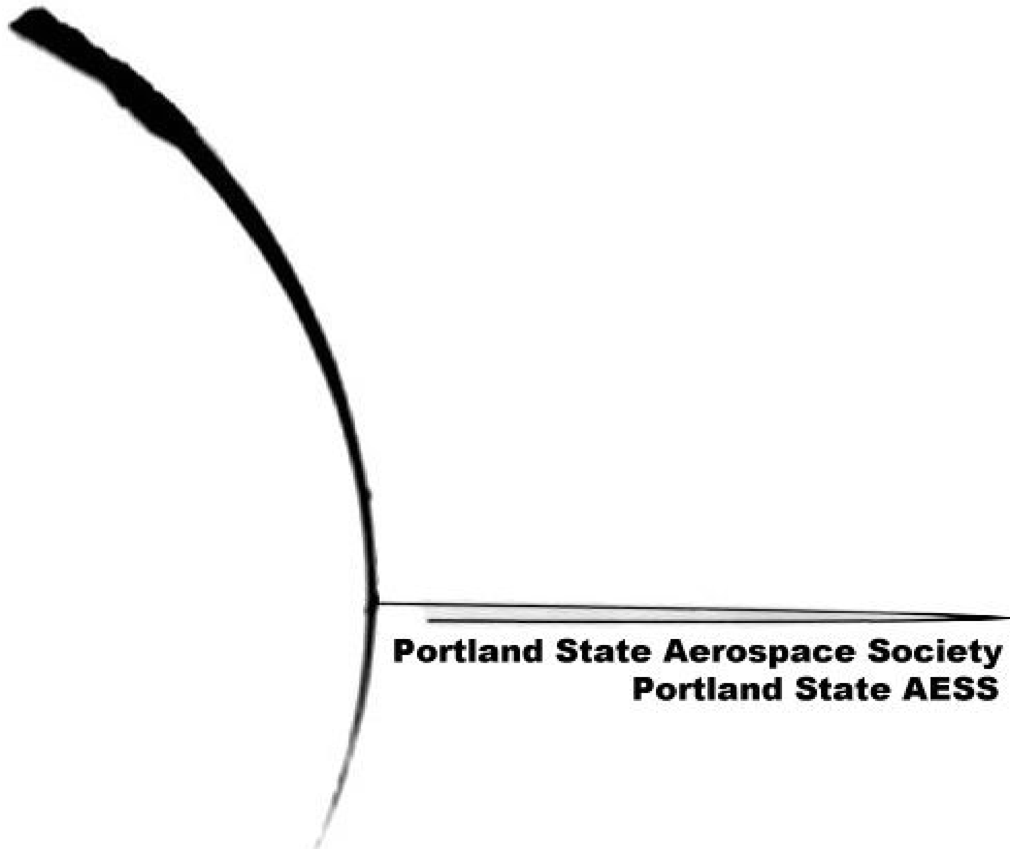
PSAS Avionics System Design Project

Michael Kennan Matt McFadden



PSAS Avionics System Design Project

Spring 1999

Revision 3.1

Michael Kennan
Matt McFadden

Faculty Advisor:	Dr. Lee Casperson
Industry Advisor:	Andrew Greenberg

Acknowledgements

This project is presented in partial fulfillment of the authors Senior Project requirements for the Electrical and Computer Engineering Department at Portland State University. We would like to thank our Faculty Advisor Dr. Lee Casperson and our Industry Advisor Andrew Greenberg for their assistance, guidance, and support throughout the course of the project.

Abstract

To meet the goal of developing an Inertial Navigation System for future rocket flights, the Portland State Aerospace Society (PSAS) requires a testbed system for developing, characterizing, and qualifying sensor and control packages. The system is required to be scalable and modular while meeting weight constraints and environmental standards such as force and temperature tolerance.

This document provides an overview of previous amateur rocketry research performed by the PSAS and describes the authors' design of a robust, scalable, and evolvable rocket avionics architecture. Emphasis is made on modularity, redundancy of critical hardware and software mechanisms, testability, and the maintenance of an open architecture to accommodate future changes to the system.

This is a living document that represents a work in progress. Revisions will continue to be made throughout the development of the LV2 rocket project.

Table of Contents

ACKNOWLEDGEMENTS **1**

ABSTRACT **2**

TABLE OF CONTENTS **3**

TABLE OF FIGURES **5**

REVISIONS **6**

INTRODUCTION **7**

BACKGROUND **8**

- AESS/PSAS 8
- LAUNCH VEHICLE ZERO (LV0) 9
- LAUNCH VEHICLE ONE (LV1) 12
- LAUNCH VEHICLE TWO (LV2) 21

PROBLEM STATEMENT **23**

CONSTRAINTS **24**

DESIGN REQUIREMENTS **25**

- OVERALL 25
- SENSORS: 25
  - *Proprioceptive sensors* *25*
  - *External sensors* *25*
- COMMUNICATIONS: 26
  - *Uplink* *26*
  - *Downlink* *26*
  - *Communications subsystem* *26*
- DATA STORAGE: 26
- FLIGHT COMPUTER: 26

ARCHITECTURAL CONSIDERATIONS **27**

- CENTRAL/MONOLITHIC ARCHITECTURE 27
- SERIAL ARCHITECTURE (DIRECT) 28
- SERIAL ARCHITECTURE (ARBITRATED) 29
- DISTRIBUTED ARCHITECTURE 30
- MODULAR ARCHITECTURE 31
- SERIAL ARCHITECTURE WITH HARDWARE ARBITRATION 32

IMPLEMENTATION **33**

- CANBUS OVERVIEW 34
  - *How does CAN work?* *34*
  - *Identifiers* *36*
  - *Addressing and Arbitration* *36*
  - *Error-Checking* *36*
  - *Realization* *37*
- MODULE OVERVIEW 40

FLIGHT COMPUTER (FC) ... 42
Hardware ... *42*
Messaging ... *44*
Software ... *48*
COMMUNICATION COMPUTER (CC) ... 50
Hardware ... *50*
Messaging ... *51*
Software ... *53*
INERTIAL MEASUREMENT UNIT (IMU) ... 54
Hardware ... *54*
Messaging ... *58*
Software ... *59*
DATA ACQUISITION MODULE (DAQ) ... 60
Hardware ... *60*
Messaging ... *64*
Software ... *65*
FLIGHT RECORDER (FR) ... 66
Hardware ... *66*
Messaging ... *67*
Software ... *69*
IGNITERS ... 72
Hardware ... *72*
Messaging ... *73*
Software ... *74*
OPERATIONS ... 75
Messages ... *75*
Modes ... *77*
GROUND SUPPORT ... 83

PLANS FOR THE FUTURE ... 84

APPENDIX A: SYSTEM SCHEMATICS ... 85

APPENDIX B: MESSAGE IDENTIFIERS ... 95

APPENDIX C: REFERENCES ... 99

Table of Figures

Figure 1: *Launch Vehicle Zero (LV0)* 9
Figure 2: *Launch Vehicle Zero (LV0) payload block diagram* 10
Figure 3: *Launch Vehicle Zero (LV0) altimeter data* 11
Figure 4: *Launch Vehicle One (LV1)* 12
Figure 5: *Launch Vehicle Zero (LV1) payload block diagram* 13
Figure 6: *Additional specification details for LV1* 14
Figure 7: *Flight profile for LV1* 15
Figure 8: *LV1 Z-axis acceleration data* 17
Figure 9: *Velocity profile for LV1* 19
Figure 10: *Position profile for LV1* 19
Figure 11: *Pressure sensor altitude profile for LV1* 20
Figure 12: *Possible Configuration of LV2* 22
Figure 13: *Monolithic architecture block diagram* 27
Figure 14: *Serial ring architecture block diagram* 28
Figure 15: *Detail of arbitrated serial bus* 29
Figure 16: *Distributed architecture block diagram* 30
Figure 17: *Modular architecture block diagram* 31
Figure 18: *CANbus architecture block diagram* 32
Figure 19: *Example of arbitration/prioritization scheme* 35
Figure 20: *Internal structure of linear accelerometers* 55
Figure 21: *Frequency-Phase characteristic* 56
Figure 22: *Connecting to the A/D converter* 57
Figure 23: *Integration approximation in the IMU* 59
Figure 24: *Power Monitoring Circuit* 61
Figure 25: *Separation Sensor Schematic* 61
Figure 26: *Temperature Sensor Schematic* 62
Figure 27: *Pressure Sensor Schematic* 63
Figure 28: *DAQ Flowchart* 65
Figure 29: *Programming the Flash RAM* 70
Figure 30: *Program flow chart for the Flight Recorder* 71
Figure 31: *Initialization State Flow Diagram* 78
Figure 32: *Preflight State Flow Diagram* 79
Figure 33: *Launch State Flow Diagram* 80
Figure 34: *Flight State Flow Diagram* 81
Figure 35: *Recovery State Flow Diagram* 82
Figure A1: *System Overview Schematic* 86
Figure A2: *CAN Interface Schematic* 87
Figure A3: *Flight Computer Schematic* 88
Figure A4: *Communication Computer Schematic* 89
Figure A5: *Inertial Measurement Unit Schematic* 90
Figure A6: *Data Acquisition Module Schematic* 91
Figure A7: *Flight Recorder Schematic* 92
Figure A8: *Flash RAM Unit Schematic* 93
Figure A9: *Igniter Module Schematic* 94

Revisions

Revision Number	Notes	Date
1.0	Initial Draft Proposal Created	1/16/99
1.1	First AESS/PSAS submission	2/5/99
1.2	Second AESS/PSAS submission	2/12/99
1.3	First academic submission to Dr. Casperson	2/18/99
2.0	Initial Draft of Design Document Created	3/18/99
3.0	CANbus implementation, first draft	6/1/99
3.1	CANbus implementation, final submission	6/25/99

Introduction

The Portland State Aerospace Society (PSAS) has been involved in Amateur rocketry developing a microcomputer-controlled intelligent rocket avionics package. The current design has a monolithic architecture specific to the present launch vehicle. To meet future launch vehicle requirements (and as an exercise in many aspects of aerospace systems engineering), we have designed a new avionics system that emphasizes modularity and testability and will aid in the development of an Inertial Navigation System to be used on future flights.

This document will provide an overview of the previous amateur rocketry work by the PSAS, a look at the work that is happening in the present, and an outline of our plans for the future. In the 'Background' section we will discuss prior rocket projects, the state of the art in amateur rocketry today, and our need for a more advanced system. We will then define the problem at hand, discussing the design requirements and constraints that apply to the avionics system design project and examining candidate system architectures. In the 'Implementation' section, we describe our avionics system design and provide an explanation of the architecture we chose to use. In the final section, we will discuss our plans for the future and the direction the PSAS is headed.

Background

AESS/PSAS

The Portland State University (PSU) student chapter of the Aerospace and Electrical Systems Society (AESS) was created in the summer of 1997. The AESS is a technical society of the Institute of Electrical and Electronics Engineers (IEEE), an international professional organization for electrical engineers. The PSU chapter of the AESS was the first student chapter of the society in the United States.

In the Fall of 1998, the Portland State Aerospace Society (PSAS) was formed as a university-recognized student group, allowing anyone from the community (not just AESS/IEEE members) to contribute to the project.

As a group of engineering students and professionals, the PSAS seeks to gain aerospace systems engineering experience and develop "real world" engineering skills by tackling a challenging project.

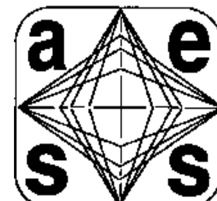
The ultimate aim of the group is to develop a sophisticated avionics system for high-powered amateur rockets. Many other groups and individuals involved in amateur rocketry are actively advancing “traditional” rocket body and motor design, but it is apparent that few are pursuing the development of any sort of advanced avionics or scientific instrumentation payload.

The original goal of the PSAS was to develop a proof-of-concept vehicle to demonstrate the feasibility of the following:

- broadcasting live video using Amateur Television,
- using the audio channel of the video stream to transmit telemetry data, and
- using inexpensive sensors under microprocessor control to gather scientific data.

This goal was met in June 1998 with the development of Launch Vehicle Zero.

PORTLAND STATE UNIVERSITY



Launch Vehicle Zero (LV0)

Launch Vehicle Zero (LV0) was the first rocket project developed by the PSAS. LV0's payload module was composed of 3 main systems: The Amateur Television Video (ATV) transmitter system, a vertically aligned (Z-axis) solid-state accelerometer that was fed into a 300 bps digital data down link, and an altimeter.

The rocket-based ATV system was composed of a miniature NTSC black-and-white CCD camera and a transmitter tuned to the 70-cm band allocated for amateur television and radio. The signal from the CCD was fed into the transmitter, which broadcasted at 440 MHz to a ground station equipped with the appropriate antenna and receiver configuration. On the ground, the video signal was viewed real-time and recorded onto videotape.

The Analog Device's ADXL50 solid-state (micro-machined) accelerometer was incorporated in a telemetry package designed to measure the rocket's acceleration profile and transmit the data to the ground station. To accomplish this, the output from the accelerometer was digitized by a PIC16C73A microcontroller and directed to a 300 bps modem. The analog signal from the modem was transmitted to the ground via the audio channel of the ATV signal. On the ground the telemetry data was recorded onto a computer for later analysis.

Also on board the rocket was a commercially available altimeter interfaced with a Motorola 68HC11 microcontroller and a 12-bit A/D converter. All data from the altimeter was stored in battery backed RAM, and retrieved for later analysis.

Specifications for LV0 include:

Airframe	:	Cardboard with 3 layers fiberglass		
Length	:	72 inches		
Weight	:	12.2 lbs.		
Motor	:	700 Ns solid propellant motor		
Recovery	:	Payload	-	4-ft parachute
		Body	-	3-ft parachute

Figure 1: Launch Vehicle Zero (LV0)

LV0 contained no flight-sequencing computer. Separation and recovery were handled by a chemically timed motor ejection charge. The block diagram of the LV0 payload is shown below in Figure 2.

