum number of channels, don't change this

#define NBR_CHANNELS 4

// frame sync delay is the first entry in the channel array

#define FRAME_SYNC_INDEX 0

// total frame duration in microseconds

#define FRAME_SYNC_PERIOD 20000

// number of iterations of the ISR to get the desired frame rate

#define FRAME_SYNC_DELAY ((FRAME_SYNC_PERIOD - (\

NBR_CHANNELS * DEFAULT_PULSE_WIDTH))/ 128)

// number of microseconds of calculation overhead to be

//subtracted from pulse timings

#define DELAY_ADJUST 8

// Tradeoff of higher speed vice higher use of memory

typedef struct {

```

    // a pin number from 0 to 31
    uint8_t nbr ;
    // false if this channel not enabled, pin only pulsed if true
    bool isActive;
} ServoPin_t;

typedef struct {
    ServoPin_t Pin;
    uint8_t counter;
    uint8_t remainder;
    int nPulseWidthMin;
    int nPulseWidthMax;
} servo_t;

class ServoTimer2
{
public:
    // constructor:
    ServoTimer2();
    // attach the given pin to the next free channel, sets pinMode,
    // returns channel number or 0 if failure
    uint8_t attach(int);
    // the attached servo is pulsed with the current pulse width value,
    //(see the write method)
    // as above but also sets min and max values for writes.
    uint8_t attach(int, int, int);
    void detach();
    // store the pulse width in microseconds (between MIN_PULSE_WIDTH
    // and MAX_PULSE_WIDTH) for this channel
    void write(int);
    // returns current pulse width in microseconds for this servo
    int read();
    // return true if this servo is attached
    bool attached();

private:
    // index into the channel data for this servo
    uint8_t chanIndex;
};

#endif

```

H.18 The ServoTimer2.cpp Code

Listing H.16: The ServoTimer2 source file.

```

/*
ServoTimer2.cpp – Modified by Ian Saxby for use with NilRTOS

Interrupt driven Servo library for Arduino using Timer2– Version 0.1
Copyright (c) 2008 Michael Margolis. All right reserved.

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110–1301
USA
*/
#include <Arduino.h>

#include "ServoTimer2.h"
static void initISR();
//static void writeChan(uint8_t chan, int pulsewidth);
// static array holding servo data for all channels
static servo_t servos[NBR_CHANNELS+1];
// counter holding the channel being pulsed
static volatile uint8_t Channel;
// iteration counter used in the interrupt routines;
static volatile uint8_t ISRCount;
// counter holding the number of attached channels
uint8_t ChannelCount = 0;
// flag to indicate if the ISR has been initialised
static bool isStarted = false;

ISR(TIMER2_OVF_vect)
{
    // increment the overflow counter
    ++ISRCount;
    // are we on the final iteration for this channel
    if (ISRCount == servos[Channel].counter )
    {
        // yes, set count for overflow after remainder ticks
        TCNT2 = servos[Channel].remainder;
    }
}

```

```

    }
    else if (ISRCCount > servos[Channel].counter)
    {
        // we have finished timing the channel so pulse it low and
        // move on
        if (servos[Channel].Pin.isActive)
        {
            // check if activated
            // pulse this channel low if active
            digitalWrite( servos[Channel].Pin.nbr ,LOW);
        }

        Channel++; // increment to the next channel
        ISRCCount = 0; // reset the isr iteration counter
        TCNT2 = 0; // reset the clock counter register
        if ((Channel != FRAME_SYNC_INDEX) && (Channel <= NBR_CHANNELS))
        {
            // check if we need to pulse this channel
            if (servos[Channel].Pin.isActive)
            {
                // check if activated
                // its an active channel so pulse it high
                digitalWrite( servos[Channel].Pin.nbr ,HIGH);
            }
        }
        else
        {
            if (Channel > NBR_CHANNELS)
            {
                // all done so start over
                Channel = 0;
            }
        }
    }
}

ServoTimer2::ServoTimer2()
{
    if ( ChannelCount < NBR_CHANNELS)
    {
        // assign a channel number to this instance
        this->chanIndex = ++ChannelCount;
    }
    else
    {
        // too many channels, assigning 0 inhibits this instance from
        // functioning
        // todo
        this->chanIndex = 0;
    }
}

```

```

uint8_t ServoTimer2::attach(int pinNo)
{
    return this->attach(pinNo, MIN_PULSE_WIDTH, MAX_PULSE_WIDTH);
}

uint8_t ServoTimer2::attach(int pinNo, int nSetPulseWidthMin,
                             int nSetPulseWidthMax)
{
    if(!isStarted)
    {
        initISR();
    }

    if(this->chanIndex > 0)
    {
        //debug("attaching chan = ", chanIndex);
        // set servo pin to output
        pinMode(pinNo, OUTPUT);
        servos[this->chanIndex].Pin.nbr = pinNo;

        servos[this->chanIndex].Pin.isActive = true;

        servos[this->chanIndex].nPulseWidthMin = nSetPulseWidthMin;
        servos[this->chanIndex].nPulseWidthMax = nSetPulseWidthMax;
    }

    return this->chanIndex;
}

void ServoTimer2::detach()
{
    servos[this->chanIndex].Pin.isActive = false;
}

void ServoTimer2::write(int pulsewidth)
{
    if( pulsewidth < (servos[this->chanIndex].nPulseWidthMin) )
    {
        pulsewidth = servos[this->chanIndex].nPulseWidthMin;
    }
    else
    {
        if( pulsewidth > (servos[this->chanIndex].nPulseWidthMax) )
        {
            pulsewidth = servos[this->chanIndex].nPulseWidthMax;
        }
    }

    // subtract the time it takes to process the start and end pulses
    // (mostly from digitalWrite)
    pulsewidth -= DELAY_ADJUST;

```

```

servos[this->chanIndex].counter = pulsewidth >> 7;
// the number of 0.5us ticks for timer overflow
servos[this->chanIndex].remainder = 255 - ((pulsewidth -
    (servos[this->chanIndex].counter << 7)) << 1);
}

int ServoTimer2::read()
{
    unsigned int pulsewidth;
    if( this->chanIndex > 0)
    {
        pulsewidth = (servos[this->chanIndex].counter << 7) +
            ((255 - servos[this->chanIndex].remainder) >> 1) +
            DELAY_ADJUST ;
    }
    else
    {
        pulsewidth = 0;
    }
    return pulsewidth;
}

bool ServoTimer2::attached()
{
    return servos[this->chanIndex].Pin.isActive ;
}

static void initISR()
{
    int pulsewidth;
    for(uint8_t i=1; i <= NBR_CHANNELS; i++)
    {
        // channels start from 1
        // subtract the time it takes to process the start and end
        // pulses(mostly from digitalWrite)
        pulsewidth = DEFAULT_PULSE_WIDTH - DELAY_ADJUST;
        servos[i].counter = pulsewidth >> 7;
        // the number of 0.5us ticks for timer overflow
        servos[i].remainder = 255 - ((pulsewidth -
            (servos[i].counter << 7)) << 1);
    }
    // store the frame sync period
    servos[FRAME_SYNC_INDEX].counter = FRAME_SYNC_DELAY;

    Channel = 0; // clear the channel index
    ISRCnt = 0; // clear the value of the ISR counter;

    /* setup for timer 2 */
    // disable interrupts
    TIMSK2 = 0;
    // normal counting mode
    TCCR2A = 0;

```

```
// set prescaler of 8
TCCR2B = _BV(CS21);
// clear the timer2 count
TCNT2 = 0;
// clear pending interrupts;
TIFR2 = _BV(TOV2);
// enable the overflow interrupt
TIMSK2 = _BV(TOIE2) ;
// flag to indicate this initialisation code has been executed
isStarted = true;
}
```


H.19 The StateMachine.h Code

The function `StateMachine.h` is included to show how to include C source code. Again, you might need an explanation of *what* it does and *how* it does it here.

Listing H.17: The StateMachine header file.

```

/*
 * StateMachine.h
 *
 * Created on: 2 Nov 2014
 * Author: Ian
 */

#ifndef STATEMACHINE_H
#define STATEMACHINE_H

// #include "I2CBuffer.h"

#define NUMSTATES 9

enum eSTATE {INITIALISE,
              PREPARED,
              BITFAIL,
              POWERDOWN,
              ABORT,
              READYCONFIRM,
              READYRELEASE,
              SEPARATION,
              DEPLOY};

class StateMachine
{
public:
    // Constructor
    StateMachine();
    eSTATE RetrieveState(void) const;
    void SetState(eSTATE eNewState);
    virtual ~StateMachine(){}

private:
    eSTATE eCurrentState;
    // Adjacency List for Sensor Manager
    uint16_t abStateAdj[NUMSTATES];
    // Setup a buffer for storing I2C Tx data for
    // storing I2C Slave Transmission Message Buffer
    // I2CMsgTx uI2CUpdate_MsgTx;
};

```

```
#endif /* STATEMACHINE_H */
```

H.20 The StateMachine.cpp Code

Listing H.18: The StateMachine source file.

```

/*
 * StateMachine.cpp
 *
 * Created on: 2 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include "StateMachine.h"
#include "BIT.h"
#include "I2CBuffer.h"

extern BIT* pcBIT;
extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

StateMachine::StateMachine()
{
    // Initialise the Adjacency List for the Sensor Manager State
    // Machine NUMSTATES long
    // bit 0 = V0, bit 8 = V8

    // (V0)INITIALISE (V1)PREPARED (V2)BITFAIL
    // (V3)POWERDOWN (V4)ABORT (V5)READYRELEASE
    // (V6)READYCONFIRM(V7)SEPARATION(V8)DEPLOY

    abStateAdj[0] = 2; // B000000010 - A(0) = { V1}
    abStateAdj[1] = 36; // B000100100 - A(1) = { V2, V5}
    abStateAdj[2] = 5; // B000000101 - A(2) = { V0, V2}
    abStateAdj[3] = 0; // B000000000 - A(3) = {}
    abStateAdj[4] = 6; // B000000110 - A(4) = { V1, V2}
    abStateAdj[5] = 84; // B001010100 - A(5) = { V2, V4, V6}
    abStateAdj[6] = 180; // B010110100 - A(6) = { V2, V4, V5}
    abStateAdj[7] = 264; // B100001000 - A(7) = { V3, V8}
    abStateAdj[8] = 8; // B000001000 - A(8) = { V3}

    // Set Initial State to
    eCurrentState = INITIALISE;
}

eSTATE StateMachine::RetrieveState(void) const
{
    return eCurrentState;
}

void StateMachine::SetState(eSTATE eNewState)

```

```

{
    uint8_t nAdjacencyListPos;

    // Validate the requested State Change before so doing
    // Shift the current state adjacency value to the right
    // Checking if new state is a valid transition
    nAdjacencyListPos = abStateAdj[eCurrentState] >> eNewState;

    // Mask off the LSB to obtain boolean answer
    // Only change if new state is included in list
    if ((nAdjacencyListPos & 1))
    {
        // Using the result of adjacency review (State change
        // authority)undertake the State Change if so authorised
        eCurrentState = eNewState;

        // Clear the Incorrect State Transition attempted BIT flag
        pcBIT->BITFlagUpdate(INVALID.STATE.CHG, false);

        // Update the Output message with current state
    }
    else
    {
        // Else incorrect requested State Transition
        // Therefore set BITFail error
        // Set the Incorrect State Transition attempted BIT flag
        pcBIT->BITFlagUpdate(INVALID.STATE.CHG, true);
    }

    // Update the I2C Tx Msg Buffer
    // First Invalidate current message area
    g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG0);

    g_uI2CUpdate_MsgTx.g_nType4EnumValue = eCurrentState;

    // Store it into the I2C Message Area
    g_cI2C_MsgTx->putMessage(MSG0, &g_uI2CUpdate_MsgTx);

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG0);
}

```

H.21 Release Controller Listings

H.22 The HostReleaseController.ino Code

Listing H.19: The main Host Release Controller Sketch.

```

/*
 * HostReleaseController.ino
 *
 * Created on: 1 Oct 2014
 * Author: Ian Saxby
 */

#include <Arduino.h>

// #include <Wire.h> // I2C library
#include <NilRTOS.h>
#include <DigitalIO.h>
// Use tiny unbuffered NilRTOS NilSerial library.
#include <NilSerial.h>

// Macro to redefine Serial as NilSerial to save RAM.
// Remove definition to use standard Arduino Serial.
#define Serial NilSerial

#include <Wire.h>
#include <NilFIFO.h>
#include <NilAnalog.h>
#include <NilTimer1.h>
#include "ServoTimer2.h"

#include "PinoutConfigRC.h"
#include "Equates.h"
#include "Power.h"
#include "BIT.h"
#include "I2CBuffer.h"
#include "StateMachine.h"
#include "Commander.h"

// FIFO of received Command Messages
NilFIFO<I2CMsgRx, FIFO_DEPTH>* pcFIFO;

// i2c settings
#define SLAVE_ADDRESS 0x33

// Task Tick Counter, initialised to 0. Range 0 to 9
uint8_t nTaskTick = 0;

// Declare global pointers to the various controller
// objects These are instantiated during setup
Commander* pcCmdr;
BIT* pcBIT;

```

```

// I2C Buffer Arrangements
// Establish a I2C Slave Transmission Message Buffer
// Class pointer
I2CBuffer *g_cI2C_MsgTx;

// Setup a buffer for storing I2C Tx data for transferring to
// I2C Slave Transmission Message Buffer
I2CMsgTx g_uI2CUpdate_MsgTx;

// Setup a global buffer for initial I2C receipt before storing
// in ICP FIFO
I2CMsgRx g_uI2CTemp_MsgRx;

// Setup a global buffer for I2C dispatch for I2C to Transmit
// from
I2CMsgTx g_uI2CTemp_MsgTx;

// declare servoTimer2 object ptrs to control the lock, Release
// and Retract servos
ServoTimer2* pcLockServo;
ServoTimer2* pcReleaseServo;
ServoTimer2* pcUmStowServo;

// Global weight off wheels variable
uint8_t g_eWheelOffWheels;

// Declare and initialize a semaphore for limiting access to a
// region.

SEMAPHORE_DECL(cSem1Hz, 0);
SEMAPHORE_DECL(cSem5Hz, 0);
SEMAPHORE_DECL(cSem10Hz, 0);

//-----
// Declare a stack with 64 bytes beyond context switch and
// interrupt needs.
NIL_WORKING_AREA(waThread1, 96);

// Declare thread function for thread 1.
NIL_THREAD(Thread1, arg)
{

    nilTimer1Start(TSC_PERIOD);

    while (true)
    {
        // Execute 10Hz Commander related functionality
        pcCmdr->Task10Hz();

        // Increment the Task Tick counter
        nTaskTick++;
    }
}

```

```

    // Now determine if slower Tasks require to be run in
    // this slot
    switch (nTaskTick)
    {
        case 4: // every 10th tick (at time slot 4) = 1 Hz
        {
            // Signal 1Hz Task to run
            // Release the Semaphore but not Reschedule RTOS
            // until end

            nilSemSignalI(&cSem1Hz);
            break;
        }
        case 2:// every 5th tick (at time slots 2 and 7) = 5 Hz
        case 7:
        {
            // Signal 5Hz Task to run
            // Release the Semaphore but not Reschedule RTOS
            // until end
            nilSemSignalI(&cSem5Hz);
            break;
        }
        case 10: // on the 10th tick reset TaskTick count to 0
        {
            nTaskTick = 0;
            break;
        }
    }

    // Sleep so lower priority threads can execute.
    nilTimer1Wait();
}

}

//-----
// Declare a stack with 64 bytes beyond context switch and interrupt
// needs.
NIL_WORKING_AREA(waThread3, 96);

// Declare thread function for thread 2.
NIL_THREAD(Thread3, arg)
{

    // 5 Hz Task

    while (TRUE)
    {
        // Wait for notification to run task
        nilSemWaitTimeout(&cSem5Hz, TIME_INFINITE);

        // Execute 5Hz Commander related functionality
        pcCmdr->Task5Hz();
    }
}

```



```

    }
}
//-----
// Declare a stack with 64 bytes beyond context switch and
// interrupt needs.
NIL_WORKING_AREA(waThread4, 96);

// Declare thread function for thread 4.
NIL_THREAD(Thread4, arg)
{

    // *****
    // 1 Hz Task
    // *****

    while (TRUE)
    {
        // Wait for notification to run task
        nilSemWaitTimeout(&cSem1Hz, TIME_INFINITE);

        // Execute 1Hz Commander related functionality
        pcCmdr->Task1Hz();
    }
} //-----
/*
 * Threads static table, one entry per thread. A thread's priority
 * is determined by its position in the table with highest priority
 * first.
 *
 * These threads start with a null argument. A thread's name is also
 * null to save RAM since the name is currently not used.
 */
NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY(NULL, Thread1, NULL, waThread1, sizeof(waThread1))
NIL_THREADS_TABLE_ENTRY(NULL, Thread3, NULL, waThread3, sizeof(waThread3))
NIL_THREADS_TABLE_ENTRY(NULL, Thread4, NULL, waThread4, sizeof(waThread4))

NIL_THREADS_TABLE_END()
//-----

void PinOutInitialisation(void);

void setup()
{

    // Instantiate the I2C Slave Transmission Message Buffers
    g_cI2C_MsgTx = new I2CBuffer();

    pcLockServo = new ServoTimer2;
    pcReleaseServo = new ServoTimer2;
    pcUmStowServo = new ServoTimer2;

```

```

// Initialise uC MiniPro Input Output Pin configuration
PinOutInitialisation();

pcFIFO = new NilFIFO<I2CMsgRx, FIFO_DEPTH>;

// Setup Serial Communication
Serial.begin(9600);

Serial.println("");
Serial.println("_RESTART_");

// initialize i2c as slave
Wire.begin(SLAVE_ADDRESS);

// define callbacks for i2c communication
Wire.onReceive(receiveData);
Wire.onRequest(sendData);

// Instantiate the Commander
pcCmdr = new Commander;

// start kernel
nilSysBegin();
}

//-----
// Loop is the idle thread. The idle thread must not invoke any
// kernel primitive able to change its state to not runnable.
void loop()
{
}

//I2C callbacks

// callback for received command
void receiveData(int byteCount)
{
    int indx = 0;
    uint8_t nOverflow;
    I2CMsgRx* psFIFOSlot;

    while(Wire.available())
    {
        // Ensure no Msg Received is longer than largest expected
        if(indx > MAXMSG_RXLENGTH)
        {
            // Create failure condition here
            // Set the Receive Buffer Overflow Attempted BIT flag
            pcBIT->BITFlagUpdate(RXBUFF_OVRFLW, true);
            // Break out of Receive process
            break;
        }
    }
}

```

```

    else
    {
        // initial fast receipt of I2C Command Data
        g_uI2CTemp_MsgRx.g_nRxAllData[indx] = Wire.read();
        indx++;
    }
}

if ((indx > 0) && (indx <= MAX_MSG_RX_LENGTH))
{
    if (g_uI2CTemp_MsgRx.g_nType1RxMsgNo == SETUP_I2C_SEND_CMD)
    {
        // Retrieve the requested message in readiness for subsequent
        // SendData() activity
        g_cI2C_MsgTx->getTxMessage(\
            (uint8_t)g_uI2CTemp_MsgRx.g_nType1RxEnumValue,
            &g_uI2CTemp_MsgTx);

    }
    else
    {
        // Store Received I2C Command into FIFO

        // Get a free FIFO slot.
        psFIFOSlot = pcFIFO->waitFree(TIME_IMMEDIATE);

        // Only store if free space. else set Bit Fail condition
        if (psFIFOSlot != 0)
        {
            // Store message into FIFO.
            (*psFIFOSlot) = g_uI2CTemp_MsgRx;

            // Signal thread data is available.
            pcFIFO->signalData();
        }
        else
        {
            // Set the Receive Buffer Overflow Attempted BIT flag
            pcBIT->BITFlagUpdate(FIFO_OVRFLW, true);
        }
    }
}

// callback for sending data via I2C
void sendData()
{
    Wire.write(g_uI2CTemp_MsgTx.g_nTxAllData, /\
        _uI2CTemp_MsgTx.nWdCount);
}

```

```
void PinOutInitialisation()
{
    // External Power Enable Output Pin 4, not controlled
    // other than set high
    pinMode(PIN_POWERENB_EXT, OUTPUT);
    // Internal Power Enable Output Pin 5
    pinMode(PIN_POWERENB_INT, OUTPUT);

    // Attach the PWM servo class to the Lock / Unlock Actuator
    pcLockServo->attach(PIN_LOCK_UNLOCK_PWM);
    // Attach the PWM servo class to the Release Actuator
    pcReleaseServo->attach(PIN_RELACTUATOR_PWM);
    // Attach the PWM servo class to the Umbilical Retraction
    // Actuator
    pcUmStowServo->attach(PIN_CONTRACT_PWM);

    // Umbilical stowage will be sorted by internal functionality

    // could be this alternate if range is an issue
    //pcRotorServo->attach(ROTORLOCK_PWM, int min, int max);

    // Initialisation for Interlock and Discrete Input / Outputs
    pinMode(PIN_PKG_INTERLOCK_OUT, OUTPUT);

    // Host connected interlock Pin Input, requires pull-up resistor
    // setting
    // Host disconnected is Active High and no cycle
    pinMode(PIN_PKG_INTERLOCK_IN, INPUT_PULLUP);

    // Active LOW
    pinMode(PIN_COMMIT_RELEASE, INPUT_PULLUP);

    // Debug Pin Input, requires pull-up resistor setting
    // Debug mode is Active Low
    pinMode(PIN_PKG_DEBUG, INPUT_PULLUP);

    // Load Push Button Input, needs debounce logic
    // Load is active LOW
    pinMode(PIN_LOAD_PKG, INPUT_PULLUP);

    // Current Sensor Fault Input Pin, circuit has pull-up resistor
    pinMode(PIN_CURRENT_SENSOR_FAULT, INPUT_PULLUP);

    // External Power Status flag, LTC4353 has internal pull-up resistor
    pinMode(PIN_PWRSTATUS_EXT_ONST1, INPUT_PULLUP);
    // Internal Power Status Flag, LTC4353 has internal pull-up resistor
    pinMode(PIN_PWRSTATUS_INT_ONST2, INPUT_PULLUP);
}
```

H.23 The Equates.h Code

Listing H.20: The Equates header file.

```

/*
 * Equates.h
 *
 * Created on: 14 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef EQUATES_H
#define EQUATES_H

// Actuator PWM in usec
#define POS_LOCK      700
#define POS_UNLOCK    1800
#define POS_CLOSE     700
#define POS_OPEN      1800
#define POS_STOW       700
#define POS_EXTEND     1800
#define SERVO_DELAY  1600

enum ePKGSTATE
{
    P_INITIALISE, P_TRANSREADY, P_INTRANSFER, P_TASKED, P_BITFAIL,
    P_ABORT, P_READYCONFIRM, P_READYRELEASE, P_DEPLOY, P_MIDCOURSE,
    P_FLARE,
    P_POWERDOWN
};

enum ePSMSTATE
{
    PSM_INITIALISE, PSM_PREPARED, PSM_BITFAIL, PSM_POWERDOWN,
    PSM_ABORT, PSM_READYCONFIRM, PSM_READYRELEASE, PSM_SEPARATION,
    PSM_DEPLOY
};

//enum eHSTATE
//{
//    H_INITIALISE, H_HRCREADY, H_PKGCONNECTED, H_TRANSREADY,
//    H_INTRANSFER, H_TASKED, H_BITFAIL, H_ERROR, H_ABORT, H_READYCONFIRM,
//    H_READYRELEASE, H_ACTIONRELEASE, H_GONE, H_HUNG, H_NOREQUESTEDSTATE
//};

//enum eHRCSTATE
//{
//    HRC_INITIALISE, HRC_PKGCHECK, HRC_LOAD, HRC_BITFAIL,
//    HRC_PKGPOWEROFF, HRC_PKGPOWERON, HRC_ABORT, HRC_READYRELEASE,
//    HRC_RELEASECONSENT, HRC_RELEASE, HRC_GONE, HRC_HUNG,
//    HRC_NOREQUESTEDSTATE

```

```
//};  
  
#define NUMSTATES 12 // count does not include NOREQUESTEDSTATE  
  
enum eSTATE {INITIALISE, PKGCHECK, LOAD, BITFAIL, PKGPOWEROFF,  
             PKGPOWERON, ABORT, READYRELEASE,  
             RELEASECONSENT, RELEASE, GONE, HUNG, NOREQUESTEDSTATE};  
  
#endif /* EQUATES_H */
```