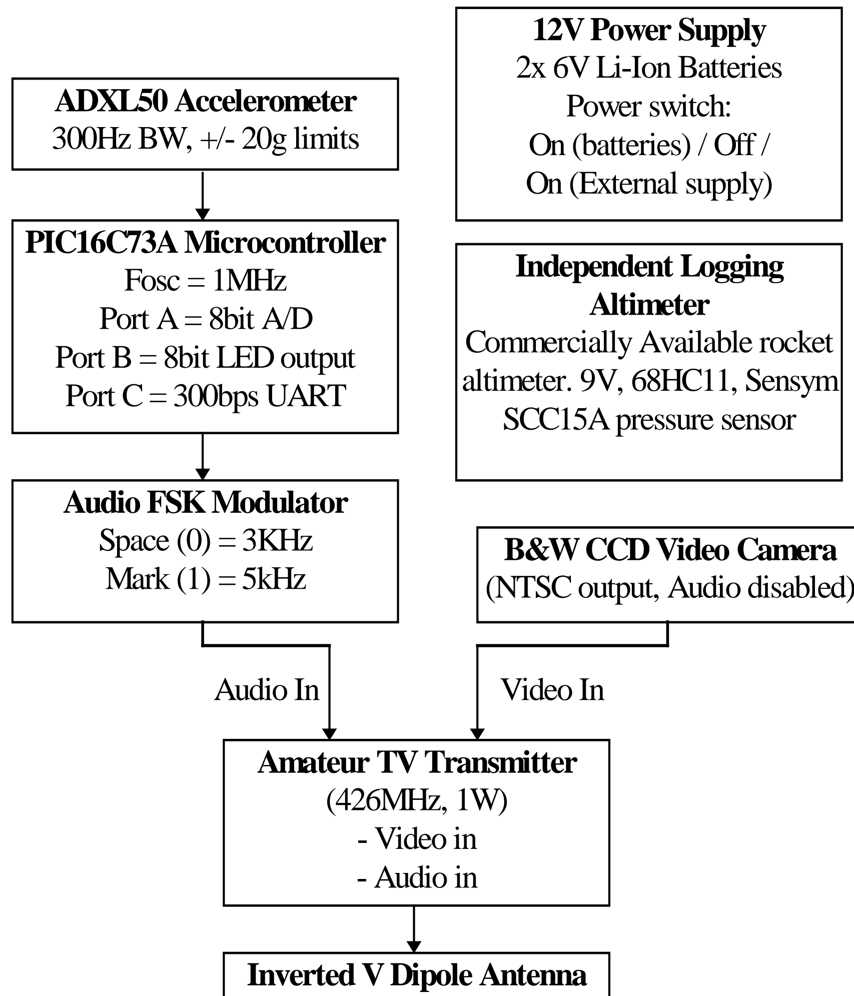


Figure 2: Launch Vehicle Zero (LV0) payload block diagram

The launch of LV0 occurred on June 7th 1998 in Monroe, Washington. The rocket reached an altitude of 1200 feet, successfully transmitting live video to the ground and internally logging altimeter data throughout the flight. The separation and recovery system worked as predicted, but the digital downlink system failed due to a short in the data transmit line.

The altimeter data from the LV0 flight is shown on the following page.

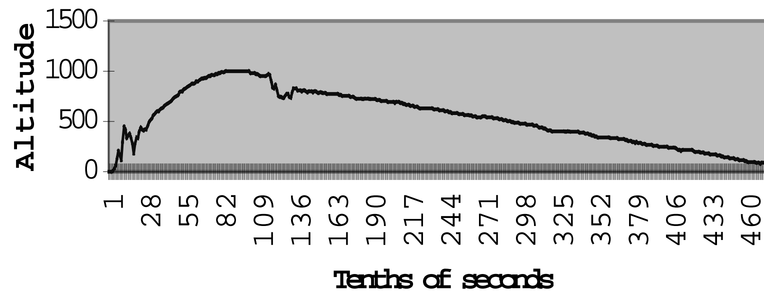


Figure 3: Launch Vehicle Zero (LV0) altimeter data

From the experience with LV0, the PSAS developed a number of improved design criteria for future launches. These include:

- robust interconnects,
- improved flight control via onboard flight computer, and
- emergency flight control via radio uplink.

Implementation of these criteria in a new rocket design led to the development of Launch Vehicle 1 (LV1).

Launch Vehicle One (LV1)

In addition to addressing the issues raised by LV0, the PSAS felt that it was necessary to begin developing basic avionics systems that could grow with future designs, and to include a scientific instrumentation payload for flight profiling and data logging.

Specifications for LV1 include:

Airframe	:	Carbon fiber body with fiberglass payload	
Length	:	132 inches	
Weight	:	46 lbs.	
Motor	:	Up to 10,000 Ns solid propellant motor	
Recovery	:	Payload	- (3) 3.5 ft parachutes
		Body	- 2 stage parachute (drogue/main)

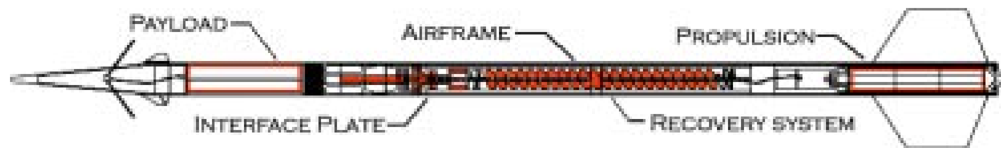


Figure 4: Launch Vehicle One (LV1)

LV1 subsystems include a payload module and an Interface Plate Release System (IPRS).

The payload module includes the following components:

- A color CCD video camera with an ATV transmitter,
- a 2400bps digital data downlink for telemetry data,
- a 2m amateur radio DTMF-activated uplink for manual recovery system control,
- accelerometers and rotational gyro's measuring 3 linear axes and 3 rotational axes,
- pressure and temperature sensors, and
- a flight computer powered by a PIC17C42 microcontroller for flight sequencing.

The accelerometers and rotational gyros form an Inertial Measurement Unit (IMU) with 6 degrees-of-freedom (6-DOF) which will serve as a prototype for the development of a future Inertial Navigation System (INS).

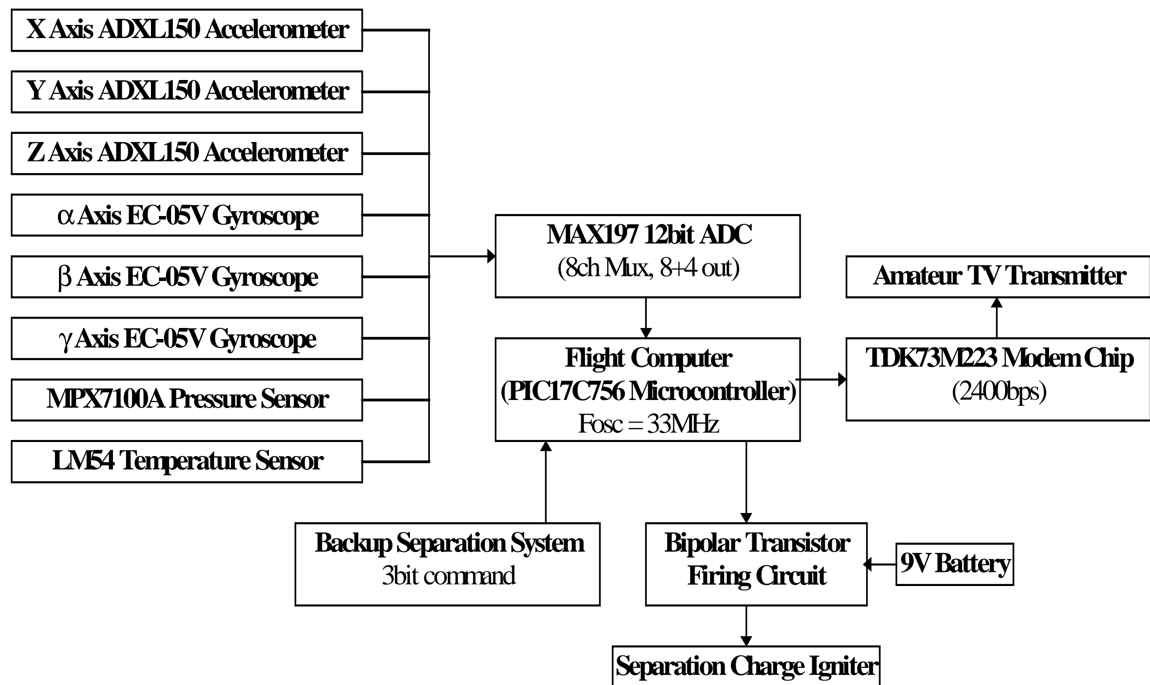


Figure 5: Launch Vehicle Zero (LV1) payload block diagram

The Interface Plate Release System (IPRS) is responsible for deploying the main rocket body's recovery chutes and is independent of the main payload module. As shown in the flight profile (shown on the next page in Figure 6), the main rocket body falls toward the earth until the pressure sensors or internal timers indicate that it is time to deploy the chutes. Earlier release of the chutes would result in excessive 'hang-time', complicating location and retrieval of the rocket body; later release of the chutes (known as a 'lawn-dart' landing) would simplify location and retrieval of the rocket body but could result in unacceptable damages.

Additional LV1 details are shown in Figure 6, on the next page:

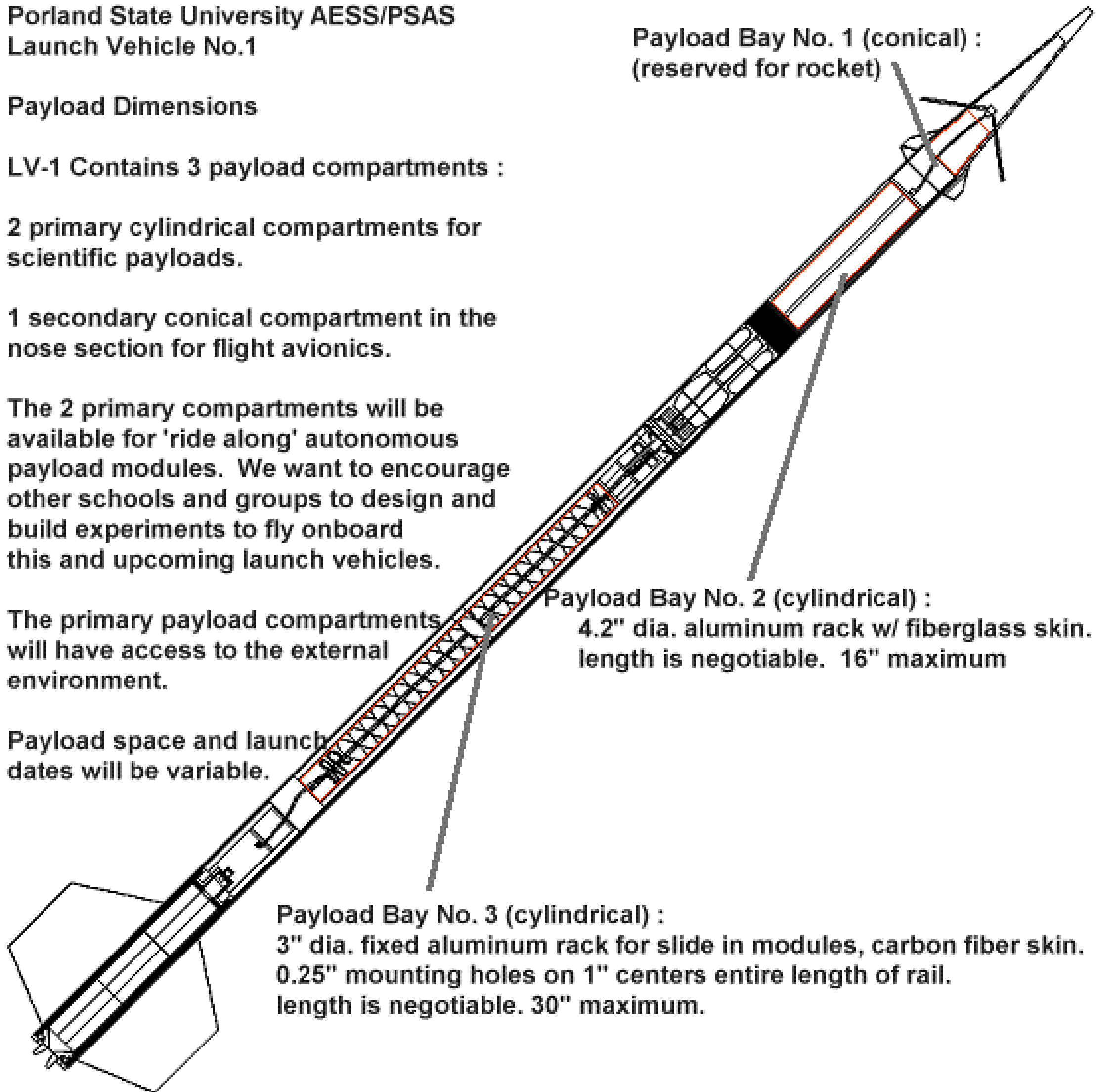


Figure 6: Additional specification details for LV1

The planned flight profile for LV1 is shown below.

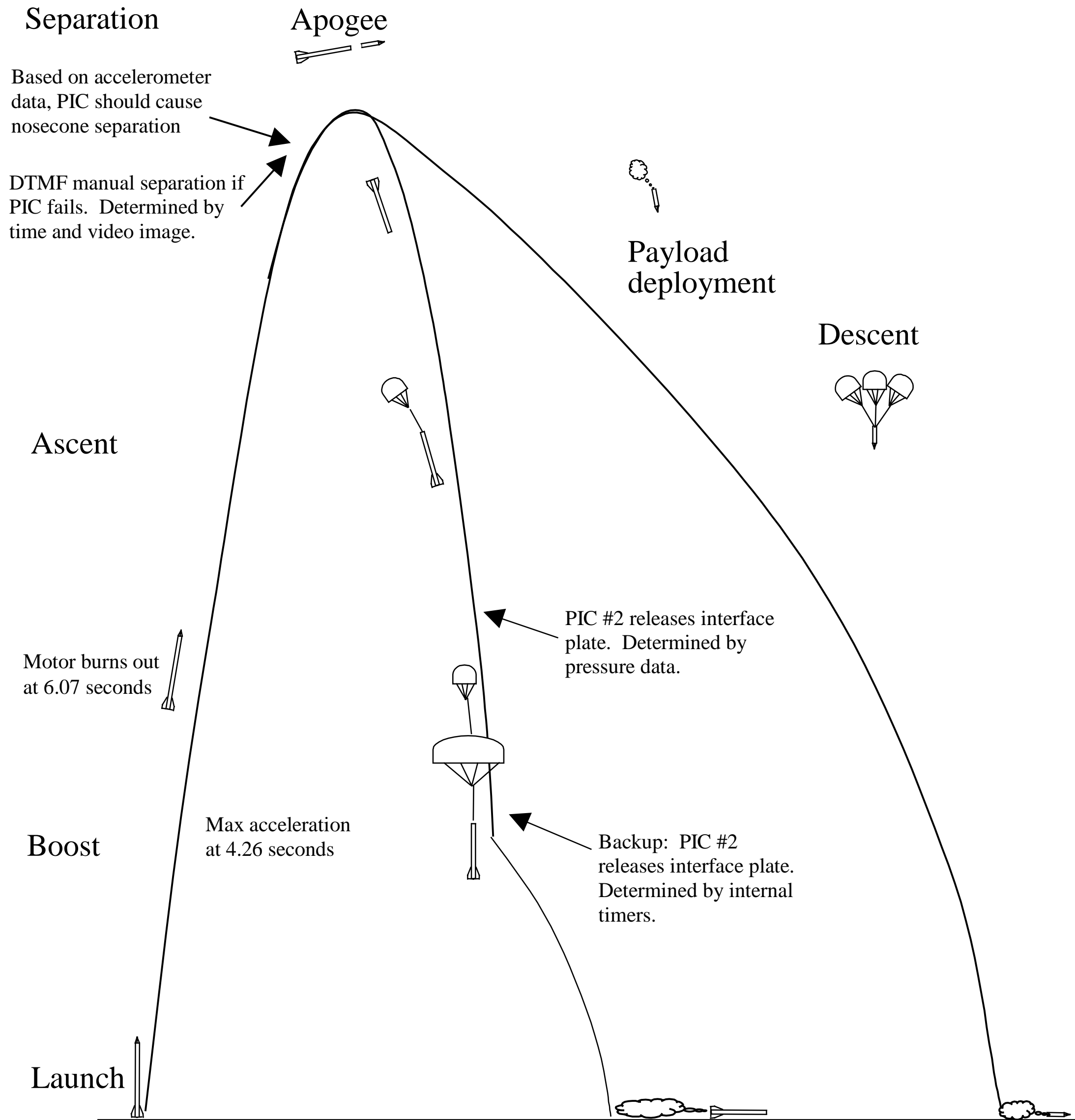


Figure 7: Flight profile for LV1

The launch of LV1 occurred on April 11th 1999 at the PSAS launch site near Millikan, Oregon. The rocket reached an altitude of slightly more than 12000 ft, successfully transmitting data and video to the ground support station throughout the flight.

A total of 11,252 Z-axis acceleration data points were received from the rocket throughout the flight, sampled by the MAX197 12-bit A/D converter. Of these, there were approximately 230 obviously erroneous values resulting from bit-flips; 60% of these errors were bit-sets (a false '1' detected) and 40% were bit-clears (a false '0' detected). The errors were corrected by determining whether or not an outlying data point differed from the general trend by the addition or subtraction of a large power of 2 corresponding to one of the 4 most significant bits (MSBs). Obviously it is likely that there were bit-flips in the lower bits as well, but these would be more difficult to detect since the deviation from the general trend would be less noticeable.

If we assume that bit-flip errors can be reliably detected only if they occur in the 4 most significant bits, then only one third of the errors were detected and the total reception error is (3 times the detected error) = 3 * 230 errors / 11252 total points = 6% error. Future systems must include cyclical redundancy check (CRC), error correction code (ECC), or some other mechanism which is capable of detecting and possible correcting this sort of error.

Figure 8 (next page) shows the Z-axis acceleration profile for the flight of LV1. The numbered labels denote interesting portions of the flight; these are discussed below. A note on terminology: all acceleration descriptions are referenced to the ground, i.e. 'negative acceleration' is the result of forces that push the body upwards or create a downward pull on the sensor (such as the rocket motor, the separation charge, and the earth's gravitational field), and 'positive acceleration' is the result of forces that push the body downward (such as wind resistance effects and the opening of the chutes while the rocket was still ascending).

1. Boost phase

   Prior to liftoff, the only acceleration experienced by the rocket was a steady 1 'g' downward. At time t = 0s, the motor ignited and the rocket left the launchpad. Maximum acceleration was reached at time t = 2.17s, when the rocket was experiencing 7.27 'g's downward.

2. Ascent phase

   As the motor continued to burn through its thrust curve, the diminishing effects of the motor thrust and the increasing effects of air resistance (which is proportional to the cube of the velocity) reduced the acceleration. At t = 6.3s, the air resistance component of the acceleration began to exceed the thrust component and the net acceleration on the avionics system became positive.

Shortly thereafter (at time t = 6.54s) the peak velocity of 596.2mph (0.795 Mach) was reached. As the rocket continued upwards, the effects of the air drag continued to negate the acceleration from the motor, as shown by the asymptotic decay of the graph towards zero.

3. Separation

   At time t = 26.75s, the payload/avionics system was forcibly separated from the rocket body by the separation charge, giving the payload a slight boost in upward velocity.

4. Chute Deployment

   As the chutes deploy from the rear of the module, the payload experiences a sudden deceleration.

5. Descent

   The avionics system floats to the ground. Acceleration oscillates around 1 ‘g’ downward due to the swinging of the avionics system on the three payload chutes. Note that the graph shows only the first 39 seconds of flight- the actual time of flight was 466 seconds.

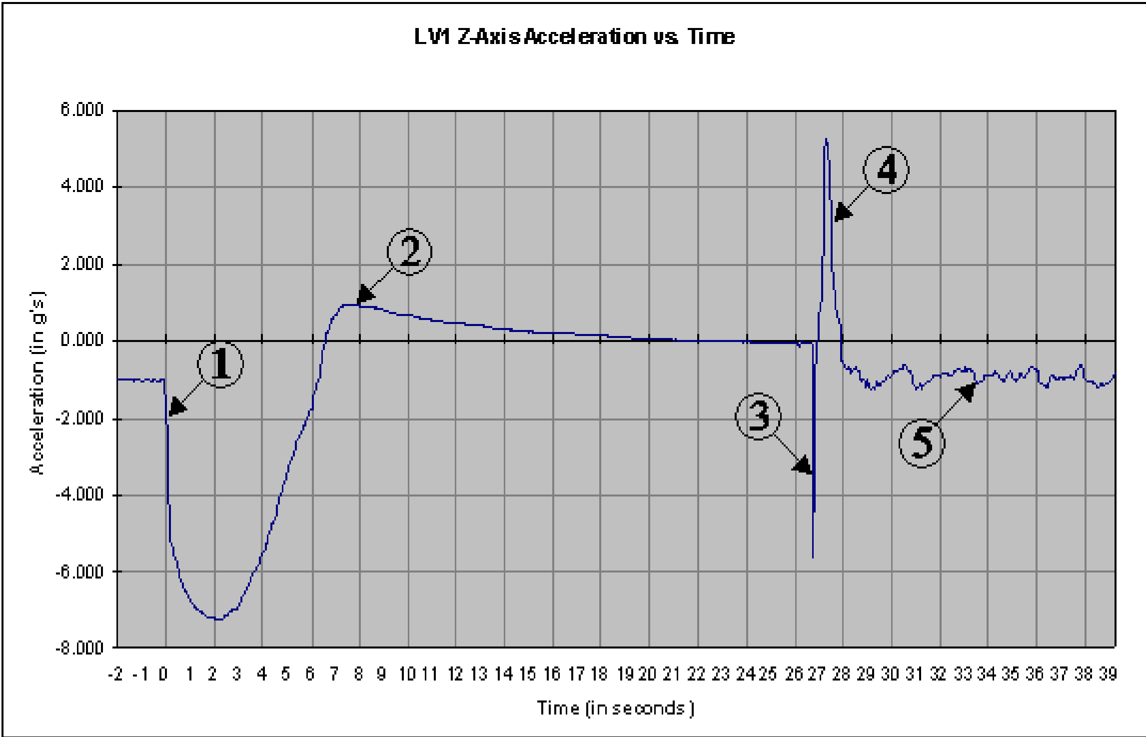


Figure 8: LV1 Z-axis acceleration data

Figure 9 (shown on the next page) shows the Z-axis velocity profile for the flight of LV1. This data was obtained by integrating the acceleration data (disregarding the constant 1'g' acceleration due to gravity). Despite the accumulative errors that result from the integration process, this profile shows the expected behavior:

1. Boost phase

   Velocity was initially zero as the rocket sat on the launch pad. At time t = 0s, the motor ignited and the rocket accelerated upwards, rising steadily throughout the boost phase.

2. Ascent phase

   At 6.54s, the motor burn is complete and the air resistance exerts its effect on the speed of the rocket throughout the coast phase. The rocket is still continuing upwards as air resistance and gravity erode its momentum.

3. Separation

   At 26.75s, the separation charge ignites, giving a slight boost to the velocity of the payload as it is ejected from the main body. Though the rocket still has a non-zero upward velocity, the separation effectively declared this point to be apogee, since the rocket will not continue upwards when the nosecone is absent and the chutes deploy. (The rocket was not in fact going 25 mph at separation as the graph shows. This discrepancy is due to accumulative integration errors).

4. Chute Deployment

   As the chutes deploy from the rear of the module, payload velocity decreases to zero and then becomes negative as the payload begins to descend. There is a sudden impulse as the nosecone, which houses the telemetry package, is stopped by the chutes and reverses direction.

5. Descent

   The chutes quickly restrict the payload to its terminal velocity.

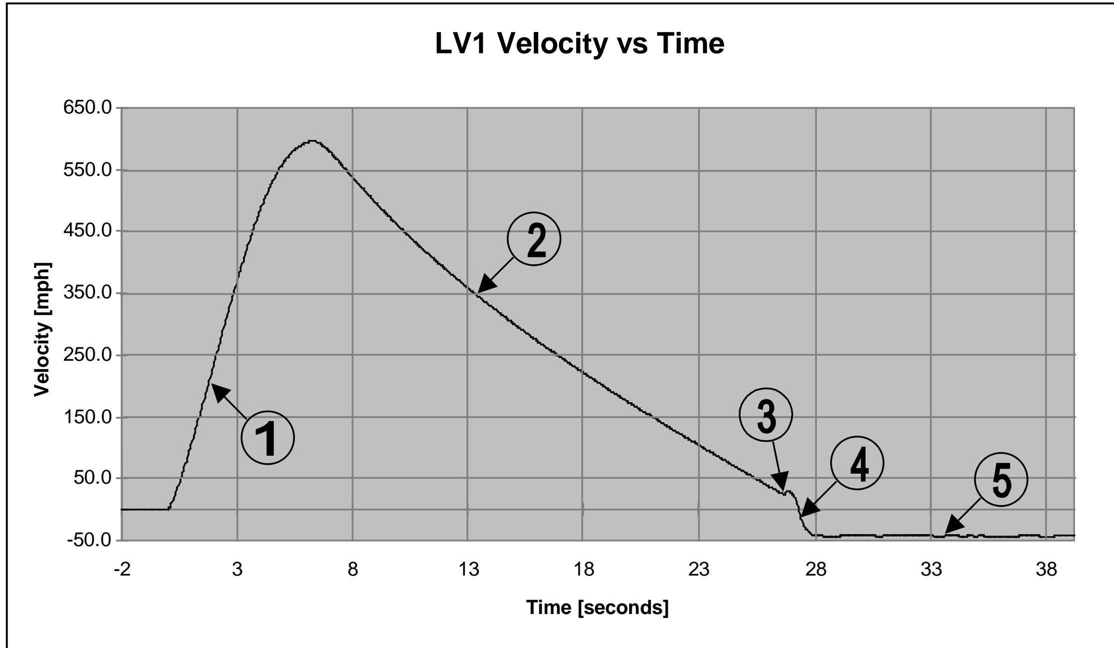


Figure 9: Velocity profile for LV1

Integrating the acceleration data a second time produces a graph of the position. The accumulated integration errors, however, severely undermine the validity of the graph, as shown in Figure 10 below.

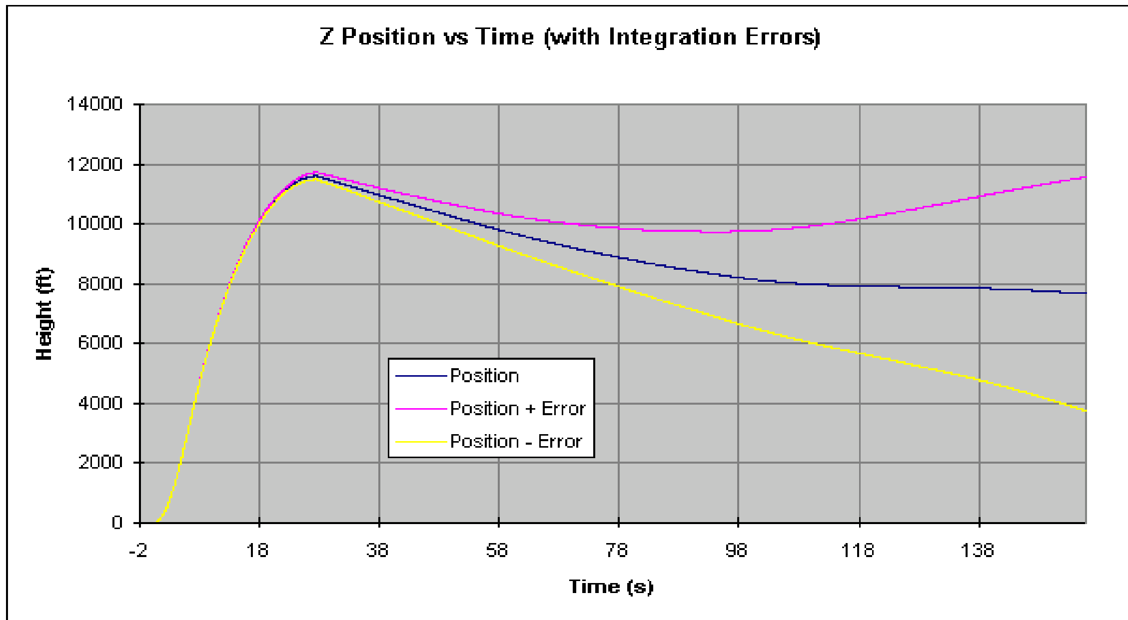


Figure 10: Position profile for LV1

To more accurately determine the height of the rocket, the pressure sensor data was analyzed as shown in Figure 11 below. This data lent itself to a much more meaningful interpretation.

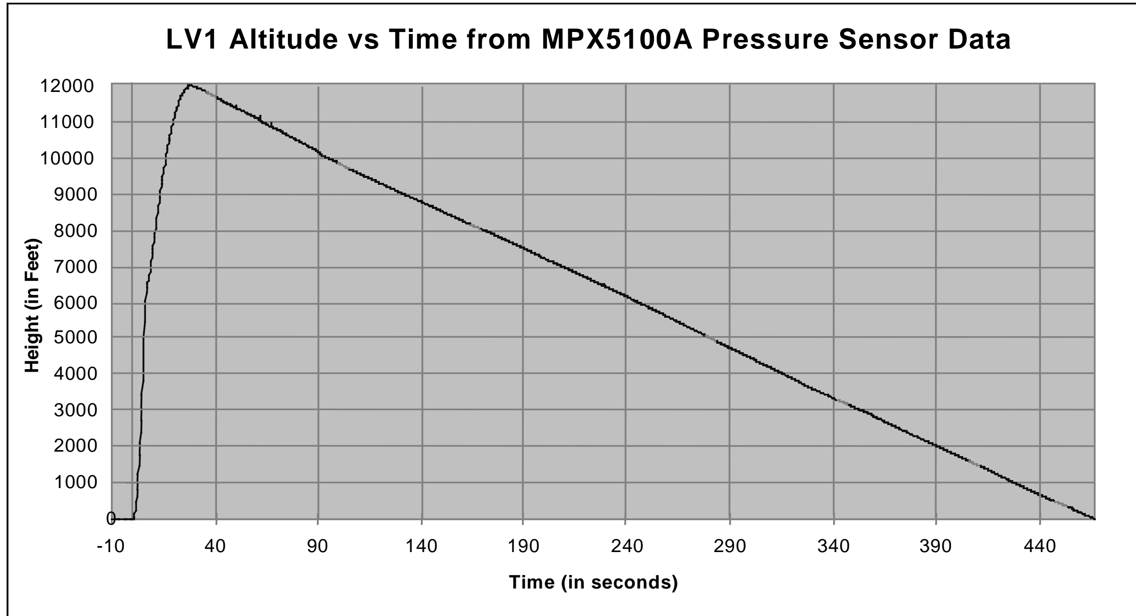


Figure 11: Pressure sensor altitude profile for LV1

As shown in the previous graphs, it may be necessary to combine a number of sources of data to determine the 'correct' telemetry. For an actively guided system, it will be necessary to combine data from the accelerometers and pressure sensors as well as possible Global Positioning System (GPS) data and radio packet time-of-flight indicators. Other possibilities include a sun sensor mounted in the nosecone and high-performance sensors such as ring laser gyros.

With all telemetry measurements it will be necessary to account for signal latency. Time-of-flight, GPS satellite linkup, and even A/D-conversion delays become significant when the rocket is moving at speeds nearing 1 Mach, since by the time a reading has been processed a large distance will have been covered by the rocket. This will necessitate a successive-approximation approach wherein the avionics package will use its current position and heading data to predict where it will be when the next reading occurs; the next reading can then be used to update the approximation and minimize positional error.