H.24 The Commander.h Code

Listing H.21: The Commander header file.

```
/*
 * Commander.h
 *
 * Created on: 3 Sep 2014
 * Author: 0050083462
 */

#ifndef COMMANDER_H
#define COMMANDER_H

#include <Arduino.h>
#include <stdint.h>
#include "I2CBuffer.h"
#include "StateMachine.h"
#include "Power.h"
#include "Interface.h"

class Commander: public StateMachine, public Power, public Interface
{
public:
    Commander();
    void Task10Hz(void);
    void Task5Hz(void);
    void Task1Hz(void);
    void ExecuteCmd(I2CMsgRx* sTempRxMsgBuffer);
    virtual ~Commander(){}

private:
    eSTATE ePreviousState;

    I2CMsgRx sTempRxMsgBuffer;
};

#endif /* COMMANDER_H */
```

H.25 The Commander.cpp Code

Listing H.22: The Commander source file.

```

/*
 * Commander.cpp
 *
 * Created on: 3 Sep 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include <NilRTOS.h>
#include <NilFIFO.h>
#include <NilSerial.h>
#include <util/atomic.h>

#include "ServoTimer2.h"
#include "Commander.h"
#include "PinoutConfigRC.h"
#include "StateMachine.h"
#include "BIT.h"
#include "Interface.h"
#include "I2CBuffer.h"

extern BIT* pcBIT;
extern NilFIFO<I2CMsgRx, FIFO_DEPTH>* pcFIFO;

// for DEBUG purposes
extern I2CMsgRx g_uI2CTemp_MsgRx;

// Commander Constructor
Commander::Commander()
{

    // Instantiate the BIT Object
    pcBIT = new BIT;

    // Turn off Package output power supply
    if (!SwitchPowerSupply(PWRSOURCEOFF))
        pcBIT->BITFlagUpdate(SUPPLY_OFF, true);

    // Setup the Package Connected Interlock wrap around,
    // LOW output
    OutputPkgInterlockSignal();

}

void Commander::Task10Hz(void)
{

```



```

// Internal counter for Abort phase
static uint8_t nExtPowerDelay = 0;

static uint8_t nHungCountDown;

switch(RetrieveState()) // In priority order
{
    case RELEASECONSENT:
    {
        // Check if Commit to Release has been removed by Package
        if(IsReleaseProhibited() ||
            IsPackageInState(PREADYCONFIRM))
        {
            // Return to READYRELEASE
            SetState(READYRELEASE);
        }

        if(ePreviousState != RELEASECONSENT)
        {
            ePreviousState = RELEASECONSENT;

            // Unlock the Release Actuator
            ActuatorLock_UnlockCmd(false);
        }
        nHungCountDown = 20;

        // Confirm the store remains attached
        if(!IsPkgInterlockPresent())
        {
            // Package has gone so set GONE condition
            SetState(GONE);
            // Notify error through BIT
            pcBIT->BITFlagUpdate(PACKAGEINTERLOCK_GONE, true);
        }

        // Check for FMU command state change
        // May receive FMU Cmd to Release
        CheckFMUCmdStateChange();

        break;
    }
    case RELEASE:
    {
        if(ePreviousState != RELEASE)
        {
            ePreviousState = RELEASE;

            // Open the hook
            ActuatorHookOpen_CloseCmd(true);
        }
    }
}

```

```

    }
    // Stop BIT Checking
    pcBIT->ModifyBITCheckFlag( false );

    // Count down to check for HUNG condition
    nHungCountDown--;

    // Confirm the store remains attached
    if( IsPkgInterlockPresent() )
    {
        if( nHungCountDown == 0 )
        {
            // Package remains attached so set HUNG condition
            SetState(HUNG);
        }
    }
    else
    {
        // Package has gone so set GONE condition
        SetState(GONE);
    }
    break;
}
case ABORT:
{

    // Lock the Release Actuator
    ActuatorLock_UnlockCmd( true );

    // Re-supply External Power Supply to the Package
    // enable the "INT" Supply having current sensing
    // and turn off "EXT" supply
    if ( !SwitchPowerSupply( PWRSOURCE.INT ) )
    {
        // Wait within the ABORT State for a maximum of
        // period
        // before commanded supply power should become available
        if( nExtPowerDelay > MAXEXTSUPPLYWAIT )
        {
            // Timeout!
            nExtPowerDelay = 0;
            // Failure to transition to INT power so set BIT
            // fail
            SetState(BITFAIL);
        }
        else
        {
            nExtPowerDelay++;
        }
    }
    else
    {

```

```

        // Reset the Supply delay switch over counter
        nExtPowerDelay = 0;

        // A successful switch to supply Host power to Package
        // was achieved
        // so transition to PREPARED State
        SetState(PKGPOWERON);
    }
    break;
}
default:
    break;
}
}

void Commander::Task5Hz(void)
{
    I2CMsgRx* psTempRxMsgBuffer;
    static bool bThispassLoadBtnState;
    static bool bLastpassLoadBtnState = false;
    static uint8_t nDebounceCounter;
    static bool bButtonDebounceUnderway = false;
    static bool bButtonValid = false;

    // Check for FMU I2C msg within FIFO
    // Use TIME_IMMEDIATE to prevent sleeping in this thread.
    psTempRxMsgBuffer = pcFIFO->waitData(TIME_IMMEDIATE);

    // Act on any received Message
    if (psTempRxMsgBuffer)
    {
        // Yes, one is available so fetch Message from the FIFO.
        sTempRxMsgBuffer = *psTempRxMsgBuffer;

        // Signal FIFO slot is free.
        pcFIFO->signalFree();

        // Act on the message
        ExecuteCmd(&sTempRxMsgBuffer);
    }

    // 5Hz State functions
    switch(RetrieveState()) // In priority order
    {
        case GONE:
        {
            if(ePreviousState != GONE)
            {
                ePreviousState = GONE;

                // Stow the Umbilical Connector

```

```

        ActuatorStow_ExtendCmd(true);
    }
    break;
}
case HUNG:
{
    // Confirm the store remains attached
    if(!IsPkgInterlockPresent())
    {
        // Package has gone so set GONE condition
        SetState(GONE);
    }

    if(ePreviousState != HUNG)
    {
        ePreviousState = HUNG;

        // Close the hook
        ActuatorHookOpen_CloseCmd(false);

        // Lock the Release Actuator
        ActuatorLock_UnlockCmd(true);
    }
    break;
}

case READYRELEASE:
{
    // Shutdown Host Power supply output to Package
    // disable the "INT" Supply having current sensing
    SwitchPowerSupply(PWRSOURCE_OFF);

    // Confirm the store remains attached
    if(!IsPkgInterlockPresent())
    {
        // Package has gone so set GONE condition
        SetState(GONE);
        // Notify error through BIT
        pcBIT->BITFlagUpdate(PACKAGEINTERLOCK_GONE, true);

        break; // No reason to continue with this state
        // functionality
    }

    // Check for FMU command state change
    // May receive ABORT CMD from FMU
    CheckFMUCmdStateChange();

    // Check if Commit to Release is set and Package is in
    // READYRELEASE
    if(IsCommitToReleasePresent() &&

```

```

        IsPackageInState(P_READYRELEASE))
    {
        // Release consent has been established
        // and the package is in the correct state
        SetState(RELEASECONSENT);
    }

    break;
}
case INITIALISE:
{

    // Allow the BIT Checks
    pcBIT->ModifyBITCheckFlag(true);

    // Check the DEBUG interlock state
    UpdateDebugInterLockState();

    if(ePreviousState != INITIALISE)
    {
        ePreviousState = INITIALISE;

        // Unlock Hook Actuator
        ActuatorLock_UnlockCmd(false);
        // Close the hook
        ActuatorHookOpen_CloseCmd(false);
        // Lock the Hook Actuator
        ActuatorLock_UnlockCmd(true);
    }

    // transition to PKGCHECK state
    SetState(PKGCHECK);

    break;
}
case PKGCHECK:
{
    // Close the Release Hook
    if(ePreviousState != PKGCHECK)
    {
        ePreviousState = PKGCHECK;

        // Unlock Hook Actuator
        ActuatorLock_UnlockCmd(false);

        // Close the hook
        ActuatorHookOpen_CloseCmd(false);

        // Lock the Hook Actuator
        ActuatorLock_UnlockCmd(true);
    }
}

```

```

// Check Load Push Button only if Weight On Wheels is valid
if (!IsWeightOffWheels())
{
    bThispassLoadBtnState = IsPkgLoadButtonPressed();

    if(bThispassLoadBtnState != bLastpassLoadBtnState)
    {
        // reset the debounce timer
        nDebounceCounter = 0;
        bButtonDebounceUnderway = true;
    }

    if(bButtonDebounceUnderway)
    {
        nDebounceCounter++;

        if(nDebounceCounter > 2)
        {
            // Debounce conditions satisfied, ignore
            // counter until next change

            bButtonDebounceUnderway = false;

            // Load button state is valid and Pressed
            if(bThispassLoadBtnState)
            {
                SetState(LOAD);
            }
        }

        bLastpassLoadBtnState = bThispassLoadBtnState;
    }
}

// Detection of Package Interlock presence
if(IsPkgInterlockPresent())
{
    // Detected Interlock wrap successful therefore
    // progress
    SetState(PKGPOWEROFF);
}

break;
}
case LOAD:
{
    if(ePreviousState != LOAD)
    {
        ePreviousState = LOAD;
        // Shutdown Host Power supply output to Package
        // disable the "INT" Supply having current sensing
    }
}

```

```

        SwitchPowerSupply(PWRSOURCEOFF);

        // Unlock Hook Actuator
        ActuatorLock_UnlockCmd(false);
        // Open the hook
        ActuatorHookOpen_CloseCmd(true);

        // Extend the Umbilical Connector
        ActuatorStow_ExtendCmd(false);
    }

    bThispassLoadBtnState = IsPkgLoadButtonPressed();

    if(bThispassLoadBtnState != bLastpassLoadBtnState)
    {
        // reset the debounce timer
        nDebounceCounter = 0;
        bButtonDebounceUnderway = true;
    }

    if(bButtonDebounceUnderway)
    {
        nDebounceCounter++;

        if(nDebounceCounter > 2)
        {
            // Debounce conditions satisfied, ignore counter
            // until next change

            bButtonDebounceUnderway = false;

            // Load button state is valid
            // If Load button is no longer pressed
            if(!bThispassLoadBtnState)
            {
                // Package Load button has been released
                // return to PKGCHECK state where a check of
                // Package Interlock will occur
                SetState(PKGCHECK);
            }
        }
        bLastpassLoadBtnState = bThispassLoadBtnState;
    }
    break;
}
case PKGPOWEROFF:
{
    // Confirm the store remains attached
    if(!IsPkgInterlockPresent())
    {
        // Package has gone so set GONE condition

```

```

        SetState(GONE);
        // Notify error through BIT
        pcBIT->BITFlagUpdate(PACKAGEINTERLOCK_GONE, true);

        break; // No reason to continue with this state
        // functionality
    }

    if(ePreviousState != PKGPOWEROFF)
    {
        ePreviousState = PKGPOWEROFF;
        // Shutdown Host Power supply output to Package
        // disable the "INT" Supply having current sensing
        SwitchPowerSupply(PWRSOURCEOFF);

        // Lock the Release Actuator
        ActuatorLock_UnlockCmd(true);
    }
    // Check for FMU command state change
    CheckFMUCmdStateChange();

    break;
}
case PKGPOWERON:
{
    // Confirm the store remains attached
    if(!IsPkgInterlockPresent())
    {
        // Package has gone so set GONE condition
        SetState(GONE);
        // Notify error through BIT
        pcBIT->BITFlagUpdate(PACKAGEINTERLOCK_GONE, true);

        break; // No reason to continue with this state
        // functionality
    }

    if(ePreviousState != PKGPOWERON)
    {
        ePreviousState = PKGPOWERON;

        // Supply Host Power output to Package
        // Re-supply External Power Supply to the Package
        // enable the "INT" Supply having current sensing
        if (!SwitchPowerSupply(PWRSOURCEINT))
        {
            // Failure to transition to INT power so set BIT
            // fail
            SetState(BITFAIL);
        }
    }
}

```



```

        // Lock the Release Actuator
        ActuatorLock.UnlockCmd(true);
    }
    // Check for FMU command state change
    CheckFMUCmdStateChange();

    break;
}
default:
    break;
}
}

void Commander::Task1Hz(void)
{
    static uint8_t Count1Hz;
    static bool bDebugFirstPass = true;

    // For DEBUG purposes only
    I2CMsgRx* psFIFOSlot;

    // 1Hz State functions
    switch(RetrieveState()) // In priority order
    {
        case BITFAIL:
        {
            if(ePreviousState != BITFAIL)
            {
                ePreviousState = BITFAIL;

                // Start BIT Checking
                pcBIT->ModifyBITCheckFlag(true);

                // Shutdown Host Power supply output to Package
                // disable the "INT" Supply having current sensing
                SwitchPowerSupply(PWRSOURCEOFF);

                // Lock the Release Actuator
                ActuatorLock.UnlockCmd(true);
            }
            break;
        }
    }
    if(pcBIT->RetrieveBITCheckFlag())
    {
        pcBIT->BITCheck();
    }

    UpdateDebugInterLockState();
}