Launch Vehicle Two (LV2)

To achieve orbit (or very high altitudes) it is necessary to integrate an Inertial Measurement Unit (IMU) with an active guidance system to create an Inertial Navigation System (INS). Prior PSAS rockets have taken steps to develop an IMU as a proof-of-concept for the INS.

The development of such an INS will require modular independent subsystems which can be individually upgraded and evolved without compromising the stability, precision, and safety of the overall system. It is toward this end that we designed the following architecture for Launch Vehicle Two (LV2).

Having observed both the present lack of advanced systems and the future need for those systems, it became apparent that an intermediate step was required. The development of a system with modular components would allow for the systematic qualification of subsystems needed for future launch capabilities. Extending the modularity idea to the airframe as well as the avionics will ensure that the entire system is scalable.

Of particular importance is the validation and testing of the IMU subsystem. The IMU sensor suite and the data streaming algorithms used to transmit and store that data need to be qualified for use with an INS system. Data from this system will be used to generate flight profiles for flight sequencing tests and flight path calculation in future rocket projects.

The initial development of LV2 embodies a clarification of the direction and goals of the PSAS. Rather than being an interim design on the path to an advanced system, LV2 will serve as a continuously upgradable modular design that allows the testing and validation of a wide range of avionics, payload, and airframe ideas.

The airframe design for LV2 will result in a modular launch vehicle which can be assembled into any configuration required for a flight. Fins, motor modules, and payload sections can be added, swapped, or removed, allowing for the testing and validation of various flight configurations. This modular approach will provide the PSAS with an evolvable launch vehicle which will serve as a testbed for avionics and propulsion design for years to come.

It should also be noted that for LV2 the concept of ‘payload’ has been redefined to a more traditional interpretation: a payload is an independent package carried by the rocket. The term was previously used to describe the flight control systems of the rocket, which will henceforth be known as ‘avionics’. Payload development is outside of the scope of this document.

Development of the avionics package began with a functional decomposition of the overall system, which resulted in a number of functionally discrete system task blocks that must be accomplished in a successful flight. In keeping with our goal that the system be as modular as possible, the avionics team chose to embody each of these blocks as a separate subsystem. This will ensure that replacement of a subsystem will not affect other subsystems that are functionally distant. If each module has a clearly defined interface, individual subsystems can be swapped for upgrade or repair without affecting the overall functionality of the avionics system.

Based on the results of LV0 and LV1, the PSAS met to discuss the requirements for the next launch vehicle and to clarify the focus and goals for the future. This discussion led to the development of a list of requirements that the avionics package must embody subject to a list of constraints. These requirements and constraints were given to the avionics team to guide the design of the new avionics package. In the next section of this document, we will define the problem at hand and discuss the motivating factors that influence the design.

Figure 12: Possible Configuration of LV2

Problem Statement

To design, implement, and validate an avionics module consisting of a flight sequence computer, an inertial measurement unit, a solid-state data recorder, a data acquisition package, and a communications system for the PSU Aeronautics Society's amateur launch vehicle LV2. This avionics module will serve as a scalable development and qualification testbed which can be expanded to accommodate a future active guidance system.

Constraints

The payload system design is subject to the following constraints:

- All systems must use the 'PIC' microcontroller architecture from *Microchip Technology Inc.* to match existing AESS systems.
- The system must minimize weight.
- The system must minimize power consumption and operate on 12VDC to match existing AESS systems.
- The system is subject to space requirements imposed by the design of the existing LV2 avionics bay.
- For forward design mobility, modular boards with robust quick disconnects should be used.
- The system must be able to withstand 20g's acceleration.
- Critical systems should include redundant backups: 2 software backups, and one physical backup; reliability is *very* important.
- The system should be inexpensive, using off-the-shelf components and (where possible) industry samples.
- The system should be reproducible, using common parts and construction techniques.
- Modularity is of the utmost importance. There must be a consistent logical and physical interface to the system.

Design Requirements

The following requirements were developed by the avionics design team in conjunction with PSAS team members. The payload system must contain the following elements and meet the goals listed below:

Overall

The entire system must be modular and evolvable to meet the functional requirements for many future launches.

Sensors:

Sensor are functionally divided into two categories, proprioceptive and external. Proprioceptive ('self-perceiving') sensors monitor the status of the rocket, while external sensors monitor environmental conditions.

Proprioceptive sensors

Dedicated sensors must include:

- Igniter validation sensors to determine if the igniters are open, shorted, or valid. This information is relayed to the flight computer and the Downlink for pre-launch systems validation.
- Launch detect monitor to determine the moment of separation from the launchpad. This information is relayed to data storage as well as the Downlink for accurate ground support flight sequencing and flight-path determination.
- Separation monitor to determine the moment of separation of the Payload module from the main body during recovery.
- Power status sensors to monitor the on-board power supply.

Subsystem status sensors must include:

- Flight state monitors which will gather data from the on-board subsystems and the radio Uplink.
- Subsystem status checks to update the states in the flight computer.

External sensors

External sensors must include:

- Inertial measurement sensors: accelerometers to determine linear motion in 3 axes and gyros to determine rotational motion in 3 axes.

Data acquisition sensors must include:

- Temperature and pressure sensors
- Strain gauges to determine the physical dynamics of the rocket in flight

Imaging sensors must include:

- Video, to be broadcast to ground support.

Communications:

Uplink

Manual recovery system which must include the ability to interface with the flight computer for manual status checks, arming of the system, and emergency control of the payload separation system.

Downlink

Telemetry data and video must be sent back to ground support.

Communications subsystem

This subsystem will handle the routing of all air-to-ground and inter-subsystem communications.

Data storage:

- '*Black box*' for raw data storage
- Raw data from the telemetry sensors and the flight computer will be stored for later recovery and analysis.
- Telemetry sent to base station on ground
- Telemetry data and flight computer status must be sent to the ground support for monitoring and storage.

Flight Computer:

The flight computer must control flight sequencing based on internal state machines, input from other subsystems, and uplink data.

Architectural considerations

To reach the goals outlined above, we examined a number of different possible architectures. A summary of the possibilities is shown below.

Central/Monolithic Architecture

The current (LV1) rocket uses a monolithic payload system in which all tasks are performed by one main computer. The limitations inherent in this architecture, a block-diagram of which is shown in Figure 13 below, were the primary source of motivation for the design of a new system.

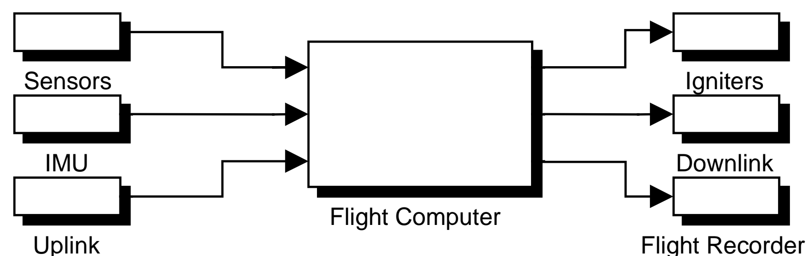


Figure 13: Monolithic architecture block diagram

Pros:

- Simple: easy to implement.
- Consolidated: all resources are part of the primary computer, so no complex communications protocols need to be developed.
- Lightweight: minimal component count keeps weight down.
- Small: this system minimizes the amount of space taken up within the payload structure.

Cons:

- Non-scaleable: future additions to the payload would place unrealistic demands on system resources due to limited pin count.
- Too localized: a failure in non-critical subsystems could lock up the primary flight computer.
- Overloaded: with all tasks being performed by one central computer, the processor that is in charge of flight sequencing must spend most of its clock cycles shuffling data.
- Requires a specific Flight Computer architecture to meet the hardware port requirements.

This architecture, while adequate for the first PSAS systems, is rapidly approaching saturation as existing subsystems are upgraded and additional subsystems are added. To adequately meet the needs of future systems, a different architecture is required.

Serial Architecture (direct)

Since all candidate microcontrollers have an onboard serial port, one option for the system architecture was a serial bus. We considered two types of serial configurations: a ring architecture, shown in Figure 14, and an arbitrated architecture, shown in the next section.

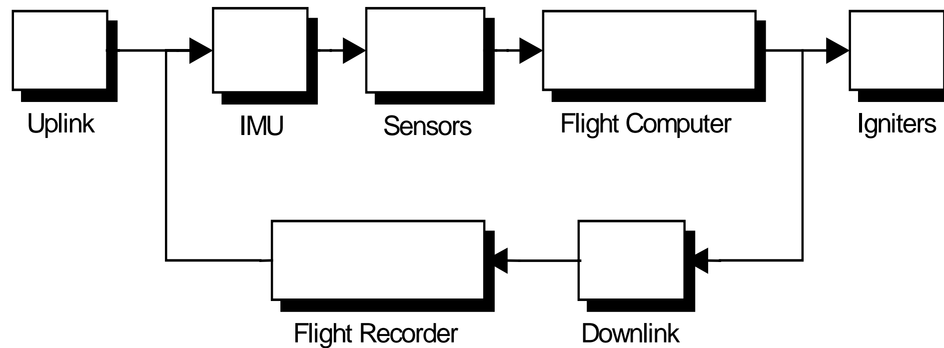


Figure 14: Serial ring architecture block diagram

The ring serial architecture provides all subsystems with a communications path to all other subsystems, but it requires each subsystem to pass along a substantial amount of data that is intended for a different target.

Pros:

- Simple: very basic structure.
- Ease of implementation: uses only on-chip serial communications hardware (SPI, USRT, and UART protocols).
- Expandable: this architecture can be 'daisy-chained' indefinitely as new subsystems are added.

Cons:

- Bandwidth: each subsystem must be capable of passing large amounts of data, most of which is not needed by any one system, or
- Ownership: each subsystem must have knowledge of what data is needed further down the chain.
- Fragile: this architecture is not fault-tolerant. If one subsystem goes offline, they all do.

Serial Architecture (arbitrated)

The arbitrated serial architecture uses a separate subsystem (the ‘arbiter’) to grant access to a single bi-directional serial bus as each subsystem needs it. When a subsystem desires access to the bus, it asserts its REQ (request) line to announce that it is requesting the bus. When the bus is free, the arbiter (using some fairness algorithm to allocate the bus where it is needed without locking out the low bandwidth subsystems) asserts the GNT (grant) line of the subsystem, allowing the subsystem to take control of the bus. Various timeout protocols and restrictions on the length of time any one subsystem can control the bus would be implemented in an attempt to ensure that all subsystems are able to transport their data. Figure 15 shows a detail of the system connections.

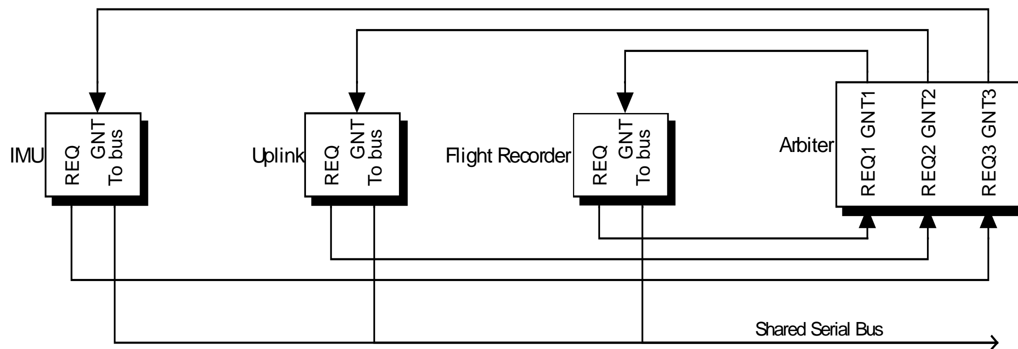


Figure 15: Detail of arbitrated serial bus

Pros:

- Orderly: no device contention due to the arbiter
- Scaleable: can be expanded indefinitely as new subsystems are added.
- Fault-tolerant: if one system goes offline, the other systems are not materially affected (though the lack of information from the offline system might affect other subsystems).

Cons:

- Implementation: requires arbitration hardware in addition to built-in serial communications hardware. As the number and complexity of payload subsystems grows, the arbiter would require additional resources and the bus would become increasingly overloaded.
- Bandwidth: one serial bus is responsible for all of the system’s data flow, thereby creating a bottleneck when more than one system needs to move large amounts of data.

For either of these two implementations of the serial architecture, the ‘cons’ far outweigh the ‘pros’. Since all of the negative aspects of this system are in direct conflict with project constraints and requirements, we discarded this architecture as a possible candidate.

Distributed Architecture

To reduce the amount of communications bandwidth required by any one subsystem, we next considered a distributed architecture, as shown in Figure 16. This architecture provides maximal interconnection between subsystems, thereby reducing the load on any single bus.

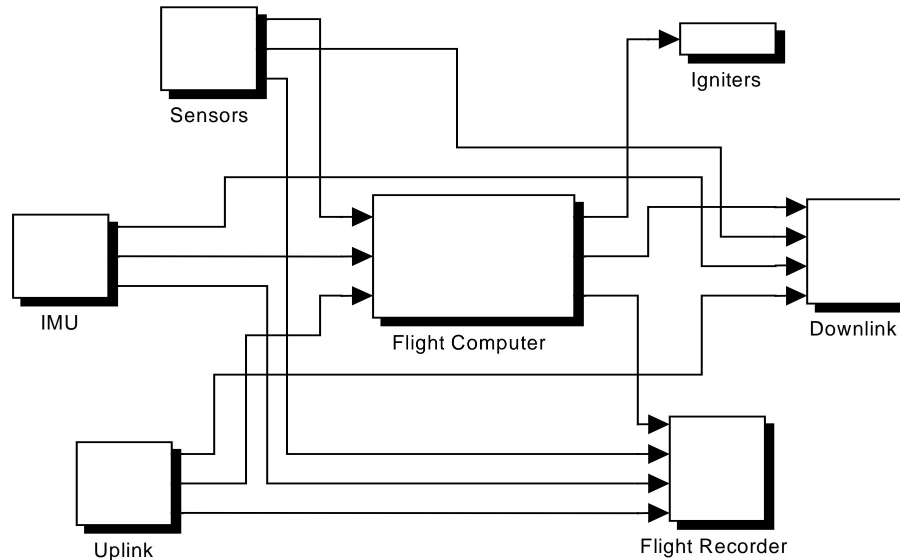


Figure 16: Distributed architecture block diagram

Pros:

- Bandwidth: each bus has significantly reduced bandwidth; subsystems are only presented with the data they need.
- Fault-tolerant: a failure in any one subsystem will only affect systems that rely on the data from that subsystem.

Cons:

- Increased hardware and software complexity: each subsystem has an increased need for communications ports to accommodate the distributed structure of the system.
- Requires a specific Flight Computer architecture to meet the hardware port requirements.

The increase in complexity that this architecture provides is fundamentally opposed to the desire to make the system modular and scaleable. If any one system needs to be replaced, numerous communications protocols must be followed to ensure proper connections to the other systems. This places additional requirements on the communications hardware that must be supported by each chip. Based on this limitation, we chose not to consider this distributed model as a candidate architecture.