```

// DEBUG statements from here to cycle through states in timely manner

```

if (RetrieveState() == PKGCHECK)
{
    if (bDebugFirstPass)
    {
        bDebugFirstPass = false;
        Count1Hz = 0;

        // Setup for receipt of MSG3 and identify WQWW
        g_uI2CTemp_MsgRx.g_nType1RxMsgNo = MSG3;
        g_uI2CTemp_MsgRx.g_nType2RxEnumValue = WQWW;
        // Store Received I2C Command into FIFO

        // Get a free FIFO slot.
        psFIFOSlot = pcFIFO->waitFree(TIME_IMMEDIATE);

        // Only store if free space. else set Bit Fail condition
        if (psFIFOSlot != 0)
        {
            // Store message into FIFO.
            (*psFIFOSlot) = g_uI2CTemp_MsgRx;

            // Signal thread data is available.
            pcFIFO->signalData();
        }
    }
}
if (RetrieveState() >= PKGCHECK)
{
    Count1Hz++;
    if (Count1Hz == 8)
    {
        if (RetrieveState() == PKGPOWEROFF)
        {
            // Setup for receipt of MSG2 and requested state
            // change to PKGPOWERON
            g_uI2CTemp_MsgRx.g_nType1RxMsgNo = MSG2;
            g_uI2CTemp_MsgRx.g_nType1RxEnumValue =
                (uint8_t)PKGPOWERON;
            // Store Received I2C Command into FIFO

            // Get a free FIFO slot.
            psFIFOSlot = pcFIFO->waitFree(TIME_IMMEDIATE);

            // Only store if free space. else set Bit Fail
            // condition
            if (psFIFOSlot != 0)
            {
                // Store message into FIFO.
                (*psFIFOSlot) = g_uI2CTemp_MsgRx;
            }
        }
    }
}

```

```

        // Signal thread data is available.
        pcFIFO->signalData();
    }
    bDebugFirstPass = true;
}
else
    Count1Hz = 3;
}
if (Count1Hz == 14)
{
    if(RetrieveState() == PKGPOWERON)
    {
        // Setup for receipt of MSG2 and requested state
        // change to READYRELEASE
        g_uI2CTemp_MsgRx.g_nType1RxMsgNo = MSG2;
        g_uI2CTemp_MsgRx.g_nType1RxEnumValue =
            (uint8_t)READYRELEASE;
        // Store Received I2C Command into FIFO

        // Get a free FIFO slot.
        psFIFOSlot = pcFIFO->waitFree(TIME_IMMEDIATE);

        // Only store if free space. else set Bit Fail
        // condition
        if (psFIFOSlot != 0)
        {
            // Store message into FIFO.
            (*psFIFOSlot) = g_uI2CTemp_MsgRx;

            // Signal thread data is available.
            pcFIFO->signalData();
        }

        bDebugFirstPass = true;
    }
    else
    {
        Count1Hz = 10;
    }
}
if((Count1Hz > 14) && (!IsCommitToReleasePresent()))
{
    Serial.println("Waiting for Commit to Rel");
}
if (Count1Hz == 18)
{
    if(RetrieveState() == READYRELEASE)
    {

        // Setup for receipt of MSG5 to indicate Package
        // has transitioned to P_READYRELEASE
    }
}

```

```

        g_uI2CTemp_MsgRx.g_nType1RxMsgNo = MSG5;
        g_uI2CTemp_MsgRx.g_nType1RxEnumValue =
            (uint8_t)P_READYRELEASE;
        // Store Received I2C Command into FIFO
        // Get a free FIFO slot.
        psFIFOSlot = pcFIFO->waitFree(TIME_IMMEDIATE);

        // Only store if free space. else set Bit Fail
        // condition
        if (psFIFOSlot != 0)
        {
            // Store message into FIFO.
            (*psFIFOSlot) = g_uI2CTemp_MsgRx;

            // Signal thread data is available.
            pcFIFO->signalData();
        }

        bDebugFirstPass = true;
    }
    else
        Count1Hz = 15;
}
if (Count1Hz == 22)
{
    if (RetrieveState() == RELEASECONSENT)
    {
        // Setup for receipt of MSG2 and requested state
        // change to RELEASE
        g_uI2CTemp_MsgRx.g_nType1RxMsgNo = MSG2;
        g_uI2CTemp_MsgRx.g_nType1RxEnumValue =
            (uint8_t)RELEASE;
        // Store Received I2C Command into FIFO
        // Get a free FIFO slot.
        psFIFOSlot = pcFIFO->waitFree(TIME_IMMEDIATE);

        // Only store if free space. else set Bit Fail
        // condition
        if (psFIFOSlot != 0)
        {
            // Store message into FIFO.
            (*psFIFOSlot) = g_uI2CTemp_MsgRx;

            // Signal thread data is available.
            pcFIFO->signalData();
        }
        bDebugFirstPass = true;
    }
    else
    {

```

```

        Count1Hz = 19;
    }
}
if (Count1Hz > 30)
{
    bDebugFirstPass = true;
    Count1Hz = 30;
}
}
else
    Count1Hz = 0;
}

void Commander::ExecuteCmd(I2CMsgRx* psRcvMsgBuffer)
{
    // Decode and react to Message No 2 through 6
    switch(psRcvMsgBuffer->g_nType1RxMsgNo)
    {
        case MSG2:
        {
            // Commanded State Change
            // Single data byte
            StoreRequestedStateChange((eSTATE)psRcvMsgBuffer->\
                g_nType1RxEnumValue);

            break;
        }
        case MSG3:
        {
            // Save notified Weight Off Wheels Status
            UpdateWeightOffWheelsState(psRcvMsgBuffer->\
                g_nType2RxEnumValue);

            break;
        }
        case MSG4:
        {
            // Initialise the Internal Battery Capacity
            SetupCapacityAhValue(psRcvMsgBuffer->\
                g_nType4RxIntValue);

            break;
        }
        case MSG5:
        {
            // Save the notified Package State
            StorePackageState((ePKGSTATE)psRcvMsgBuffer->\
                g_nType1RxEnumValue);

            break;
        }
        case MSG6:
        {
            // Debug Interface Messages
            // todo
            break;
        }
    }
}

```

```
        }  
    }  
}
```

H.26 The BIT.h Code

Listing H.23: The BIT header file.

```

/*
 * BIT.h
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef BIT_H_
#define BIT_H_

enum eBITFLAG {
    SUPPLY_INT,           // BIT Pos 0
    SUPPLY_EXT,           // BIT Pos 1
    SUPPLY_OFF,           // BIT Pos 2
    CURRENT_FAULT,        // BIT Pos 3
    PACKAGEINTERLOCK_GONE, // BIT Pos 4
    BATCAP_BELOW_WARN,    // BIT Pos 5
    FMU_NOT_COMM,         // BIT Pos 6
    INVALID_STATE_CHG,    // BIT Pos 7
    SUPPLY_NOT_CMD,        // BIT Pos 8
    RXBUFF_OVRFLW,        // BIT Pos 9
    FIFO_OVRFLW,          // BIT Pos 10
    HOOK_LOCKED,          // BIT Pos 11
    SPARE1,               // BIT Pos 12
    SPARE2,               // BIT Pos 13
    SPARE3,               // BIT Pos 14
    BIT_FLAG_OVRFLW       // BIT Pos 15
};

class BIT
{
public:
    BIT();
    void BITFlagUpdate(eBITFLAG nBITFlagPosition, bool bCondition);
    uint16_t ReadBITFlags() const;
    bool RetrieveBITCheckFlag(void) const;
    void ModifyBITCheckFlag(bool bFlag);
    bool BITCheck(void);

private:
    uint16_t nBITCondition;
    // Authority to undertake BIT checks
    bool bBITCheckAuthFlag;
};

#endif /* BIT_H_ */

```

H.27 The BIT.cpp Code

Listing H.24: The BIT source file.

```

/*
 * BIT.cpp
 *
 * Created on: 7 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include "BIT.h"
#include "Commander.h"
#include "PinoutConfigRC.h"

extern Commander* pcCmdr;
extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

// BIT Constructor
BIT::BIT()
{
    nBITCondition = 0;

    // Initialise Allow BIT Checking
    bBITCheckAuthFlag = true;
}

void BIT::BITFlagUpdate(eBITFLAG nBITFlagPosition, bool bCondition)
{
    // the flag to be shifted into BIT Flag position
    uint16_t nFlag = 1;

    // Assert that nBITFlagPosition < 16 positions
    if (nBITFlagPosition > 15)
    {
        BITFlagUpdate(BIT_FLAG_OVRFLW, true);
        //pcCmdr->SetState(BITFAIL);
        return;
    }

    // Shift and Set or clear the BIT Flag position
    if(bCondition)
    {
        // The update is to record a BIT FAIL
        // Need to shift and OR Mask into place
        nBITCondition |= (nFlag << nBITFlagPosition);
        // Set State to BITFAIL
        pcCmdr->SetState(BITFAIL);
    }
}

```



```

    }
    else
    {
        // The update is to record a BIT PASS
        // need to shift, Invert and AND Mask into place
        nBITCondition &= (~(nFlag << nBITFlagPosition));

        // Check if all BIT Flag positions are false
        if(!(nBITCondition & 0xFFFF))
        {
            // Now check if the system is already in BITFAIL
            // State If so then this allows the system to
            // transition out of BITFAIL
            if(pcCmdr->RetrieveState() == BITFAIL)
            {
                pcCmdr->SetState(INITIALISE);
            }
        }
    }

    // Update the I2C Tx Msg Buffer
    // First Invalidate current message area
    g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG4);

    g_uI2CUpdate_MsgTx.g_nType3UIntValue = nBITCondition;

    // Store it into the I2C Message Area
    g_cI2C_MsgTx->putMessage(MSG4, &g_uI2CUpdate_MsgTx);

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG4);
}

uint16_t BIT::ReadBITFlags() const
{
    return nBITCondition;
}

bool BIT::RetrieveBITCheckFlag(void) const
{
    return bBITCheckAuthFlag;
}

void BIT::ModifyBITCheckFlag(bool bFlag)
{
    bBITCheckAuthFlag = bFlag;
}

bool BIT::BITCheck(void)
{
    // Execute BIT

```

```
// Check that the Current Sensor has not set the Over Current
// FAULT flag
// Current Sensor FAULT condition is Active LOW
if (pcCmdr->ReadSensorFAULT())
{
    // Fault exists so set the Current Fault BIT flag
    BITFlagUpdate(CURRENT_FAULT, true);
}
else
{
    // No problem here so clear the Power BIT flag
    BITFlagUpdate(CURRENT_FAULT, false);
}
}
```

H.28 The I2CBuffer.h Code

Listing H.25: The I2CBuffer header file.

```

/*
 * I2CBuffer.h
 *
 * Created on: 31 Aug 2014
 * Author: Ian Saxby
 */

#ifndef I2CBUFFER_H_
#define I2CBUFFER_H_

#include <Arduino.h>
#include <stdint.h>
#include "StateMachine.h"
#include "Interface.h"

#define MAX_MSG_TX_LENGTH 5 // not including msg no. or wordcount
// at last element
#define MAX_MSG_RX_LENGTH 2
#define MAX_TX_BUFFERS 6
#define FIFO_DEPTH 2

#define SETUP_I2C_SEND_CMD 1

enum eMSG {MSG0, MSG1, MSG2, MSG3, MSG4, MSG5, MSG6};

typedef struct I2CMsgTx
{
    struct {
        bool bInvalid;
        uint8_t g_nMsgNo;
    };
    union
    {
        struct {
            uint8_t g_nType1EnumValue;
            uint8_t g_nType1Rem[MAX_MSG_TX_LENGTH - 1];
        };
        struct {
            int16_t g_nType2IntValue;
            uint8_t g_nType2Rem[MAX_MSG_TX_LENGTH - 2];
        };
        struct {
            uint16_t g_nType3UIntValue;
            uint8_t g_nType3Rem[MAX_MSG_TX_LENGTH - 2];
        };
        struct {
            eSTATE g_nType4EnumValue;

```

```

        uint8_t g_nType4Rem [MAX_MSG_TX_LENGTH - 1];
    };
    struct {
        bool      g_nType5BoolValue1;
        bool      g_nType5BoolValue2;
        bool      g_nType5BoolValue3;
        uint8_t   g_nType5EnumValue [MAX_MSG_TX_LENGTH - 3];
    };
    uint8_t g_nTxAllData [MAX_MSG_TX_LENGTH];
};
uint8_t nWdCount;
} I2CMsgTx;