Modular Architecture

In attempt to find some optimal middle ground, we took desired features from the previously-discussed architectures and combined them into a modular architecture, as shown in Figure 17.

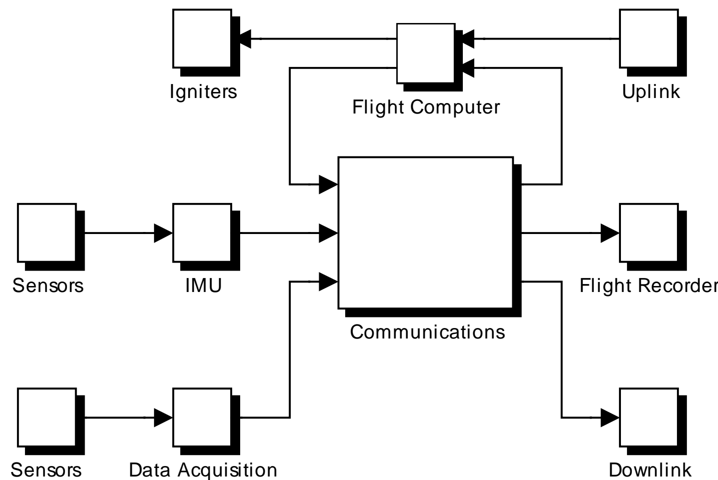


Figure 17: Modular architecture block diagram

Pros:

- Minimizes the complexity of the communications systems within modules that are not primarily communications oriented.
- Modules can be upgraded or interchanged with other modules that have the same interconnect specifications.

Cons:

- Communications computer must be able to handle significant bandwidth through a large number of connections.
- Hardware port requirements dictate Flight Computer architecture.

In keeping with the idea that individual modules should be isolated according to function, we considered the addition of a subsystem whose sole job would be to route communications between the other modules. This Communications Computer would be the only chip in the system with complex communications needs; all other subsystems could focus on their tasks without the additional burden of routing data packets.

The primary drawback of this architecture is that all system functionality is dependent on the proper operation of the CC. While it is still possible to revert to backup mechanisms (such as internal timers) to accomplish the flight sequencing, subsystems that are operational will be effectively disabled by a CC failure. We therefore chose to consider other architectures for our implementation.

Serial Architecture with Hardware Arbitration

Recent advances in serial bus design have resulted in the creation of hardware-arbitrated buses that provide a modular and robust method for component interconnection. One specification for this type of network is the Controller Area Network (CAN or CANbus). Originally developed for the automotive industry, the CANbus serial protocol was designed to provide reliable real-time data connections in non-ideal industrial environments.

CAN provides many features which will be useful in the context of avionics systems. Hardware arbitration, automatic retransmission of messages when an error occurs, and message prioritization will simplify the task of developing a robust fault-tolerant network.

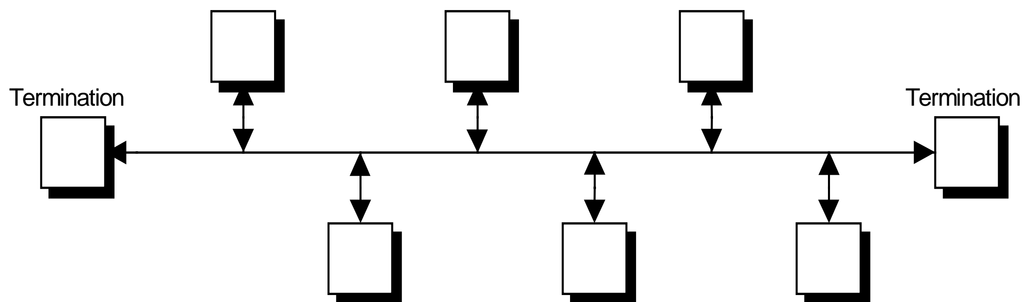


Figure 18: CANbus architecture block diagram

Pros:

- Emphasizes modularity. Any subsystem can be added to the overall design by adding bus interface hardware to the subsystem.
- Emphasizes reliability. Many fault recovery and prevention features are built into the hardware specification.
- Consistent hardware interface for all modules.
- Provides a convenient mechanism for prioritizing data packets.

Cons:

- Limited bandwidth. Maximum bus speed is 1Mbps.

The CANbus architecture allows for the creation of a scalable, modular network which provides significant hardware-based mechanisms for ensuring the reliable transmission of data. As with the arbitrated serial bus discussed previously, there is a reduction in available bandwidth when using this architecture; nevertheless the benefits afforded by this implementation greatly outweigh the drawbacks.

Implementation

In the final analysis, we chose to use the hardware-arbitrated serial CANbus architecture (discussed in the previous section) as the foundation for our avionics system design. This architecture efficiently embodies the motivating characteristics required for the target system, providing us with a very scalable and modular implementation that will support hardware revisions and upgrades throughout the development cycle of the next few upcoming launch vehicles.

Use of the CANbus allows us to embody each of the functionally separate task blocks (listed in the Design Requirements) as physically distinct devices capable of communicating with all other devices on the network. This dramatically simplifies the task of upgrading and replacing avionics components, since each subsystem can be replaced with any device that follows that subsystem's communications protocol. The functional isolation also lends itself to a modular development style in which each subsystem can be independently qualified and tested before it is interfaced to the rest of the network.

In this section we will provide an introduction to the CANbus, starting with an overview of the CAN specification, some discussion on arbitration and prioritization techniques, and a description of the method in which CAN was implemented in our design. We will then discuss each of the subsystems in the avionics package, describing their hardware, software, and messaging details.

Finally, in the 'Operations' section we will discuss how the individual subsystems are united to form a coherent unified avionics package, and the 'Ground Support' section will describe some of our ideas for linking our airborne design to the computers on the ground.

CANbus Overview

CAN (Controller Area Network) is an ISO approved standard for a low-cost, fault tolerant, and robust real-time communication protocol. Initially developed by Robert Bosch for in-vehicle data transfer in 1984, silicon became available in 1987 and CAN was first used in cars in 1992. The draft international standard was introduced in 1991 and this became a full standard (ISO 11898) in 1994.

The serial bus structure gives two advantages over parallel bus systems: increased transfer reliability even over large distances and more favorable costs. The CAN specifications define a multi-master priority based bus access which uses carrier sense multiple access with collision detection and non-destructive arbitration. (CSMA/CD + NDA). This hardware-based arbitration scheme provides system wide data consistency.

CAN allows multicast reception with time synchronization error detection and error signaling. Corrupted messages are automatically retransmitted and defective nodes are automatically removed from the circuit. Nodes can make remote data requests. The signaling uses non-return to zero (NRZ) bit encoding and allows full isolation of the interconnecting wires.

The CAN implementation of the arbitrated serial bus architecture is being used in other aeronautics projects as well: the Alpha Magnetic Spectrometer (AMS), tested aboard the space shuttle Discovery in June of 1998 and ultimately destined for the international space station, uses the CAN architecture for internal communications between subsystems.

How does CAN work?

Information transmitted on the CANbus is organized into small packets called "Frames". A frame consists of some data-addressing information (a message Identifier which also determines the priority of the message), up to eight bytes of data, and some error checking. When a frame is transmitted, each receiving node will acknowledge that the frame has been correctly received by inserting an acknowledge bit into a space left in the frame by the transmitting node, which can then determine that at least one node has correctly received the frame.

The CANbus protocol is data-addressed rather than node-addressed. The content of the message (e.g. Acceleration data, Igniter checks, Subsystem status, etc.) is labeled by an identifier that is unique throughout the network. All other nodes on the network receive the message and each performs an acceptance test on the identifier to determine if the message, and thus its content, is relevant to that particular node. The "Identifier" part of the CANbus frame is used to both identify a message and indicate the message’s priority.

Bus arbitration is handled through CSMA/CD with NDA (Carrier Sense Multiple Access / Collision Detect with Non-Destructive Arbitration). When a node is ready to transmit a message across the network, it must first verify that the bus is in the idle state (Carrier Sense). If the bus is idle, the node becomes the bus master and transmits its message. All other nodes detect the beginning of a message frame, switch to receive mode, and send an acknowledge after correct reception of the message.

If more than one node starts to transmit at the same time (Multiple Access), message collision is avoided by a bit-wise arbitration scheme. As each transmitting node sends the bits of its message identifier, it monitors the bus level. A node that sends a recessive bit but detects a dominant one loses bus arbitration and switches into receive mode (Collision Detect/Non-Destructive Arbitration). This results in a prioritization scheme in which higher priority messages are given an Identifier with higher order dominant bits. Nodes whose messages are subsumed by higher-priority messages will attempt to resend the message when the bus becomes idle.

Figure 19 demonstrates the arbitration process by which a node determines that it has lost bus-master status. The top signal shows the output from the node in question, while the bottom signal shows the actual signal level present on the bus. (Signal sense is electrically active-low, not logical; a 'low' level represents a dominant bit). When the ninth identifier bit (ID.2) is transmitted, the transmitting node observes that the signal level on the bus is dominant, while the output from the node is recessive. Logically, this represents a case where the node in question wishes to transmit a message with priority '11000100000' (logical) while a remote node on the bus wishes to send a message with priority '11000100100'. The transmitting node therefore loses arbitration since the remote node wishes to transmit a message with a higher priority. When the transmitting node detects that it has lost bus-master status (when it receives ID.2, in this case), it switches to receive mode.

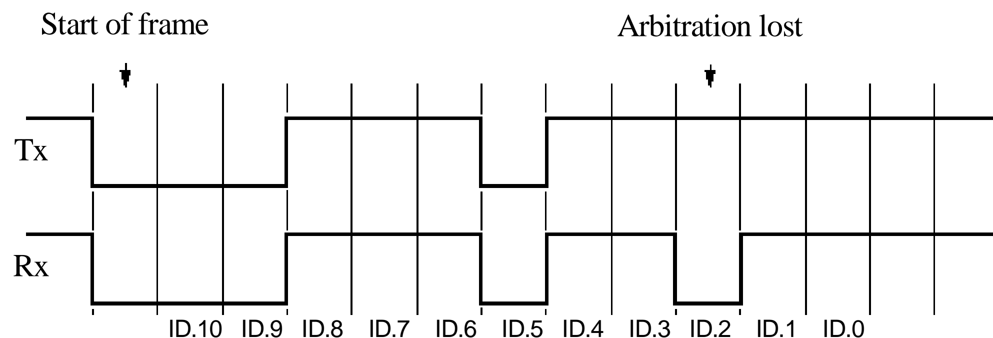


Figure 19: Example of arbitration/prioritization scheme

Identifiers

As mentioned above, the unique identifier also determines the priority of the message. The lower the (electrical) numerical value of the identifier, the higher the priority of the message.

The higher priority message is guaranteed to gain bus access as if it were the only message being transmitted. Lower priority messages are automatically re-transmitted in the next bus cycle, or in a subsequent bus cycle if there are still other, higher priority messages waiting to be sent.

Addressing and Arbitration

While an arbitrary 11-bit message identifier allows for up to $2^{11} = 2048$ unique messages, the CANbus implementation allows for only 2032. The 16 messages that correspond to the case where the 7 MSB of the message identifier are recessive are not allowed due to the fact that the Dataframe 'end-of-frame' delimiter consists of 7 consecutive recessive bits.

There are 2 bus electrical states: dominant, which represents '0' or 'actively driven', and recessive, which represent '1' or 'inactive'. The bus must be a "wired-or": in other words, if any node transmits a dominant bit, then the bus will be in the dominant state. All nodes must transmit a recessive bit for the bus to be recessive.

Each node monitors the bus, and does not start a transmission if a frame is being transmitted by another node. If two nodes commence transmission at exactly the same time, the identifiers (which are transmitted first) will overlap. As the protocol prohibits two nodes transmitting the same identifier, the output state of the two should differ at some time during the identifier. In this case, the node transmitting the recessive bit will see a dominant bit on the bus and cease transmission. Since the MSB of the identifier is transmitted first, the lowest binary value of identifier always "wins", hence has the higher priority.

Error-Checking

CANbus uses self-checking, bit stuffing, an "in-frame" acknowledge and a CRC to check that messages are correctly transmitted and received. An incorrect message will be flagged by all nodes on the bus, and a re-transmission will be automatically triggered. The probability of a received message being erroneous is very low.

The CAN protocol has five methods of error checking, three at the message level and two at the bit level. If a message fails any one of these error detection methods it will not be accepted and an error frame will be generated which will

cause all other nodes to ignore the defective message and the transmitting node to resend the message.

At the message level, a cyclical-redundancy check (CRC), an Acknowledge field within the message, and a form check are used to detect errors. The 15-bit CRC is carried out over the Identifier field and the Data bytes, and then compared to the CRC filed within the message. If the values differ, an error is generated. The Acknowledge field is two bits long and consists of an acknowledge bit and an acknowledge delimiter bit. The transmitter will place a recessive bit in the acknowledge field. Any node that receives the message correctly will write a dominant bit in the acknowledge field. If the transmitter does not detect a dominant bit in the acknowledge field, it will generate an error frame and retransmit the message. The third message-level error check is a form check which verifies that certain fields in the message contain only recessive bits. If a dominant bit is detected, an error is generated. The bits checked are the start-of-frame, end-of-frame, Acknowledge delimiter and the CRC delimiter bits.

At the bit level, each message is monitored by the transmitter. If a bit is written onto the bus and its compliment is read, an error is generated. The exceptions to this rule are the Identifier field, which uses the bus contention in its non-destructive arbitration (NDA) scheme, and the acknowledge slot, which requires a recessive bit to be overwritten by a dominant bit when a message is correctly received by a remote node.

An additional error detection method uses the bit-stuffing rule, wherein a node that transmits a message that contains five consecutive bits of the same logic level must then transmit a single bit of the complimentary level to prevent the presence of a net DC level on the bus. If the next bit is not a compliment after five consecutive bits of the same logic level, then an error is generated. Bit-stuffing is used for bits between the start-of-frame and the CRC delimiter.

Realization

The CANbus is a multi-layer protocol. As implemented here, there are three primary layers:

- Application Layer
- Protocol Layer
- Physical Layer

The Application Layer is the device that needs to speak on the CANbus. In our design, each subsystem that requires access to the bus contains a PIC microcontroller, which will view the entire CANbus as simple Memory-Mapped IO (MMIO).

The Protocol Layer maintains coherency and manages all CAN-related activities. Duties include:

- Message filtering and validation
- Fault confinement
- Error detection and signaling
- Arbitration
- Message framing
- Transfer rate and timing management

Typically, the Protocol Layer is implemented using an off-the-shelf CAN controller chip. The CAN controller used in the avionics package is the Microchip MCP2510. This is a stand-alone controller with the following features:

- Implements Full CAN specification,
- Supports standard and extended frames,
- Programmable bit rate of up to 1Mbps,
- Remote frame support,
- Two receive buffers,
- Six full acceptance filters
- Two full acceptance filter masks
- High Speed SPI Interface
- Temperature range from –40C to +85C

The SPI interface allows for a simple high-speed connection between the CAN controller and the PIC.

The Physical Layer defines how signals are actually transmitted, providing the signal-level representation of the data. This layer connects the CAN controller to the transmission medium, which is often a shielded cable or twisted-pair to reduce electro-magnetic interference (EMI). The Physical Layer used in the avionics package in the Philips PCA82C250 transceiver. This device is capable of 1Mbps data transfers using a differential-pair implementation that minimizes EMI and provides hardware protection from power surges on the CANbus. In addition, current-limiting devices protect the transmitter output from damage in the event of short circuit to Vcc or ground. The PCA82C250 is rated from –40C to +125C.