```

```

typedef struct I2CMsgRx
{
    union {
        struct {
            uint8_t g_nType1RxMsgNo;
            uint8_t g_nType1RxEnumValue;
            uint8_t g_nType1RxRem;
        };
        struct {
            uint8_t g_nType2RxMsgNo;
            eWOFFW g_nType2RxEnumValue;
            uint8_t g_nType2RxRem;
        };
        struct {
            uint8_t g_nType3RxMsgNo;
            uint8_t g_nType3RxEnumValue;
            uint8_t g_nType3RxRem;
        };
        struct {
            uint8_t g_nType4RxMsgNo;
            uint16_t g_nType4RxIntValue;
        };
        uint8_t g_nRxAllData [MAX_MSG_RX_LENGTH];
    };
} I2CMsgRx;

```

```

class I2CBuffer
{
public:
    // Constructor
    I2CBuffer ();

    void I2CInvalidateTxMsg (eMSG eMsg);
    void I2CValidateTxMsg (eMSG eMsg);
    // Store Message into identified Message Buffer
    void putMessage (eMSG eMsg, I2CMsgTx *pMsgData);

    // Retrieve Message from identified Message Buffer

```

```
    void getTxMessage( uint8_t MsgNoTx, I2CMsgTx *pMsgOut );

private:

    I2CMsgTx sMsgTxArray [MAX_TX_BUFFERS];

};
#endif /* I2CBUFFER_H_ */
```

H.29 The I2CBuffer.cpp Code

Listing H.26: The I2CBuffer source file.

```

/*
 * I2CBuffer.cpp
 *
 * Created on: 31 Aug 2014
 * Author: Ian Saxby
 */

#include <stdio.h>
#include "I2CBuffer.h"

I2CBuffer::I2CBuffer()
{
    // Initialise the Output (TX) Message buffer area
    for (int MsgNo = 0; MsgNo < MAX_TX_BUFFERS; MsgNo++)
    {
        for (int indy = 0; indy < MAX_MSG_TX_LENGTH; indy++)
        {
            sMsgTxArray[MsgNo].g_nTxAllData[indy] = 0;
        }
        // Invalidate each message
        sMsgTxArray[MsgNo].bInvalid = true;

        // Initialise the Word count for each message
        switch (MsgNo)
        {
            case MSG0: // Sensor Manager State
            {
                sMsgTxArray[MSG0].nWdCount = 2;
                break;
            }
            case MSG1: // Reserved in HRC (not used)
            {
                sMsgTxArray[MSG1].nWdCount = 0;
                break;
            }
            case MSG2: // Host Battery Capacity Supplied
            {
                sMsgTxArray[MSG2].nWdCount = 3;
                break;
            }
            case MSG3: // Supply Source
            {
                sMsgTxArray[MSG3].nWdCount = 2;
                break;
            }
            case MSG4: // Built In Test Results
            {
                sMsgTxArray[MSG4].nWdCount = 3;
            }
        }
    }
}

```

```

        break;
    }
    case MSG5: // Interlocks and possibly actuator positions
    {
        sMsgTxArray[MSG5].nWdCount = 4;
        break;
    }
}
}

void I2CBuffer::I2CInvalidateTxMsg(eMSG eMsg)
{
    sMsgTxArray[eMsg].bInvalid = true;
}

void I2CBuffer::I2CValidateTxMsg(eMSG eMsg)
{
    sMsgTxArray[eMsg].bInvalid = false;
}

// Store Message into identified Message Buffer
void I2CBuffer::putMessage(eMSG eMsg, I2CMsgTx *pMsgIn)
{
    for(uint8_t indx = 1; indx < MAXMSG.TXLENGTH; indx++)
    {
        sMsgTxArray[eMsg].g_nTxAllData[indx] =
            (*pMsgIn).g_nTxAllData[indx];
    }
}

// Retrieve Message from identified Message Buffer into local buffer
// for I2C
void I2CBuffer::getTxMessage(uint8_t MsgNoTx, I2CMsgTx *pMsgOut)
{
    (*pMsgOut) = sMsgTxArray[MsgNoTx];
}

```

H.30 The Interface.h Code

Listing H.27: The Interface header file.

```

/*
 * Interface.h
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef INTERFACE_H_
#define INTERFACE_H_

// Separation counter for 1 second at 10 * Task10Hz
#define SEPARATIONCOUNT 10
// Define Abort delay in 10Hz iterations
#define MAXEXTSUPPLYWAIT 20 // = 2 seconds

enum eWOFFW {WONW, WOFFW};

class Interface
{
public:

    Interface();

    bool IsDebugMode(void) const;
    // Detect the Debug Interlock state (Active LOW) and update
    // Mode flag
    bool UpdateDebugInterLockState(void);

    // Output the Host Interlock wrap signal at state (Active LOW)
    // and update
    // flag
    void OutputPkgInterlockSignal(void);
    // Check if Pkg Interlock is present (still LOW)
    bool IsPkgInterlockPresent(void);

    // Uses PkgInterlock Signal as source for active LOW signal
    bool IsPkgLoadButtonPressed(void);
    bool LoadButtonDebounce(bool bThisPass);

    bool IsCommitToReleasePresent(void);
    bool IsReleaseProhibited() const;

    void UpdateWeightOffWheelsState(eWOFFW bWoffW);
    bool IsWeightOffWheels(void) const;

    void ActuatorLock_UnlockCmd(bool bCmd);

```



```
void ActuatorHookOpen_CloseCmd( bool bCmd);
void ActuatorStow_ExtendCmd( bool bCmd);

virtual ~Interface() {}

private:

void UpdateI2CMsg( void );

eWOFFW eWOffWStatus;

bool bDebugMode;

bool bCommitToRelease;

bool bPkgInterLockPresent;

bool bPkgLoadBtnPressed;

bool bActuatorLockStatus;
bool bActuatorHookStatus;
bool bActuatorUmbilicalStatus;

bool bThispassLoadBtnState;
bool bLastpassLoadBtnState = false;
uint8_t nDebounceCounter;
bool bButtonDebounceUnderway = false;
bool bButtonValid = false;

};

#endif /* INTERFACE_H_ */
```

H.31 The Interface.cpp Code

Listing H.28: The Interface source file.

```

/*
 * Interface.cpp
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#include <Arduino.h>
#include <NilRTOS.h>
#include "ServoTimer2.h"
#include "Interface.h"
#include "Equates.h"
#include "PinoutConfigRC.h"
#include "I2CBuffer.h"
#include "Bit.h"

extern ServoTimer2* pcLockServo;
extern ServoTimer2* pcReleaseServo;
extern ServoTimer2* pcUmStowServo;

extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;
extern BIT* pcBIT;

Interface::Interface()
{
    eWOffWStatus = WQNW;

    UpdateDebugInterLockState();

    bCommitToRelease = false;

    bPkgLoadBtnPressed = false;

    IsPkgInterlockPresent();
}

bool Interface::IsDebugMode(void) const
{
    return bDebugMode;
}

bool Interface::UpdateDebugInterLockState(void)
{
    // Return the Debug Input Interlock state
    // This pin is active LOW using pull-up resistors
    if(digitalRead(PIN_PKG_DEBUG))

```

```

    {
        // Input is HIGH so Debug mode IS NOT enabled
        bDebugMode = false;
    }
    else
    {
        // Input is LOW so Debug mode IS enabled
        bDebugMode = true;
    }

    UpdateI2CMsg();
    return bDebugMode;
}

void Interface::OutputPkgInterlockSignal(void)
{
    // This pin is active LOW as is using pull-up resistors
    digitalWrite(PIN_PKG_INTERLOCK_OUT, LOW);
}

bool Interface::IsPkgInterlockPresent(void)
{
    // This pin is active LOW as there are pull-up resistors
    if(digitalRead(PIN_PKG_INTERLOCK_IN))
    {
        // Input is HIGH so Pkg Interlock IS NOT present
        bPkgInterLockPresent = false;
    }
    else
    {
        // Input is LOW so Pkg Interlock IS present
        bPkgInterLockPresent = true;
    }

    UpdateI2CMsg();

    return bPkgInterLockPresent;
}

bool Interface::IsPkgLoadButtonPressed(void)
{
    // This pin provides Active LOW input signal for Load button
    // as well
    digitalWrite(PIN_PKG_INTERLOCK_OUT, LOW);

    // This pin is active LOW as is using pull-up resistors
    if(digitalRead(PIN_LOAD_PKG))
    {
        // Load buttons are not depressed
        bPkgLoadBtnPressed = false;
    }
    else

```

```

    {
        // Input is LOW so Load buttons are depressed
        bPkgLoadBtnPressed = true;
    }

    return bPkgLoadBtnPressed;
}

bool Interface::IsCommitToReleasePresent(void)
{
    // This pin is active LOW as is using pull-up resistors
    if (digitalRead(PIN_COMMIT_RELEASE))
    {
        // Input is HIGH so Pkg is not authorising release
        bCommitToRelease = false;
    }
    else
    {
        // Input is LOW so Pkg is notifying Commit to Release
        bCommitToRelease = true;
    }
    // Update the Interface output I2C msg
    UpdateI2CMsg();

    return bCommitToRelease;
}

bool Interface::IsReleaseProhibited(void) const
{
    return !bCommitToRelease;
}

void Interface::UpdateWeightOffWheelsState(eWOFFW bWoffW)
{
    eWOffWStatus = bWoffW;
}

bool Interface::IsWeightOffWheels(void) const
{
    if (eWOffWStatus == WOFFW)
        return true;
    else
        return false;
}

void Interface::ActuatorLock_UnlockCmd(bool bCmd)
{
    // bCmd == true commands Lock Condition
    if (bCmd)
    {
        pcLockServo->write(POS_LOCK);
    }
}

```

```

    }
    else // bCmd == false; Command Unlock Condition
    {
        pcLockServo->write(POS_UNLOCK);
    }
    systime_t waketime = nilTimeNow();
    waketime += MS2ST(2000);
    nilThdSleepUntil(waketime);

    // Trusting that the Actuator faithfully acts as commanded
    bActuatorLockStatus = bCmd;
}

void Interface::ActuatorHookOpen_CloseCmd(bool bCmd)
{
    systime_t waketime;

    // Only progress if logically UNLOCKED as well
    if(bActuatorLockStatus == false)
    {
        // bCmd == true commands Open Condition
        if(bCmd)
        {
            pcReleaseServo->write(POS_OPEN);
        }
        else // bCmd == false; Command Close Condition
        {
            pcReleaseServo->write(POS_CLOSE);
        }
        waketime = nilTimeNow();
        waketime += MS2ST(2000);
        nilThdSleepUntil(waketime);

        // Trusting that the Actuator faithfully acts as commanded
        bActuatorHookStatus = bCmd;
    }
    else
    {
        // Flag BIT error
        pcBIT->BITFlagUpdate(HOOK_LOCKED, true);
    }
}

void Interface::ActuatorStow_ExtendCmd(bool bCmd)
{
    systime_t waketime;

    // bCmd == true commands Stow Condition
    if(bCmd)
    {
        pcUmStowServo->write(POS_STOW);
    }
    else // bCmd == false; Command Extend Condition

```

```
{
    pcUmStowServo->write(POS.EXTEND);
}
waketime = nilTimeNow();
waketime += MS2ST(SERVO_DELAY);
nilThdSleepUntil(waketime);

// Trusting that the Actuator faithfully acts as commanded
bActuatorUmbilicalStatus = bCmd;
}

void Interface::UpdateI2CMsg(void)
{
    // Update the I2C Tx Msg Buffer
    // First Invalidate current message area
    g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG5);

    g_uI2CUpdate_MsgTx.g_nType5BoolValue1 = bPkgInterLockPresent;
    // Transmit the Debug state
    g_uI2CUpdate_MsgTx.g_nType5BoolValue2 = bDebugMode;

    // Transmit the Commit To Release Status
    g_uI2CUpdate_MsgTx.g_nType5BoolValue3 = bCommitToRelease;

    // Store it into the I2C Message Area
    g_cI2C_MsgTx->putMessage(MSG5, &g_uI2CUpdate_MsgTx);

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG5);
}
```

H.32 The PinoutConfigRC.h Code

Listing H.29: The PinoutConfigRC header file.

```

/*
 * PinoutConfigRC.h
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef PINOUTCONFIGRC_H
#define PINOUTCONFIGRC_H

// Definition of Release Controller Pro Mini Pinout Assignments
// Spare
// #define PIN_CHANNEL_A 2
// Spare
// #define PIN_CHANNEL_B 3

// Package Release Lock / Unlock
#define PIN_LCK_UNLK_PWM 6
// Package Release Actuator
#define PIN_REL_ACTUATOR_PWM 9
// Package Release Actuator
#define PIN_CON_RETRACT_PWM 10

// External Power Enable, always set high in HRC to disable EXT power
// supply
#define PIN_POWERENB_EXT 4
// Host Power Supply Enable
#define PIN_POWERENB_INT 5

// Host Interconnect wrap Output
#define PIN_PKG_INTERLOCK_OUT 7
// Commit to Release
#define PIN_COMMIT_RELEASE 8
// Host Interconnect wrap Input
#define PIN_PKG_INTERLOCK_IN 11
// Debug Input
#define PIN_PKG_DEBUG 14
// Load Package Input Switch
#define PIN_LOAD_PKG 12

// Current Sensor Fault Input
#define PIN_CURRENT_SENSOR_FAULT 15
// External Power State Status ONST1 Input
#define PIN_PWR_STATUS_EXT_ONST1 16
// Internal Power State Status ONST2 Input
#define PIN_PWR_STATUS_INT_ONST2 17

```

```
// Current Sensor Value input analog
#define PIN_VIOUTSENSE A7
// Voltage Supply Sense input analog
#define PIN_VOLTAGESENSE A6
// I2C SDA
#define PIN_I2C_SDA A4
// I2C SCL
#define PIN_I2C_SCL A5

#endif  /* PINOUTCONFIGRC_H */
```


H.33 The Power.h Code

Listing H.30: The Power header file.

```

/*
 * Power.h
 *
 * Created on: 7 Nov 2014
 * Author: Ian Saxby
 *          0050083462
 */

#ifndef POWER_H
#define POWER_H

#include <Arduino.h>
#include <stdint.h>
#include "QuadEncoder.h"

// Power Controller is Active Low
#define POWERON LOW
#define POWEROFF HIGH

// Constants defining Power Cmds for External and Internal
// supply selection
enum ePWRSUPPLY {PWRSOURCEEXT, PWRSOURCEINT, PWRSOURCEOFF};

class Power
{
public:
    Power();
    void SetupCapacityAhValue(uint16_t nValue);

    void UpdateInstantCurrent(void);
    void UpdateInstantVoltage(void);
    uint16_t ReadCapacityRemain(void) const;
    uint16_t ReadInstantVoltage(void) const;

    bool SwitchPowerSupply(ePWRSUPPLY eRequestedSupply);
    bool ReadSupplyOutputState(ePWRSUPPLY eSupply) const;
    ePWRSUPPLY ReadCommandedSupply(void) const;
    ePWRSUPPLY ReadRecordedSupply(void) const;

    bool ReadSensorFAULT(void) const;

    virtual ~Power(){}

private:

    // Initial Current Rating
    uint16_t nSetupCapacityAhRating;

```

```
// Power Statistics
uint16_t nRemainingAhCapacity;
uint16_t nInstantCurrent;
uint16_t nInstantVoltage;
uint32_t tPrevTimeSense;

// Next two variables relate to Power Source
// enumerated variable contents
ePWRSUPPLY ePowerCommand;
// ePowerSource reflects actual Power Source
ePWRSUPPLY ePowerSource;

// ACS711 Current Sensor Over Current Fault input
// bool bSensorFAULT;

};

#endif /* POWER_H */
```

H.34 The Power.cpp Code

Listing H.31: The Power source file.

```

/*
 * Power.cpp
 *
 * Created on: 7 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include <NilAnalog.h>
#include "Power.h"
#include "PinoutConfigRC.h"
#include "BIT.h"
#include "I2CBuffer.h"

extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

// Power Constructor
Power::Power()
{
    nSetupCapacityAhRating = 0;

    // Power Statistics
    nRemainingAhCapacity = 0;
    nInstantVoltage = 0;
}

void Power::SetupCapacityAhValue(uint16_t nValue)
{
    nSetupCapacityAhRating = nValue;
}

void Power::UpdateInstantCurrent(void)
{
    // This is coded to match the calling timing of 1Hz
    // This functionality needs considerable verification before
    // utilisation

    uint32_t tCurrentSenseNow;
    uint16_t nVIOUT;

    // Determine time of Sense
    tCurrentSenseNow = micros();

```

```

    // Will assume that the sensed current is the same for the
    // preceding delta time period. Don't need to average it.
    nVIOUT = (uint16_t)nilAnalogRead((char)PIN_VIOUTSENSE);

    // Determine Amp usage

    // using ACS 711 equation the mV / Amp calculation is
    //  $V_{IOUT} = (0.11 * i - (V_{cc}/2)) * V_{cc} * 3.3V$ 
    // with Vcc at 5V
    nInstantCurrent = (nVIOUT * 0.0726) - 0.275;

    // this is missing the * dt * portion of calculation

    // Now calculate Capacity Remaining
    nRemainingAhCapacity = nRemainingAhCapacity - (nInstantCurrent *
        (tCurrentSenseNow - tPrevTimeSense)) >> 12;

    tPrevTimeSense = tCurrentSenseNow;
}

void Power::UpdateInstantVoltage(void)
{
    nInstantVoltage = (uint16_t)nilAnalogRead((char)PIN_VOLTAGESENSE);
}

bool Power::SwitchPowerSupply(ePWRSUPPLY eRequestedSupply)
{
    bool bSwitchSuccess;

    // Record the Power Command for BIT purposes
    ePowerCommand = eRequestedSupply;

    switch(ePowerCommand)
    {
        case PWRSOURCEEXT: // IF EXTERNAL Power Commanded
        {
            // Set the External Power Enable output
            digitalWrite(PIN_POWERENB_EXT, POWERON);

            // Confirm requested power source is now on
            bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEEXT);

            // If Requested source is supplying response should be true
            // then turn off alternate
            if (bSwitchSuccess)
            {
                // Turn off Internal Power Supply
                digitalWrite(PIN_POWERENB_INT, POWEROFF);

                // Confirm Internal Supply is turned off
            }
        }
    }
}

```

```

        bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEINT);

        // Alternate supply should be off so response should
        // be false
        if (!bSwitchSuccess)
        {
            // Record the Power Command for BIT purposes
            ePowerCommand = eRequestedSupply;
            ePowerSource = ePowerCommand;
            bSwitchSuccess = true;
            //return true;
        }
        else
        {
            bSwitchSuccess = false;

            // Debug line
            //bSwitchSuccess = true;
        }
        else
        {
            bSwitchSuccess = false;

            // Debug line
            //bSwitchSuccess = true;
        }
        break;
    }
    case PWRSOURCEINT:    // INTERNAL Power Commanded
    {
        // Output the Internal Power Enable (Active Low)
        digitalWrite(PIN_POWERENB_INT, POWERON);

        //Confirm requested power source is now on
        bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEINT);

        // If Requested source is supplying response should be true
        // then turn off alternate
        if (bSwitchSuccess)
        {
            // Turn off External Power Supply
            digitalWrite(PIN_POWERENB_EXT, POWEROFF);

            // Confirm External Supply is turned off
            bSwitchSuccess = ReadSupplyOutputState(PWRSOURCEEXT);

            // Alternate supply should be off so response should be
            // false
            if (!bSwitchSuccess)
            {
                // Record the Power Command for BIT purposes
                ePowerCommand = eRequestedSupply;
                ePowerSource = ePowerCommand;
            }
        }
    }

```

```

        bSwitchSuccess = true;
        //return true;
    }
    else
    {
        bSwitchSuccess = false;

// Debug line
//bSwitchSuccess = true;
    }
    }
    else
    {
        bSwitchSuccess = false;

// Debug line
//bSwitchSuccess = true;
    }
    break;
}
case PWRSOURCE_OFF:
{
    // Turn off Measured Supply
    // Output the Internal Power Enable (Active Low)
    digitalWrite(PIN_POWERENB_INT, POWEROFF);

    // Output the External Power Enable (Active Low)
    digitalWrite(PIN_POWERENB_EXT, POWEROFF);

    //Confirm power sources are both now off
    // Active LOW, so should be HIGH input
    bSwitchSuccess = ReadSupplyOutputState(PWRSOURCE_OFF);

    if(bSwitchSuccess)
    {
        // Record the Power Command for BIT purposes
        ePowerCommand = PWRSOURCE_OFF;
        ePowerSource = ePowerCommand;
    }
    break;
}
}

// Update the I2C Tx Message Buffer with current Power State
// First Invalidate current message area
g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG3);

g_uI2CUpdate_MsgTx.g_nType1EnumValue = (uint8_t)ePowerSource;

// Now save into I2C Tx Message Common Area
g_cI2C_MsgTx->putMessage(MSG3, &g_uI2CUpdate_MsgTx);

```

```

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG3);

    return bSwitchSuccess;
}

uint16_t Power::ReadCapacityRemain(void) const
{
    return nRemainingAhCapacity;
}

uint16_t Power::ReadInstantVoltage(void) const
{
    return nInstantVoltage;
}

bool Power::ReadSupplyOutputState(ePWRSUPPLY eSupply) const
{
    bool bDigitalState = false;

    switch (eSupply)
    {
        case PWRSOURCE_EXT:
        {
            // Power is ON when Active LOW output on ONST1
            if(digitalRead(PIN_PWRSTATUS_EXT_ONST1) == 0)
            {
                bDigitalState = true;
            }
            break;
        }
        case PWRSOURCE_INT:
        {
            // Power is ON when Active LOW output on ONST2
            if(digitalRead(PIN_PWRSTATUS_INT_ONST2) == 0)
            {
                bDigitalState = true;
            }
            break;
        }
        case PWRSOURCE_OFF:
        {
            // Power is ON when Active LOW output either ONST1 or ONST2
            if(digitalRead(PIN_PWRSTATUS_EXT_ONST1) == 0)
            {
                bDigitalState = false;
            }
            else if(digitalRead(PIN_PWRSTATUS_INT_ONST2) == 0)
            {
                bDigitalState = false;
            }
            else
            {
                // Both are turned off so response success
            }
        }
    }
}

```

```
        bDigitalState = true;
    }

    break;
}
}
return bDigitalState;
}

bool Power::ReadSensorFAULT(void) const
{
    // Current Sensor FAULT condition is Active LOW
    if (!digitalRead(PIN_CURRENTSENSORFAULT))
    {
        // Result is LOW so fault exists, set response = true
        return true;
    }
    else
    {
        // Result is HIGH so NO fault exists, set response = false
        return false;
    }
}

ePWRSUPPLY Power::ReadCommandedSupply(void) const
{
    return ePowerCommand;
}

ePWRSUPPLY Power::ReadRecordedSupply(void) const
{
    return ePowerSource;
}
```


H.35 The BIT.cpp Code

Listing H.32: The BIT source file.

```

/*
 * BIT.cpp
 *
 * Created on: 7 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include "BIT.h"
#include "Commander.h"
#include "PinoutConfigRC.h"

extern Commander* pcCmdr;
extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

// BIT Constructor
BIT::BIT()
{
    nBITCondition = 0;

    // Initialise Allow BIT Checking
    bBITCheckAuthFlag = true;
}

void BIT::BITFlagUpdate(eBITFLAG nBITFlagPosition, bool bCondition)
{
    // the flag to be shifted into BIT Flag position
    uint16_t nFlag = 1;

    // Assert that nBITFlagPosition < 16 positions
    if (nBITFlagPosition > 15)
    {
        BITFlagUpdate(BIT_FLAG_OVRFLW, true);
        //pcCmdr->SetState(BITFAIL);
        return;
    }

    // Shift and Set or clear the BIT Flag position
    if(bCondition)
    {
        // The update is to record a BIT FAIL
        // Need to shift and OR Mask into place
        nBITCondition |= (nFlag << nBITFlagPosition);
        // Set State to BITFAIL
        pcCmdr->SetState(BITFAIL);
    }
}