Using the parts mentioned above, we developed a consistent CAN interface that will provide connectivity to any device that has a SPI communications port. The circuit schematic for this interface is shown in Appendix A.

Connection to the CAN interface is a 7 wire interconnect that provides the essential 4-wire SPI functionality as well as 3 additional signals:

- CS# (active-low chip select)
- RESET# (active-low CAN controller reset)
- STANDBY (active-high CAN transceiver reset)

These supplemental pins can be used to place the entire system into low-quiescent-current standby mode. When in this mode, all devices on the bus (with the exception of one device, whose job is to wake everyone else up) are essentially asleep. The appearance of a message on the CANbus causes the transceiver to pulse its connection to the controller. This in turn causes the controller to wake, whereupon it triggers an interrupt in the application layer device.

Standby mode can be used to minimize power consumption and maximize safety by placing the entire avionics package in a harmless deactivated state. The entire system can then be woken up with a transmitted command from ground support.

This mode is not required; if a specific subsystem does not wish to use this mode, the lines can be left unconnected; the CAN interface circuit contains pull-ups and pull-downs to ensure that the pins revert to the correct default states when they are unused.

Module Overview

The final avionics design embodies a functional decomposition approach by creating a physical module to represent each functional division. In addition, it is possible to further decompose each of the modules along functional lines (e.g. by dividing each module into a sensor section and an actuator section). The avionics modules are listed below:

- Flight Computer (FC)
- Communication Computer (CC)
- Inertial Measurement Unit (IMU)
- Data Acquisition board (DAQ)
- Flight Recorder (FR)
- Igniters (IGN)
- Test Module (TM)
- PSAS systems

Each of the modules listed above is a self-contained independent device whose sole connection to the avionics package is via the CANbus. As a result, subsystems can be upgraded or replaced without requiring any modifications of the other subsystems or the avionics package as a whole.

The FC uses data from the other subsystems to determine flight sequencing. Once a flight stage has been detected, the FC is responsible for taking any necessary action, such as triggering stage separation at apogee.

The CC provides a data connection between the avionics system and the outside world. In addition to handling all communications with ground support via the Uplink and Downlink, the CC also provides communications to any third-party scientific instrumentation devices which might be present in the payload.

The IMU has sensors to measure linear acceleration and angular velocity; data from these is used to calculate the position, orientation, and velocity of the rocket.

The DAQ provides a view of conditions both internal and external to the rocket. The proprioceptive sensors measure flight critical systems such as stage-separation sensors and battery levels. The external sensors measure quantities such as temperature and pressure.

The FR is a solid-state data recording device. It stores all information sent to it in nonvolatile Flash RAM memory. The data stored inside the FR will be used for post-flight analysis.

The igniters are smart, self-contained devices connected to the CANbus. Each IGN is capable of testing its igniter pair, relaying status data to the rest of the

system, and triggering ignition in response to a command or an internal timer preset.

The TM provides a mechanism by which false data can be inserted into the avionics system for testing and flight profiling. It also allows the external monitoring of the sensors for sensor characterization and validation. This module is beyond the scope of this project, and will be developed in the future.

The PSAS is responsible for following systems

- Communications package, consisting of an uplink and a downlink
- Ground support

The communications package will consist of a bi-directional high-speed serial connection to the ground. The uplink allows the ground station to communicate with the rocket, providing the ability to manually command the avionics to perform tasks such as start on-board diagnostics, go to a known state, or activate the separation igniters. The downlink is a digital channel that is used to transmit data from the rocket to the ground station. The channel includes IMU data, DAQ data, and FC state information.

Ground support systems include a ‘smart’ launch tower with shore power, a ground control station with flight control/monitoring computers, and radio systems to maintain communications.

Flight Computer (FC)

The Flight Computer (FC) is the 'brain' of the system, responsible for all actions taken by the avionics system. This subsystem is responsible for the sequencing of flight states based on internal state machines and inputs from other system modules.

This modules main tasks include:

- Flight sequencing
- Launch initiation
- Stage separation

Hardware

The microcontroller used in the FC is a PIC 16F873. The choice to use this particular microcontroller was determined by the following criteria:

- The 16F873 is the smallest PIC that still provides the SPI communication hardware required by the CAN interface.
- In-circuit programming and debugging capabilities
- reduced pin count
- reduced package size
- reduced weight

The 16F873 is a 28-pin device that has the following features:

- Fosc = 20 MHz clock speed (Instruction speed Fcyc = Fosc / 4)
- 4K x 14-bit program word Flash memory
- 192 byte data memory
- 128 byte EEPROM data memory
- 1 high-speed USART communications port
- 1 high-speed Serial Port (SPI)
- watch-dog timer (WDT)
- 3 internal timers

This device has advanced in-circuit programming and in-circuit debugging features that significantly reduce program development and implementation times. Additional useful features include an energy-saving sleep mode and a small instruction set. The 16F873 is rated from –55C to +125C.

The FC is connected to the CANbus via the standard CAN interface discussed in the CAN section of this document.

Complete schematics for the FC can be found in Appendix A.

An internal timer system allows the FC to keep track of where the rocket is in the flight, as well as providing a backup mechanism for initiating critical events in case of a subsystem failure. If the IMU experienced a failure, for instance, and was unable to report apogee detection, the timers in the FC would automatically trigger payload separation at a pre-calculated time.

Timers can also be used to control sensors which might not be needed until a certain time in the flight, such as a scientific instrumentation package that is only concerned with readings near apogee.

The internal timers in the FC consist of 8-bit free-running counters with a frequency of 5.0MHz. These counters roll over every 6.55mS. Each time the counter rolls over, an Interrupt Service Routine (ISR) is run that increments a number of different counter registers. These registers are used as the internal event timers to determine time since liftoff, apogee timeout, and sensor activation times.

Messaging

The FC is designed to accept all valid messages that are critical to determining flight sequencing. These include all EVENT and STATUS messages from all subsystems:

- IMU.EVENT
  IMU-originated message showing launch/apogee detection.
- IMU.STAT
  IMU-originated message showing internal status of IMU.
- DAQ.EVENT
  DAQ-originated message showing launch/apogee/separation detection.
- DAQ.STAT
  DAQ-originated message showing internal status of DAQ.
- FR.EVENT
  FR-originated message showing status of Flash RAM.
- FR.STAT
  FR-originated message showing internal status of FR.
- IGN.ICHK.n
  IGN-originated message showing condition of igniter #n
- IGN.EVENT.n
  IGN-originated response to arming/disarming of igniter #n.
- IGN.STAT.n
  IGN-originated message showing internal status of the igniter #n. In addition to providing information on igniter status and whether or not the igniter is armed or in diagnostic mode, the STAT message also lets the FC know that the CANbus connection to the igniter is still valid.
- CC.STAT
  CC-originated message showing internal status of CC. CC.STAT also returns a byte showing the last known state of the FC; if the FC wakes from a watch-dog timer (WDT) reset, it will adopt this state as its own.

In addition to these messages, the FC must respond to all manual override commands from the CC and track (if possible) the command decisions sent through the CC. These commands include:

- CC.REQ.POWU
- CC.CMD.POWD
  CC-originated messages requesting power-up/down of all systems.
- CC.REQ.ACK
  CC-originated message acknowledging valid FC boot-up configuration.

- CC.CMD.ARML
- CC.CMD.DISL
  CC-originated messages requesting arming or disarming of launch igniter.
- CC.REQ.LAU
  CC-originated message requesting ignition of launch igniter.
- CC.CMD.SCHK
  CC-originated message requesting status from all modules.
- CC.CMD.ARMn
- CC.CMD.DISn
  CC-originated message commanding arming or disarming of igniter #n.
- CC.CMD.IGNn
  CC-originated message commanding triggering of igniter #n.
- CC.CMD.FCON
- CC.CMD.FCOFF
  CC-originated messages enabling/disabling FC status output.
- CC.CMD.FKS
  CC-originated message forcing FC into known state.
- CC.CMD.DIAG
  CC-originated message commanding all systems to enter diagnostic mode.
- CC.CMD.NORM
  CC-originated message commanding all subsystems to enter normal mode.

Note that some of the messages received from the CC are requests rather than commands. This ensures that certain critical events are never initiated unless the FC is operational.

The FC generates the following messages:

- FC.CMD.POWU (Power up)
- FC.CMD.POWD (Power down)
  Commands all subsystems to power up/down, to preserve power and to force the system into a known initialized state.
- FC.CMD.ARML (Arm launch igniter)
- FC.CMD.DISL (Disarm launch igniter)
  Arms/disarms the launch igniter. This is accomplished by sending a launch code which the launch igniter must have in order to respond to an ignition command. These launch codes can be erased by a disarm command at any time prior to actual ignition, thereby preventing the ignition.

- FC.CMD.LAU (Launch the rocket)
  Triggers the launch igniter.
- FC.CMD.ICHK (Check igniters (masked))
  Instructs igniter modules to check the validity of their igniters. A byte-mask is included to allow the targeting of a specific igniter or set of igniters. Those igniter modules that are covered by the byte-mask will respond by sending an IGN.ICHK.n message.
- FC.CMD.SCHK (Check subsystem status (masked))
  Instructs all modules to return a status check. A byte-mask is included to allow the targeting of a specific module or set of modules. Those modules that are covered by the byte-mask will respond by sending a STAT message. Note: modules will respond to an FC.STAT command (discussed below) in the same manner as they would upon receiving an FC.CMD.SCHK with full byte-mask.
- FC.CMD.ARMn (Arm igniter #n)
- FC.CMD.DISn (Disarm igniter #n)
  Arms/disarms the auxiliary igniter #n. This is accomplished by sending a launch code which the igniter must have in order to respond to an ignition command. These launch codes can be erased by a disarm command at any time prior to actual ignition, thereby preventing the ignition.
- FC.CMD.IGNn (Trigger igniter #n)
  Triggers igniter #n.
- FC.CMD.FREN (Enable FR recording)
- FC.CMD.FRDIS (Disable FR recording)
  Enables/disables FR data recording. This functionality is included to prevent the recording of spurious data when launch delays occur and to provide a mechanism for initiating the recording process when the rocket is on the launchpad.
- FC.CMD.IMUON (Enable IMU output)
- FC.CMD.IMUOFF (Disable IMU output)
  Enables/disables output from the IMU. The IMU will still sample data, but that data will not be placed on the bus. Allows for diagnostic modes and introduction of test signals as well as reducing bus bandwidth and FR storage requirements while on the launchpad.
- FC.CMD.DAQON (Enable DAQ output)
- FC.CMD.DAQOFF (Disable DAQ output)
  Enables/disables output from the DAQ. The DAQ will still sample data, but that data will not be placed on the bus. Allows for diagnostic modes and introduction of test signals as well as reducing bus bandwidth and FR storage requirements while on the launchpad.

- FC.STAT (FC status frame)

  Informs the system of the status of the FC. This message is sent out regularly at a frequency of 100Hz, corresponding to the frequency of the master control loop within the FC. Allows for fault recovery, since systems that do not hear from the FC for a specified amount of time can assume that either the FC is damaged or their CANbus connection to the FC is severed, and can therefore transfer control to their internal timers. All modules on the CANbus are required to accept this message. This message is the 'heartbeat' or synchronizing pulse of the system, and contains the primary system timestamp. Upon receipt of this message, all modules on the CANbus must return their own STAT messages, thereby resulting in a continuous time-stamp-referenced data block in both the FR and the Downlink stream that contains the status of the entire system. FC status bytes include a time-stamp, the current flight state identifier (discussed below), and the status of the system. If other modules on the CANbus wake from a WDT reset, they can utilize the system status information to resynchronize their internal states with the FC.

A summary of all system messages and commands is given in Appendix B.

Software

The basic functionality of the FC comes from an internal state machine that uses data from other subsystems and an internal timer to trigger state flow. The state machine is responsible for all flight sequencing including launch initiation, sensor activation, and stage separation. All command decisions in normal operation are made by this subsystem in response to messages from other subsystems and internal timers. In the event of an emergency, the FC state flow can be subsumed by manual commands via the CC.

The FC accepts STAT messages from all subsystems on the CANbus. The status bytes from these messages are parsed and used to create a system-wide status indicator that is included in the FC.STAT message.

The FC state machine is at all times in one of five possible modes :

- Initialization
  At power-up, a number of tests are performed to ensure that the entire system booted correctly and all communications pathways are operating correctly.
- Pre-flight
  Prior to launching, additional tests are performed and the primary launch igniter is given the authorization codes (without which it is not possible to launch the rocket). This is the idle state for the rocket; deviations from the ideal state flow will result in a return to this mode.
- Launch
  Launch mode covers the time from the moment the launch command is given to the moment primary ignition occurs. This mode, which roughly corresponds to the '10-second countdown', is initiated by a CC.REQ.LAU request from the CC. In response, the FC performs igniter checks and status checks on all subsystems, starts a 10-second countdown (which is transmitted to ground support), and enables data output from all sensors. If any tests fail, the FC will disarm the primary launch igniter and revert to pre-flight mode; otherwise, the FC will initiate the primary launch igniter at time t = 0s.
- Flight
  Flight mode covers the time from the actual launch of the rocket to the time when the avionics system lands on the ground. All actual flight sequencing takes place in this mode, including sensor management, payload management, and stage separation.

- Recovery

  When the system lands after a successful flight, all sensor data (with the possible exception of GPS data) will be silenced. The FC will continue to broadcast FC.STAT to provide a beacon for the recovery team.

These modes are discussed in more detail in the flight sequencing section of this document.

Critical FC tasks utilize a number of backup mechanisms. The most catastrophic failure that can occur in the course of a normal single-stage flight is the lack of separation or untimely separation of the stages from the rocket body. If separation occurs while the rocket is accelerating, the chutes will be destroyed and the resulting collision (between the rocket sections or between the rocket and the ground) will most likely result in the complete loss of the system. The flight computer uses a number of independent indicators to determine the suitable time to deploy the recovery system. This is accomplished by the following mechanisms:

- Z-axis position begins to decrease. If the IMU detects a negative Z-axis velocity for 10 consecutive samples, the rocket is near apogee and the IMU will send an EVENT notification. Resolution of the IMU sensors is approximately 10ft.
- Pressure begins to increase. If the DAQ detects an increase in pressure over 10 consecutive samples, the rocket has passed apogee and the DAQ will send an EVENT notification. The pressure sensor outputs an analog voltage proportional to the pressure, with a resolution of approximately 15ft.
- Manual intervention via the Uplink. If the ground crew determines that the rocket has reached apogee (via visual observation using theodolites), a manual separation code (CC.CMD.IGNn) can be sent to the rocket to force separation.
- Internal timer. If the timer onboard the FC determines that apogee *must* have been reached (based on a comparison of the time since liftoff to a pre-calculated value), the FC will deploy the chutes. This is a final fail-safe that will only be used if all other avenues have failed.

In addition to the above-mentioned mechanisms, it is possible to force the FC into any specific state via the Uplink.

Communication Computer (CC)

The Communications Computer (CC) is responsible for all communications with the world beyond the avionics system. This module must route all data between the avionics package, the communications system, and the payload. This module's main tasks include:

- Collect and package relevant data for distribution to the payload and ground communications.
- Route commands and requests from ground support to the CAN bus.

Hardware

The microcontroller used in the CC is a PIC 17C756. This high-end microcontroller was chosen since it provides all of the communications hardware required by the CC. The 17C756 is a 64-pin device that has the following features:

- F_{osc} = 33 MHz clock speed (Instruction speed $F_{cyc} = F_{osc} / 4$)
- 16384 x 16-bit program word memory
- 902 byte data memory
- 50 digital I/O ports
- 2 high-speed universal synchronous/asynchronous (USART) communications ports
- 1 high-speed serial peripheral interface (SPI) communications port
- watch-dog timer (WDT)
- brown-out reset
- 4 internal timers

Communications to and from the ground station come from an off the shelf RF modem channeled through a serial port on the PIC. To facilitate the use of a modem supplied by the PSAS, the interface between the modem and the CC consists of a standard RS232 connection. The signal level shifting (between RS232 and digital levels) is handled by a MAX 3221 transceiver.

Control and information passing to the payload is supported primarily through another UART port on the CC. The exact data, commands and communication protocol sent via this port are customized according to the needs of the payload. In addition, one 8-bit parallel port is available as needed.

The CC is connected to the CANbus via the standard CAN interface discussed in the CAN section of this document.

Complete schematics for the CC can be found in Appendix A.

Messaging

The CC must be able to monitor all inter-module messaging to perform its role of communicating with the ground station and payload. The CC accepts and processes all messages from all subsystems. Additional details can be found in the Software sections.

Messages originating from the CC are primarily the result of commands sent from the ground. They include:

- CC.REQ.POWU
  A request from the CC to activate the FC.
- CC.CMD.POWD
  Requests power-up/down of all systems.
- CC.REQ.ACK
  Acknowledges valid FC boot-up configuration.
- CC.CMD.ARML
- CC.CMD.DISL
  Requests that the launch igniter be armed or disarmed.
- CC.REQ.LAU
  Request ignition of launch igniter.
- CC.CMD.SCHK
  Requests status from all modules.
- CC.CMD.ARMn
- CC.CMD.DISn
  Commands that igniter #n be armed or disarmed.
- CC.CMD.IGNn
  Commands the triggering of igniter #n.
- CC.CMD.FCON
- CC.CMD.FCOFF
  Enables/disables FC status output.
- CC.CMD.FRON
- CC.CMD.FROFF
  Enables/disables the FR status output.
- CC.CMD.FREN
- CC.CMD.FRDIS
  Enables/disables the recording of data by the FR.
- CC.CMD.FRERA
  Commands the FR to erase the contents of the Flash, and reinitialize the device.

- CC.CMD.IMUON
- CC.CMD.IMUOFF
  Enable/disable IMU status and data output.
- CC.CMD.FKS
  Forces the FC into known state.
- CC.CMD.DIAG
  Commands all systems to enter diagnostic mode.
- CC.CMD.NORM
  Commands all subsystems to enter normal mode.
- CC.STAT
  Status message that shows the current state of the CC. It also includes information for the FC in order to recover from a WDT.

Note that some of the messages sent by the CC are requests rather than commands. This ensures that certain critical events are never initiated unless the FC is operational.

Software

The functionality of the CC is broken down into providing communication to three areas: the ground station, CANbus devices, and the payload. The CC must parse messages from all sources to determine the appropriate routing measures and to generate the necessary CAN messages.

All CAN messages are monitored to determine which of those messages are appropriate for packaging into frames which can be sent to the ground station via the serial downlink. All command decisions will be sent as well as a selected sampling of the data generated by the IMU and the DAQ.

Commands originating from the ground via the uplink are parsed for content and the appropriate CAN messages are generated. This allows ground support to intervene manually in emergency situations. The CC can generate messages having the same functionality (but a higher priority) as the corresponding FC messages.

The CC provides the payload with data gathered from a sampling of the data generated by the IMU and DAQ. This data is sampled and forwarded to the payload to aid it in its operation. Pertinent data includes current state of the avionics system, current rocket position, flight time information, and the current flight state.

Inertial Measurement Unit (IMU)

The Inertial Measurement Unit (IMU) is the guiding force behind many of the engineering decisions made in the avionics package. The modularity requirements that dictate an easily-implementable upgrade/development path to a functional INS were a major determining factor in the use of the CANbus in the avionics package architecture.

This module's main tasks include:

- Measure raw linear acceleration (X,Y, and Z axes)
- Measure raw angular acceleration (α, β, and γ)
- Calculate acceleration
- Calculate velocity (integrate acceleration)
- Calculate position (integrate velocity)
- Calculate orientation
- Detect apogee events
- Transfer all data to the FR and CC

Hardware

The microcontroller used in the IMU is a PIC 17C756. In addition to providing us with the processing capabilities we need for this generation of payload, the 17C756 will allow for scalability in the future.

The primary goal of the IMU is to monitor the linear acceleration and angular velocity sensors to determine as precisely as possible the position, orientation, and heading of the rocket. The sensor package designed to facilitate this consists of three sections:

- 3 linear accelerometers
- 3 angular velocity sensors
- 1 multi-channel A/D converter

These sections are described below:

Linear accelerometers

The linear accelerometers are micro-machined monolithic components with on-chip filtering and a selectable ±25g or ±50g full-scale range (Analog Devices, Part Number #ADXL150AQC/ADXL250AQC). The ADXL150 is a single axis accelerometer; the ADXL250 contains two independent accelerometers physically arranged in quadrature on the plane of the chip. The 'AQC' suffixed parts have a temperature rating from –40 °C to +85 °C.

The payload module will contain one ADXL150 for Z-axis monitoring and one ADXL250 for X- and Y-axis monitoring. The accelerometers will be configured for ±25g full-scale operation to increase the resolution.

Subjecting the accelerometer to an acceleration causes a small change in the capacitance between a fixed plate and a force-sensitive plate within the device. This change in capacitance is then used to determine an appropriate change in output voltage. The output voltage varies by approximately ±38mV / g.

A diagram of the internal structure is shown below in Figure 20.

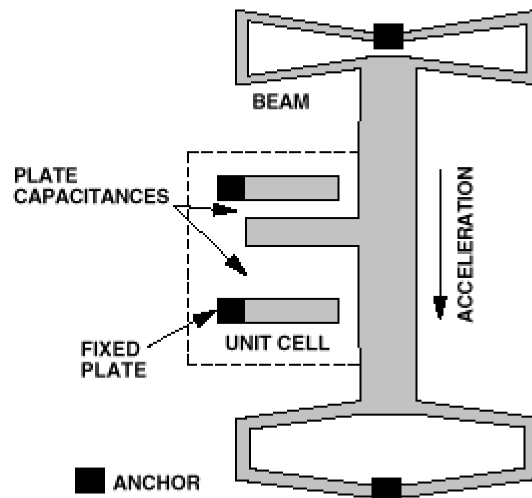


Figure 20: Internal structure of linear accelerometers

The accelerometers are each sampled at approximately 1.5kHz and the resulting data is passed through an A/D converter before entering the IMU for processing and distribution.

Angular velocity sensors

The angular sensors used in the payload are ceramic angular rate gyros (Tokin, Part Number #CG-16D0). The sensors output voltage is proportional to the rate of spin, with a sensitivity of 1.1mV/degree/S. The CG-16D0 is characterized for operation from –5 °C to +75 °C. The frequency-phase characteristics of this device (shown below in Figure 21) limit the maximum sampling rate to approximately 150Hz.

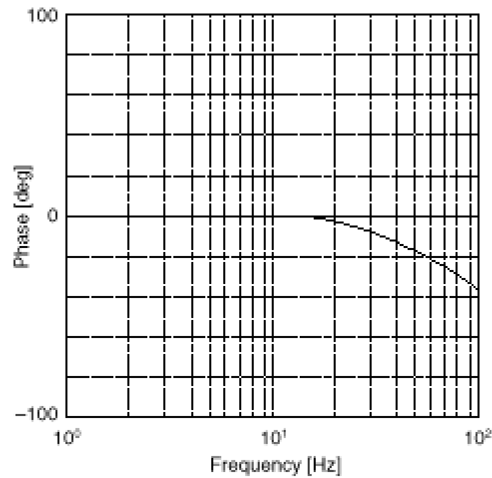


Figure 21: Frequency-Phase characteristic of Tokin angular rate gyro

A/D Converter

The A/D conversion is handled by an 8-input, 12-bit, switched-capacitor, successive-approximation analog-to-digital converter (Maxim, Part Number #MAX197). The 8 input channels are multiplexed (MUXed) to a single A/D converter and are selectable via serial input line from the IMU. The device returns a 12-bit digital value via a separate serial line. The MAX197 is characterized for operation from 0 °C to +70 °C.

The IMU-A/D connection consists of 13 wires:

- CS# Chip select, active low
- WR# Write signal from IMU
- RD# Read signal from IMU
- HBEN High-bit enable
- INT# 'End of conversion' signal from A/D
- D0-D11 12-bit (multiplexed) Data bus

The A/D converter utilizes a standard chip select signal, with WR# and RD# controlling writes and reads respectively.

The HBEN pin selects whether the least significant byte or the most significant nibble is output on the data bus.

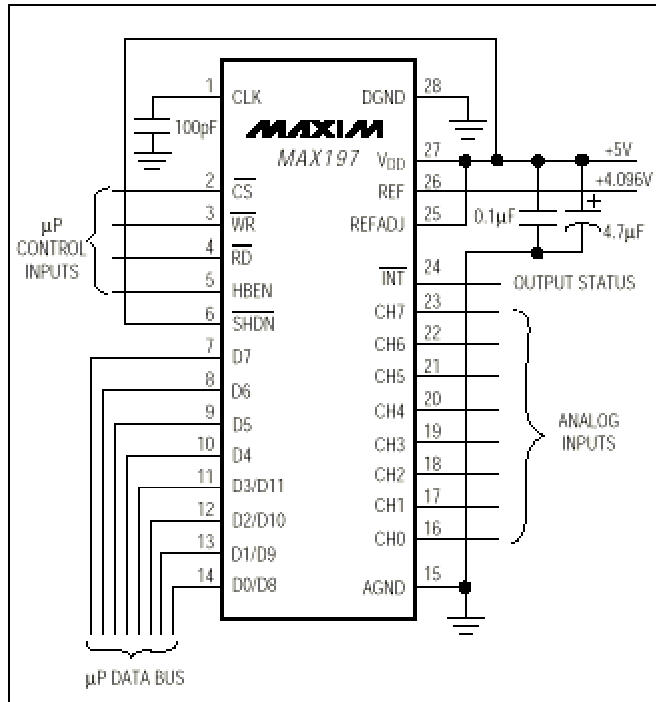


Figure 22: Connecting to the A/D converter

Conversions are accomplished as follows:

- The PIC selects a MUX channel by placing the channel code on the data bus and writing (asserting WR# and CS#).
- The ADC samples the sensor connected to the selected MUX channel (the conversion phase typically lasts 4µS).
- The ADC asserts INT# to alert the PIC that the operation is complete.
- With HBEN deasserted, the PIC latches in the 8 LSB's of the reading from D0-D7.
- With HBEN asserted, the PIC latches in the 4 MSB's of the reading from D8-D11.

The IMU is connected to the CANbus via the standard CAN interface discussed in the CAN section of this document.

Complete schematics for the IMU can be found in Appendix A.

Messaging

The IMU accepts only those messages that directly affect its own operation:

- FC.CMD.POWU
- xC.CMD.POWD
  FC- or CC-originated messages requesting power-up/down of all systems.
- xC.CMD.SCHK
  FC- or CC-originated message requesting status from all modules.
- FC.STAT
  FC-originated message showing internal status of FC as well as status bits for the rest of the system.
- CC.CMD.IMUON
- CC.CMD.IMUOFF
  CC-originated messages enabling/disabling IMU data output.
- CC.CMD.DIAG
  CC-originated message forcing all systems into diagnostic mode.
- CC.CMD.NORM
  CC-originated message forcing all systems into normal mode.

The IMU generates the following messages:

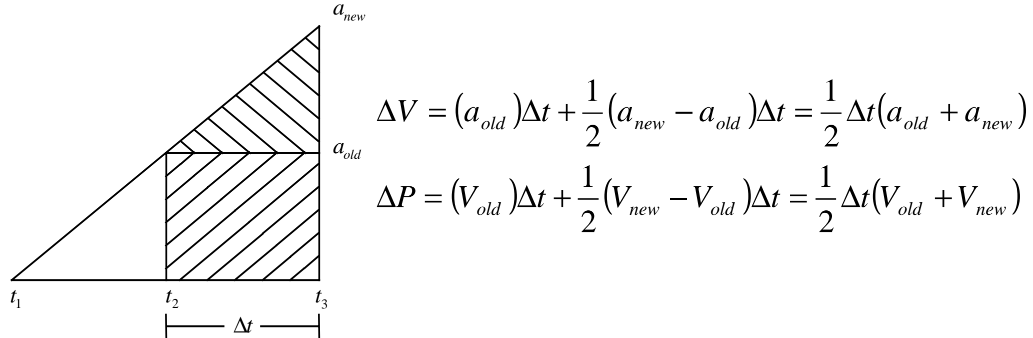
- IMU.XYZ (X,Y, & Z-axis accelerometer data, raw)
- IMU.ABC (α, β, & γ-axis gyrocompass data, raw)
  Raw data packages from all inertial navigation sensors. This data will be stored in the FR and broadcast to ground support via the CC. Each message contains 3 12-bit values corresponding to the X,Y, & Z or α, β, & γ readings.
- IMU.ACC.n (n-axis acceleration data, n = {X,Y,Z})
- IMU.VEL.n (n-axis velocity data, n = {X,Y,Z})
- IMU.POS.n (n-axis position data, n = {X,Y,Z})
- IMU.ORI.n (n-axis orientation data, n = {α,β,γ})
  Acceleration, velocity, position, and orientation data. This data has been highly processed by algorithms in the IMU.
- IMU.EVENT (Event flags)
  Event flag message generated when the IMU determines that apogee has been reached.
- IMU.STAT (IMU status frame)
  Status message that shows the current state of the IMU (i.e. enabled/disabled, diagnostic/normal mode).

A summary of all system messages and commands is given in Appendix B.

Software

The IMU will use the raw sensor data to calculate the absolute position (relative to the launchpad) of the rocket. This will involve numerous calculations; an overview of the operation is shown below.

Linear acceleration data is integrated to obtain linear velocity using a midpoint-rule approximation as shown below in Figure 23. A second integration is then performed to calculate the position.



$$\Delta V = (a_{old})\Delta t + \frac{1}{2}(a_{new} - a_{old})\Delta t = \frac{1}{2}\Delta t(a_{old} + a_{new})$$

$$\Delta P = (V_{old})\Delta t + \frac{1}{2}(V_{new} - V_{old})\Delta t = \frac{1}{2}\Delta t(V_{old} + V_{new})$$

Figure 23: Integration approximation in the IMU

Further algorithms will be developed in the future to incorporate the angular measurements into the position calculation.

The IMU will also send an EVENT message when it detects apogee. This occurs when the IMU detects a negative Z-axis velocity for 10 consecutive samples, implying that the rockets upward velocity has decreased to zero and is beginning to become negative. The EVENT messages will be used by the FC to determine flight sequencing.