```

```

    }
    else
    {
        // The update is to record a BIT PASS
        // need to shift, Invert and AND Mask into place
        nBITCondition &= (~(nFlag << nBITFlagPosition));

        // Check if all BIT Flag positions are false
        if(!(nBITCondition & 0xFFFF))
        {
            // Now check if the system is already in BITFAIL
            // State If so then this allows the system to
            // transition out of BITFAIL
            if(pcCmdr->RetrieveState() == BITFAIL)
            {
                pcCmdr->SetState(INITIALISE);
            }
        }
    }

    // Update the I2C Tx Msg Buffer
    // First Invalidate current message area
    g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG4);

    g_uI2CUpdate_MsgTx.g_nType3UIntValue = nBITCondition;

    // Store it into the I2C Message Area
    g_cI2C_MsgTx->putMessage(MSG4, &g_uI2CUpdate_MsgTx);

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG4);
}

uint16_t BIT::ReadBITFlags() const
{
    return nBITCondition;
}

bool BIT::RetrieveBITCheckFlag(void) const
{
    return bBITCheckAuthFlag;
}

void BIT::ModifyBITCheckFlag(bool bFlag)
{
    bBITCheckAuthFlag = bFlag;
}

bool BIT::BITCheck(void)
{
    // Execute BIT

```

```
// Check that the Current Sensor has not set the Over Current
// FAULT flag
// Current Sensor FAULT condition is Active LOW
if (pcCmdr->ReadSensorFAULT())
{
    // Fault exists so set the Current Fault BIT flag
    BITFlagUpdate(CURRENT_FAULT, true);
}
else
{
    // No problem here so clear the Power BIT flag
    BITFlagUpdate(CURRENT_FAULT, false);
}
}
```

H.36 The StateMachine.h Code

Listing H.33: The StateMachine header file.

```

/*
 * StateMachine.h
 *
 * Created on: 2 Nov 2014
 * Author: Ian
 */

#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

// #include "I2CBuffer.h"
#include "Equates.h"

class StateMachine
{
public:
    // Constructor
    StateMachine();
    eSTATE RetrieveState(void) const;

    void SetState(eSTATE eNewState);
    void StoreRequestedStateChange(eSTATE eReqState);
    void CheckFMUCmdStateChange(void);

    void StorePackageState(ePKGSTATE eState);
    bool IsPackageInState(ePKGSTATE eState) const;

    virtual ~StateMachine(){}

private:
    eSTATE eCurrentState;
    eSTATE eFMUCmdState;
    // Adjacency List for Sensor Manager
    uint16_t abStateAdj[NUMSTATES];

    ePKGSTATE eRecordedPKGState;
};

#endif /* STATEMACHINE_H_ */

```

H.37 The StateMachine.cpp Code

Listing H.34: The StateMachine source file.

```

/*
 * StateMachine.cpp
 *
 * Created on: 14 Nov 2014
 * Author: 0050083462
 */

#include <Arduino.h>
#include <stdio.h>
#include "StateMachine.h"
#include "BIT.h"
#include "I2CBuffer.h"
#include "Equates.h"

extern BIT* pcBIT;
extern I2CBuffer *g_cI2C_MsgTx;
extern I2CMsgTx g_uI2CUpdate_MsgTx;

StateMachine::StateMachine()
{
    // Initialise the Adjacency List for the Host Release
    // Controller State Machine
    // NUMSTATES long
    // bit 0 = V0, bit 11 = V11

    // (V0)INITIALISE (V1)PKGCHECK (V2)LOAD (V3)BITFAIL
    // (V4)POWERDOWN (V5)POWERON (V6)ABORT (V7)READYRELEASE
    // (V8)READYCONSENT (V9)RELEASE (V10)GONE (V11)HUNG

    abStateAdj[0] = 2; // B0000000000010 - A(0) = {V1}
    abStateAdj[1] = 28; // B000000011100 - A(1) = {V2, V3, V4}
    abStateAdj[2] = 2; // B0000000000010 - A(2) = {V1}
    abStateAdj[3] = 9; // B0000000000000 - A(3) = {V0, V3}
    abStateAdj[4] = 1064; // B010000101000 - A(4) = {V3, V5, V10}
    abStateAdj[5] = 1176; // B010010011000 - A(5) = {V3, V4, V7, V10}
    abStateAdj[6] = 40; // B000000101000 - A(6) = {V3, V5}
    abStateAdj[7] = 1352; // B010101001000 - A(7) = {V3, V6, V8, V10}
    abStateAdj[8] = 1736; // B011011001000 - A(8) = {V3, V6, V7, V9, V10}
    abStateAdj[9] = 3072; // B110000000000 - A(9) = {V10, V11}
    abStateAdj[10] = 8; // B000000001000 - A(10) = {V3}
    abStateAdj[11] = 1024; // B010000000000 - A(11) = {V10}

    // Set Initial State to
    eCurrentState = INITIALISE;

    // Set FMU Cmd State to invalid state
    eFMUCmdState = NOREQUESTEDSTATE;
}

```

```

eSTATE StateMachine::RetrieveState(void) const
{
    return eCurrentState;
}

void StateMachine::SetState(eSTATE eNewState)
{
    uint8_t nAdjacencyListPos;

    // Validate the requested State Change before so doing
    // Shift the current state adjacency value to the right
    // Checking if new state is a valid transition
    nAdjacencyListPos = abStateAdj[eCurrentState] >> eNewState;

    // Mask off the LSB to obtain boolean answer
    // Only change if new state is included in list
    if ((nAdjacencyListPos & 1))
    {
        // Using the result of adjacency review (State change authority)
        // undertake the State Change if so authorised
        eCurrentState = eNewState;

        // Clear the Incorrect State Transition attempted BIT flag
        pcBIT->BITFlagUpdate(INVALID.STATE.CHG, false);

        // Update the Output message with current state
    }
    else
    {
        // Else incorrect requested State Transition
        // Therefore set BITFail error
        // Set the Incorrect State Transition attempted BIT flag
        pcBIT->BITFlagUpdate(INVALID.STATE.CHG, true);
    }

    // Update the I2C Tx Msg Buffer
    // First Invalidate current message area
    g_cI2C_MsgTx->I2CInvalidateTxMsg(MSG0);

    g_uI2CUpdate_MsgTx.g_nType4EnumValue = eCurrentState;

    // Store it into the I2C Message Area
    g_cI2C_MsgTx->putMessage(MSG0, &g_uI2CUpdate_MsgTx);

    // Finally Validate Msg area
    g_cI2C_MsgTx->I2CValidateTxMsg(MSG0);
}

```

```
void StateMachine::StoreRequestedStateChange(eSTATE eReqState)
{
    // Store away the FMU requested state change
    eFMUCmdState = eReqState;
}

void StateMachine::CheckFMUCmdStateChange(void)
{
    eSTATE eFMUCmdStateCopy;

    eFMUCmdStateCopy = eFMUCmdState;

    if (eFMUCmdStateCopy != NOREQUESTEDSTATE)
    {
        // Invalidate the FMU Cmd
        eFMUCmdState = NOREQUESTEDSTATE;

        // Execute the FMU requested State Change
        SetState(eFMUCmdStateCopy);
    }
}

void StateMachine::StorePackageState(ePKGSTATE eState)
{
    // Store away the Package FMU state
    eRecordedPKGState = eState;
}

bool StateMachine::IsPackageInState(ePKGSTATE eState) const
{
    if (eRecordedPKGState == eState)
    {
        // Yes the Package is in the questioned state
        return true;
    }
    else
    {
        // The states do not match; the Package is in some other state
        return false;
    }
}
```

H.38 The MsgBuff.h Code

Listing H.35: The MsgBuff header file.

```

/*
 * MsgBuff.h
 *
 * Created on: 30 Oct 2014
 * Author: Ian saxby 0050083462
 */

#ifndef MSGBUFF_H
#define MSGBUFF_H

#include <Arduino.h>
#include <stdint.h>

#define NUMBER_ROWS 2    // P+1 where P = 1 No of readers

#define BUFFER_CLR      0 // false
#define MSG_NEW         1 // true

class MsgBuff
{
public:
    MsgBuff();

    void WriteBuff(Message *pMsgIn);

    void ReadBuff(Message *MsgOut);

private:
    // Index to the row with the latest message
    uint8_t Latest;
    // NRows = NUMBER_ROWS = Number of rows in
    // the message buffer
    // Message Buffer
    Message Buff[NUMBER_ROWS][2];
    // Flag identifying requirement to clear
    bool MsgValid[NUMBER_ROWS];
    // Message count for each row
    uint8_t ReaderCnt[NUMBER_ROWS];
    // Column with more up-to-date message
    uint8_t Cl[NUMBER_ROWS];
};

#endif /* MSGBUFF_H */

```


H.39 The MsgBuff.cpp Code

Listing H.36: The MsgBuff source file.

```

/*
 * MsgBuff.cpp
 *
 * Created on: 30 Oct 2014
 * Author: Ian Saxby 0050083462
 */

#include <stdio.h>
#include "MsgBuff.h"

MsgBuff::MsgBuff()
: Latest(0)
{
    // Initialise status pointers to known state
    for(uint8_t i = 0; i < NUMBER_ROWS; i++)
    {
        ReaderCnt[i] = 0;
        Cl[i] = 0;
    }

    // Initialise all buffer contents to known state
    for(uint8_t col = 0; col < 2; col++)
    {
        for(uint8_t i = 0; i < NUMBER_ROWS; i++)
        {
            for(uint8_t j=0; j< MAX_MSG_LENGTH; j++)
            {
                Buff[i][col].u_nData[j] = 0;
            }
        }
    }
}

/* Referenced from Improved Double Buffer Algorithm
 * within Improving Wait-Free Algorithms for Interprocess
 * Communication in Embedded Real-Time Systems.
 * USENIX 2002 Annual Technical Conference Paper
 * Pp 303 - 316 of the proceedings
 */

void MsgBuff::WriteBuff(Message *pMsgIn)
{
    uint8_t i;
    uint8_t cl;

    //for (i = latest; ; i++);
    //{
    //    if(ReaderCnt[i mod NRows] == 0) break;

```

```

        //}
        //cl = not Cl[i];
        //write Buff[i][cl];
        //Cl[i] = cl;
        //Latest = i;

        for (i = Latest; ; i++)
        {
            if (ReaderCnt[i % NUMBER_ROWS] == 0)
                break;
        }

        if (Cl[i] == 0)
            cl = 1;
        else
            cl = 0;

        //Save the Message to the available buffer
        Buff[i][cl] = (*pMsgIn);

        Cl[i] = cl;
        Latest = i;
    }

    void MsgBuff::ReadBuff(Message *pMsgOut)
    {
        // Reader to retrieve latest I2C message output
        uint8_t ridx;
        uint8_t cl;

        // ridx = Latest
        // inc ReaderCnt[ridx]
        // cl = Cl[ridx]
        // read Buff[ridx][cl]
        // dec ReaderCnt[ridx]

        ridx = Latest;
        ReaderCnt[ridx]++;

        cl = Cl[ridx];

        (*pMsgOut) = Buff[ridx][cl];
        ReaderCnt[ridx]--;
    }

```

Appendix I

RDS Interface Control Document

This Appendix includes the RDS Interface Control Document developed as part of this project. The ICD is broken into Seven parts, three each for the RDS FMU to Sensor Manager and Host FMU to Release Controller covering pin allocation, Tx and Rx message definitions respectively.

The last part is the Host RPAS to RDS Interconnector pin assignment.

USQ Final Year Project 2014

Title: Regulating Rescue Package Descent through Controlled Autorotation

RDS Interface Control Document

This document provides for Interface assignment and definition of message data between Host and RDS subsystems. The Sensor Manager (SM) and Host Release Controller (RC) functions are hosted on 16MHz 5V Pro Mini Arduino boards. All communication between FMU and SM or RC is via I2C. The FMU is the Master I2C Node.

Table 1. Sensor Manager Pin Assignment	2
Table 2. RDS FMU to RDS Sensor Manager I2C Rx Messages.....	3
Table 3. Sensor Manager to RDS FMU I2C Tx Messages	5
Table 4. Host Release Manager Pin Assignment	8
Table 5. Host FMU to Host Release Controller I2C Rx Messages	9
Table 6. Host Release Controller to Host FMU I2C Tx Msgs	11
Table 7. Host to RDS Connector Pin out	14

Sensor Manager Pin Assignment (Pro Mini Arduino)

Table 1. Sensor Manager Pin Assignment

Group Capability	Function	Pin Name	Type	Input / Output Direction
Actuator Control				
	Rotor Lock Release	PD6 (6)	PWM	Output
Power Control				
	External Power Enable	PD4 (4)	Level	Output
	Internal Power Enable	PD5 (5)	Level	Output
Interlock				
	Host Connected Input	PD7 (7)	Level	Input
	Commit to Release	PB0 (8)	Level	Output
	Host Connected Input	PB3 (11)	Level	Input
	Debug	PC0 (14)	Level	Input (debnc)
Status				
	Current Sensor $\overline{\text{FAULT}}$ detection	PC1 (15)	Level	Input
	External Power State Status $\overline{\text{ONST1}}$	PC2 (16)	Level	Input
	Internal Power State Status $\overline{\text{ONST2}}$	PC3 (17)	Level	Input
Analogue Sense				
	Current Usage	A7 (22)	Analog	Input
	Voltage Supply	A6 (19)	Analog	Input
Quadrature Input	Quadrature Sensor A	INT0 (2)	Pulse	Input – IRQ
	Quadrature Sensor B	INT1 (3)	Level	Input
Communication				
	I2C SDA	PC5 (A4)	Pulse	Bidirectional
	I2C SCL	PC4 (A5)	Pulse	Bidirectional
	Serial TX	PD0(RX1)	Pulse	Bidirectional
	Serial Rx	PD1(TX0)	Pulse	Bidirectional
Supply				
	Vcc	Vcc	Supply	Power
	Earth	GND	Supply	Power

Table 2. RDS FMU to RDS Sensor Manager I2C Rx Messages

RDS FMU to RDS Sensor Manager I2C Rx Messages (Message in priority order based upon Code value; 1 being highest) (Up to 3 Byte Message): Byte 1 = Command Number Byte 2 = Command Data (if required) Byte 3 = Command Data (if required)		
Commands	Command	Data (if applicable)
Slave I2C Response Setup	1	Data Bytes: 2
		Type: uint8_t
		Byte 1: Response Address Request Address Range: 0 – 9 (see Receive Table)
		Byte 2: Number of bytes to transfer Range: 1 – 3 (see Receive Table)
Commanded State Change	2	Data Bytes: 1
		Type: uint8_t
		Value:
		1 - PowerDown
		2 – Not Used
		3 - ReadyConfirm
		4 - ReadyRelease
		5 - Not Used
		6 - Abort
Weight off Wheels Status	3	Data Bytes: 1
		Type: uint8_t
		Value:
		0 – Weight on Wheels
		1 – Weight off Wheels
Current Usage Initialise	4	Data Bytes: 2
		Type: Unsigned Integer uint16_t
		Range: 0 – 1300 mAh
		Byte 1: LSB
		Byte 2: MSB
State Change Acknowledge	5	Data Bytes: 0
Debug Interface	6	Data Bytes: 1
		Type: uint8_t
		Value:

		1 - PowerDown
		2 - RotorLock Open
		3 - ReadyConfirm
		4 - ReadyRelease
		5 - RotorLock Close
		6 - Abort

Table 3. Sensor Manager to RDS FMU I2C Tx Messages

Sensor Manager to RDS FMU I2C Tx Messages		
Data function	Receiver Index (Decimal Offset)	Description
Sensor Manager	0	Total bytes: 2
Sensor Manager State	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 0 Msg 0
		Units: Message Number
	(1)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value:
		0 Initialise
		1 Prepared
		2 BITFailure
		3 PowerDown
		4 Abort
		5 ReadyConfirm
		6 ReadyRelease
		7 Separation
		8 Deploy
		Units: Not Applicable
Quadrature Data	1	Total bytes: 3
Quadrature Speed and Direction	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 1 Msg 1
		Units: Message Number
	(1)	Internal Buffer Offset Address Value:LSB byte
	(2)	Internal Buffer Offset Address Value:MSB byte
		Type: Integer int16_t
		Range: -3000 – 3000
		-3000 – 3000 Angular Speed
		Units: RPM
Sensor Data	2	Total bytes: 3
Internal Battery Current Usage	(0)	Internal Buffer Offset Address Value

Sensor Manager to RDS FMU I2C Tx Messages		
Data function	Receiver Index (Decimal Offset)	Description
		Type: Enumerated uint8_t
		Value: 2 Msg 2
		Units: Message Number
	(1)	Internal Buffer Offset Address Value:LSB byte
	(2)	Internal Buffer Offset Address Value:MSB byte
		Type: Unsigned Integer uint16_t
		Range: 0 – 1300
		Units: mAh
Power Mode	3	Total bytes: 2
Power Mode State	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 3 Msg 3
		Units: Message Number
	(1)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value:
		0 Using External Power
		1 Using Internal Power
		2 Both Power Supplies disabled
		Units: Not Applicable
Built In Test	4	Total bytes: 3
Built In Test Result	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 4 Msg 4
		Units: Message Number
	(1)	Internal Buffer Offset Address Value:LSB byte
	(2)	Internal Buffer Offset Address Value:MSB byte
		Type: Enumerated uint16_t
		Value:
		SUPPLY_INT, // BIT Pos 0
		SUPPLY_EXT, // BIT Pos 1
		SUPPLY_OFF, // BIT Pos 2
		CURRENT_FAULT, // BIT Pos 3
		PACKAGE_INTERLOCK_GONE, // BIT Pos 4
		BATCAP_BELOW_WARN, // BIT Pos 5
		FMU_NOT_COMM, // BIT Pos 6
		INVALID_STATE_CHG, // BIT Pos 7
		SUPPLYNOTCMD, // BIT Pos 8

Sensor Manager to RDS FMU I2C Tx Messages		
Data function	Receiver Index (Decimal Offset)	Description
		RXBUFF_OVRFLW, // BIT Pos 9 FIFO_OVRFLW, // BIT Pos 10 HOOK_LOCKED, // BIT Pos 11 SPARE1, // BIT Pos 12 SPARE2, // BIT Pos 13 SPARE3, // BIT Pos 14 BIT_FLAG_OVRFLW // BIT Pos 15
		Units: Not Applicable
Interlocks	5	Total bytes: 4
RDS Interlock Status	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 5 Msg 5
		Units: Message Number
	(1)	Internal Buffer Offset Address Value
		Type: Boolean
		Value:
		false Gone
		true Connected
		Units: Not Applicable
Debug Interlock Status	(2)	Internal Buffer Offset Address Value
		Type: Boolean
		Value:
		false Not asserted
		true Debug Mode Asserted
		Units: Not Applicable
Commit to Release Interlock Status	(3)	Offset Address Value
		Type: Boolean
		Value:
		false Not Asserted
		true Commit to Release Asserted
		Units: Not Applicable

Host Release Manager Pin Assignment (Pro Mini Arduino)

Table 4. Host Release Manager Pin Assignment

Group Capability	Function	Pin	Type	Input / Output Direction
Actuator Control				
	RDS Release Lock/Unlock	PD6 (6)	PWM	Output
	RDS Release Actuator	PB1 (9)	PWM	Output
	Cable Retraction / extension	PB2 (10)	PWM	Output
Power Control				
	RDS Power Enable	PD4 (4)	Level	Output
	Spare	PD5 (5)	Spare	Spare
Interlock				
	RDS Connected Wrap Output	PD7 (7)	Level	Output
	Commit to Release	PB0 (8)	Level	Input
	RDS Connected Wrap Input	PB3 (11)	Level	Input
	Debug	PC0 (14)	Level	Input (debnc)
	Load RDS	PB4 (12)	Level	Input (debnc)
Status				
	Current Sensor $\overline{\text{FAULT}}$ detection	PC1 (15)	Level	Input
	External Power State Status $\overline{\text{ONST1}}$	PC2 (16)	Level	Input
	Internal Power State Status $\overline{\text{ONST2}}$	PC3 (17)	Level	Input
Analogue Sense				
	Current Usage	ADC7 (A7)	Analog	Input
	Voltage Supply	ADC6 (A6)	Analog	Input
Communication				
	I2C SDA	PC4 (A4)	Pulse	Bidirectional
	I2C SCL	PC5 (A5)	Pulse	Bidirectional
	Serial TX	RXD	Pulse	Bidirectional
	Serial Rx	TXD	Pulse	Bidirectional
Supply				
	Vcc	Vcc	Supply	Power
	Earth	GND	Supply	Power

Table 5. Host FMU to Host Release Controller I2C Rx Messages

Host FMU to Host Release Controller I2C Rx Messages (Message in priority order based upon Code value; 1 being highest) (Up to 3 Byte Message): Byte 1 = Command Number Byte 2 = Command Data (if required) Byte 3 = Command Data (if required)		
Commands	Command	Data (if applicable)
Slave I2C Response Setup	1	Data Bytes: 2
		Type: uint8_t
		Byte 1: Response Address Request Address Range: 0 – 9 (see Receive Table)
		Byte 2: Number of bytes to transfer Range: 1 – 4 (see Receive Table)
Commanded State Change	2	Data Bytes: 1
		Type: Enumerated uint8_t
		Value:
		1 – Host Power On
		2 – Host Power Off
		3 - ReadyRelease
		4 - Release
		5 - Abort
Weight off Wheels Status	3	Data Bytes: 1
		Type: Enumerated uint8_t
		Value:
		0 – Weight on Wheels
		1 – Weight off Wheels
RDS State	4	Data Bytes: 1
Debug Commands		
RDS Release Unlock / Lock	5	Data Bytes: 1
		Type: Enumerated uint8_t
		Value:
		0 – Lock
		1 – Unlock
Cable Retraction / Extension	6	Data Bytes: 1
		Type: Enumerated uint8_t

		Value:
		0 – Retract
		1 – Extend
RDS Release Action	7	Data Bytes: 1
		Type: Enumerated uint8_t
		Value:
		0 – Hooks Closed
		1 – Hooks Open

Table 6. Host Release Controller to Host FMU I2C Tx Msgs

Host Release Controller to Host FMU I2C Tx Messages		
Data function	Receiver Address (Decimal Offset)	Description
Sensor Manager	0	Total bytes: 2
Host Release Controller State	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 0 Msg 0
		Units: Message Number
	(1)	Offset Address Value
		Type: Enumerated uint8_t
		Value:
		0 Initialise
		1 BITFail
		2 PackageCheck
		3 Load
		4 PackageConnectedOff
		5 PackageConnectedOn
		6 ReadyRelease
		7 ReleaseConsent
		8 Release
		9 Gone
		10 Hung
		Units: Not Applicable
Sensor Data	1	Total bytes: 3
Host Battery Supply Amount	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 1 Msg 1
		Units: Message Number
	(1)	Offset Address Value: LSB byte
	(2)	Offset Address Value: MSB byte
		Type: Unsigned Integer uint16_t
		Range: 0 – 2600
		Units: mAh
		The current RDS mAh usage when powered by Host Power supply
Power Mode	2	Total bytes: 2
Power Mode State	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 2 Msg 2

Host Release Controller to Host FMU I2C Tx Messages		
Data function	Receiver Address (Decimal Offset)	Description
		Units: Message Number
	(1)	Offset Address Value
		Type: Boolean
		Value:
		false Host Power Off
		true Host Power On
		Units: Not Applicable
Built In Test	3	Total bytes: 2
Built In Test Result	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 3 Msg 3
		Units: Message Number
	(1)	Offset Address Value
		Type: Enumerated int8_t
		Value:
		SUPPLY_INT, // BIT Pos 0
		SUPPLY_EXT, // BIT Pos 1
		SUPPLY_OFF, // BIT Pos 2
		CURRENT_FAULT, // BIT Pos 3
		PACKAGE_INTERLOCK_GONE, // BIT Pos 4
		BATCAP_BELOW_WARN, // BIT Pos 5
		FMU_NOT_COMM, // BIT Pos 6
		INVALID_STATE_CHG, // BIT Pos 7
		SUPPLYNOTCMD, // BIT Pos 8
		RXBUFF_OVRFLW, // BIT Pos 9
		FIFO_OVRFLW, // BIT Pos 10
		HOOK_LOCKED, // BIT Pos 11
		SPARE1, // BIT Pos 12
		SPARE2, // BIT Pos 13
		SPARE3, // BIT Pos 14
		BIT_FLAG_OVRFLW // BIT Pos 15
		Units: Not Applicable
Interlocks	4	Total bytes: 3
Host Interlock Status	(0)	Internal Buffer Offset Address Value
		Type: Enumerated uint8_t
		Value: 4 Msg 4
		Units: Message Number
	(1)	Offset Address Value

Host Release Controller to Host FMU I2C Tx Messages		
Data function	Receiver Address (Decimal Offset)	Description
		Type: Enumerated int8_t
		Value:
		0 Gone
		1 Connected
		Units: Not Applicable
Commit to Release Interlock Status	(2)	Offset Address Value
		Type: Enumerated int8_t
		Value:
		0 Gone
		1 Commit to Release Asserted
		Units: Not Applicable

Host to RDS Connector Pin out**Top (1 – 5) = Shorter edge; Bottom (11 – 15) = Wider edge***Table 7. Host to RDS Connector Pin out*

1	2	3	4	5
RDS Interlock	Host Power	TX	Host GND	Commit To Release
6	7	8	9	10
Commit To Release	RX	Debug	TX	Host Interlock
11	12	13	14	15
Host Interlock	Host Power	RX	Host GND	RDS Interlock