When the IMU is operational, it will constantly sample the sensors regardless of whether or not data output has been enabled by the FC or the CC. The sampling rate will be substantially higher than the rate at which data is output from the IMU; this will allow for higher resolution input to an internal Kalman filter which will preprocess the data before transmitting on the CANbus.

In future implementations, the IMU will be part of a larger Inertial Navigation System (INS). This INS will be capable of actively guiding the rocket along a pre-established flight profile and delivering a payload to a specific target. The INS will also be an experiment in Control Engineering, allowing the removal of the stabilizing fins from the rocket; this will in effect create a high-speed ‘inverted pendulum’ model which must be actively stabilized throughout the flight.

Data Acquisition Module (DAQ)

The DAQ is the main subsystem by which the rocket avionics system gathers information about the physical state in and around the rocket. This includes the sensing of both proprioceptive and external conditions.

Proprioceptive measurements gather information on conditions internal to the rocket, while the external measurements gather information on conditions external to the rocket. Proprioceptive sensors include sensors to measure and detect:

- power system status,
- module separation, and
- shore power disconnect.

External sensors monitor:

- temperature, and
- air pressure.

Hardware

The microcontroller used in the DAQ is a PIC 17C756. This device is compact, but still provides many ports for monitoring multiple digital signals, and interfacing with external circuitry such as A/D converters. The primary connections to the DAQ are described below.

A/D Converter

The A/D converter, a MAX197 from Maxim, is identical to that used in the IMU. All analog signals will be converted through this device. The internal A/D on the 17C756 will not be used.

Battery level

The health of the power system will need to be monitored. This is particularly important while the rocket waits on the launch pad prior to pre-flight and launching. The power system consists of off the shelf 12V Lithium batteries and a series of linear regulators and bypass capacitors. (The power system will be provided by the PSAS).

The test circuit for the power system will employ a simple voltage divider that drops the 12-volt supply down to the 5 volts range of the A/D converter. See Figure 24 below for the interface circuit. The interface will access the 12 volt power from the system bus.

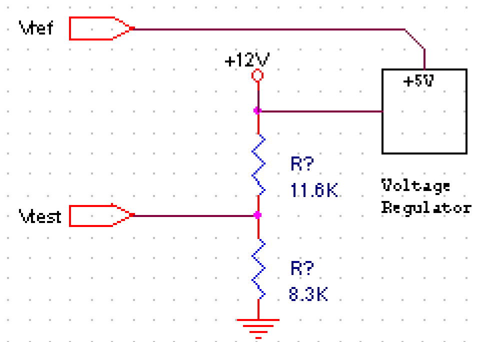


Figure 24: Power Monitoring Circuit

Separation monitors

The separation monitors verify that rocket modules have physically separated. An example of this would be the payload and body modules separating at apogee. This measurement is required to determine the need for a manual intervention in the case of a FC failure.

The sensor used is a slotted optical switch, (QT Optoelectronics, Part Number #QVB11123) that returns a logic high value until separation has occurred. The DAQ will poll the sensor to determine its state. Figure 25, below, shows the switch and interface circuitry.

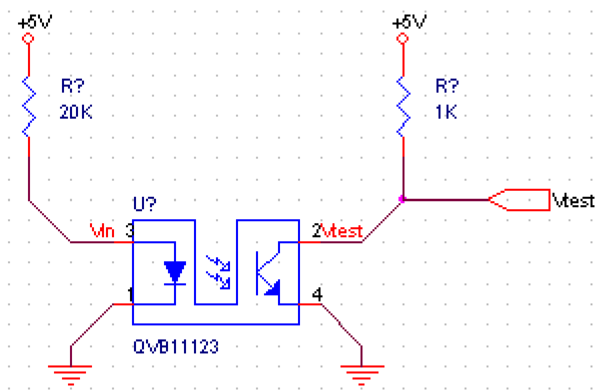


Figure 25: Separation Sensor Schematic

Temperature

The temperature data will be gathered from various points in the rocket. This will be accomplished via a 3-terminal adjustable current source (National Semiconductor, Part Number #LM234). This device produces a current source proportional to the temperature when wired as shown in Figure 26 below. The LM234 features a 10,000:1 range in operating

current, excellent current regulation and a wide dynamic voltage range of 1V to 40V.

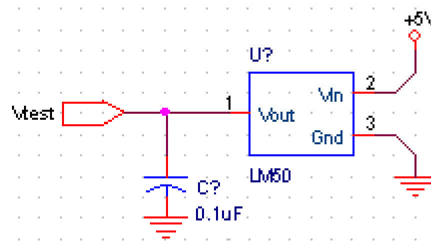


Figure 26: Temperature Sensor Schematic

The sense voltage used to establish operating current in the LM234 is 64 mV at 25°C and is directly proportional to absolute temperature (°K). The LM234 is specified as a true temperature sensor with guaranteed initial accuracy of ±3°C. In addition, the LM234 is guaranteed over a temperature range of -25°C to +100°C. The subcircuit shown above in Figure 26 produces a voltage change of 10mV/°K.

Pressure

The pressure sensor used is an integrated silicon pressure sensor with on-chip signal conditioning, temperature compensation, and calibration (Motorola, Part Number #MPX5100A). Device characteristics include:

- on-chip signal conditioning,
- on-chip temperature compensation,
- absolute pressure measurement,
- 15 to 115 kPa range,
- maximum error of +/- 2.5 kPa,
- 1m Sec response time,
- temperature operating range from –40° to 125° C,
- output voltage range from 0.2 to 4.7 V,
- consumes 10mA max.

Interfacing this device to an A/D requires a simple R-C circuit to provide low pass filtering, as shown in Figure 27 on the next page.

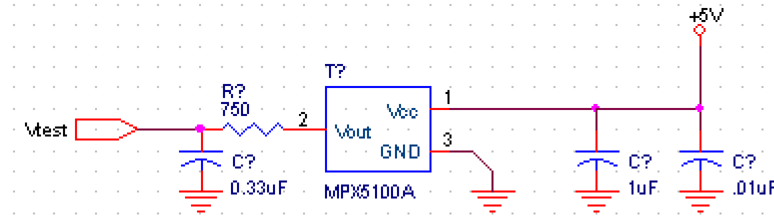


Figure 27: Pressure Sensor Schematic

Upon sampling the sensors, the DAQ processes the signals as needed and outputs the data to the CC via a high-speed serial interface. The DAQ also has an input channel that provides a means of enabling and disabling sensors and/or third party systems during the flight.

The DAQ is connected to the CANbus via the standard CAN interface discussed in the CAN section of this document.

Complete schematics for the DAQ can be found in Appendix A.

Messaging

The DAQ accepts only those CAN bus messages that pertain to its functionality. Accepted message are:

- FC.STAT
- xC.CMD.SCHK

  These commands signal the need for status information, and cause the DAQ to generate a DAQ.STAT message. FC.STAT acts as a trigger for all subsystems to send their status messages.
- FC.CMD.POWU
- xC.CMD.POWD

  These commands cause the FR to go into and out of the low power sleep mode.
- xC.CMD.DAQON
- xC.CMD.DAQOFF

  Enable/disables output of data from the DAQ. Events are still generated in response to critical flight stages, but DAQ.TEMP and DAQ.PRESS readings are not sent over the CAN bus.
- CC.CMD.DIAG
- CC.CMD.NORM

  CC-originated message commanding the FR to enter or leave diagnostic mode.

To capture in flight data, and help in flight sequencing, the DAQ has the capability to generate data and event messages.

- DAQ.TEMP (Temperature data)

  The temperature readings around the rocket are included in this data message.
- DAQ.PRESS (Pressure data)

  Pressure data from the pressure sensor.
- DAQ.EVENT (Apogee/separation detect)

  This event message marks the occurrence of apogee, stage separation or a WDT reset by the DAQ. The apogee detect is based on the pressure sensor and is used by the FC to aid in flight sequencing. Stage separation is used for flight sequence verification.
- DAQ.STAT (DAQ status frame)

  The DAQ.STAT message is generated when the DAQ receives a FC.STAT, CC.CMD.SCHK, or FC.CMD.SCHK message. It contains information that indicates the internal state and status of the DAQ, the voltage of the avionics power system, and the condition of the systems it monitors.

Software

The DAQ software has several tasks to perform. It must sample and send all environmental and proprioceptive sensor data over the CAN bus, monitor the pressure data for apogee detect, and monitor and act on commands from the FC and CC.

The flow chart for the DAQ is shown below:

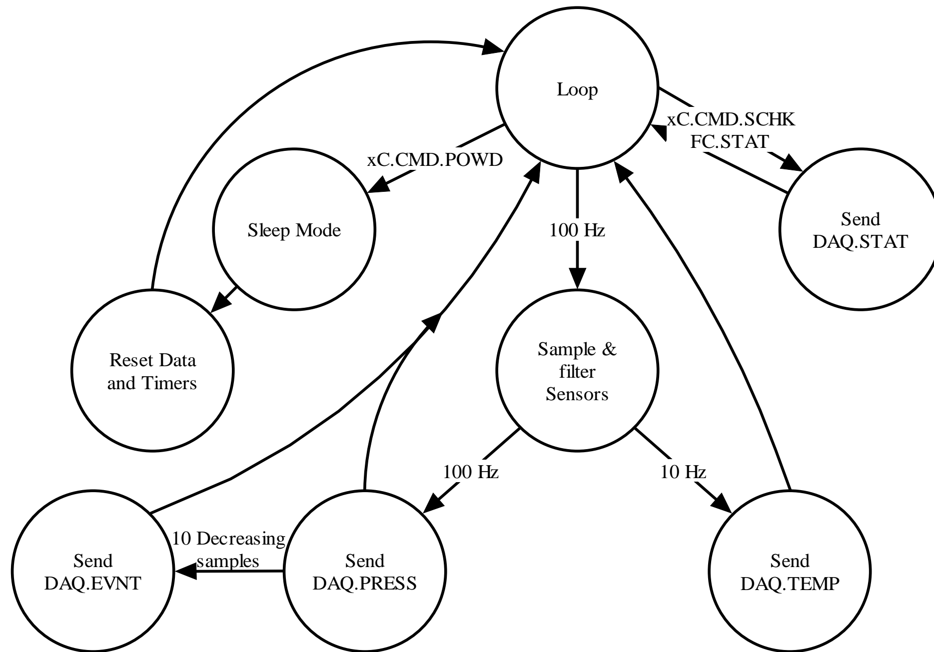


Figure 28: DAQ Flowchart

The main loop for the DAQ involves sampling the sensors at 100 Hz, performing any digital filtering on the readings, and sending messages. The pressure data message (DAQ.PRESS) is sent at a frequency of 100 Hz, the temperature data (DAQ.TEMP) is sent at 10Hz, and the battery level is sent as part of the DAQ status message (DAQ.STAT). The flow diagram also illustrates the DAQ's response to the commands xC.CMD.POWD and xC.CMD.SCHK.

Flight Recorder (FR)

The Flight Recorder (FR) is designed to function as a "Black Box," recording any and all data it sees. It is composed of three parts: the processor, the CAN bus interface, and the Flash bank. It functions by capturing all messages from the CAN bus, processing them as necessary, and writing them to the bank Flash memory. Retrieving the FR data is done on the ground as a post-flight activity.

Hardware

The microcontroller used in the FR is a PIC 17C756. The choice to use this particular microcontroller was determined by the following criteria:

- The 17C756 has 50 I/O pins and allowed us to interface with the Flash bank without any support logic, providing a single chip solution,
- remain in the '17C' family, thereby allowing us to use a command set common the other avionics subsystems,
- hardware based SPI and USARTs are available for communications,
- reduced package size,
- reduced weight, and
- the capabilities of the chip still exceed performance requirements for this module.

The Flash RAM selected is the AMD Am29F032B with the following characteristics:

- CMOS 5.0 V single power supply operation,
- Access times as fast as 70ns,
- reduced package size,
- reduced weight,
- Flexible sector architecture,
- Minimum 1,000,000 write/erase cycles, and
- Ready/Busy output signal.

The FR is connected to the CANbus via the standard CAN interface discussed in the CAN section of this document.

Complete schematics for the FR can be found in Appendix A.

Messaging

To perform the function of recording all messages on the bus, the FR is designed to accept and record all valid messages.

In addition to recording all messages, the FR must also act on command messages specific to it. Command messages can come from the CC or the FC and include:

- xC.CMD.POWU (Power up)
- xC.CMD.POWD (Power down)
  Wake up from, or go into power saving sleep mode. This command can come from ground control (CC.CMD.POWx) or from the FC (FC.CMD.POWx).
- FC.STAT (Message containing FC status)
- xC.CMD.SCHK (Check subsystem status)
  These commands signal the need for status information, and cause the FR to generate a FR.STAT message. FC.STAT acts as a trigger for all subsystems to send their status messages.
- CC.CMD.DIAG (Enter diagnostics mode)
- CC.CMD.NORM (Enter normal mode)
  CC-originated message commanding the FR to enter or leave diagnostic mode.
- xC.CMD.FREN (Enable FR recording)
- xC.CMD.FRDIS (Disable FR recording)
  This command pair originates from CC and is used to control when the FR records CAN bus message traffic.
- CC.CMD.FRERA (Erase FR Flash memory)
  Commands the FR to erase the Flash memory. After the erase procedure, all registers are reset and the FR restarts as if just powered up. The Flash erase is a privileged command and can only be initiated by ground control via the uplink and CC. The FC cannot initiate a FRERA.

The FR must generate the following messages:

- FR.STAT (FR status frame)
  The FR.STAT message is generated either in response to a specific status check command from the FC or CC (xx.CMD.SCHK), or after a FC.STAT is sent. The FC.STAT message is used to synchronize and trigger status messages from all modules. It is a 4 byte message with 1 byte for state information, and 3 bytes for the

amount of Flash memory used. Listed below is the state information contained in the FR.STAT message:

    - Recording on/off,
    - FIFO is more than ½ full,
    - Flash is full, and
    - FIFO is full.

- FR.EVENT

  The FR.EVENT is generated in response to internal FR errors and commands from the FC or CC. The event message has 1 byte of data indicating a change in the following states:

    - Recording On/off, and
    - WDT occurred.

To allow the FR status and events to be recorded, the FR will write its status and event information directly into its internal output FIFO, as well as to the CAN controller.

A summary of all system messages and commands is given in Appendix B.

Software

The FR software has several groups of tasks that it must be capable of performing. It must initialize internal ports and registers, initialize the CAN controller, and interact with the Flash memory. The algorithm for the FR must accomplish all of the following:

- Initialize FR internal registers and ports
- determine operation mode,
- initialize the CAN controller,
- find the first available memory location on bootup and WDT reset,
- send and receive messages from the CAN bus controller,
- strip out all unnecessary message information,
- prepend a Flight Recorder header,
- place the message in an internal FIFO buffer,
- increment the Flash address counter,
- recognize and act on FR commands,
- generate and transmit events messages,
- write the data to the Flash,
- execute chip erase command, and
- execute data dump command.

The FR is implemented with two modes of operation: flight mode, and ground mode. The mode of operation is selected at system boot by detecting the presence or absence of a hardware jumper.

Ground mode is used in post-flight processing to transfer the data from the FR to a PC for storage and analysis. Upon activating this mode, the FR simply transmits the entire Flash memory contents over the UART port at 19.2Kbps. To connect to a PC via a serial cable, the signal will need to be shifted to RS232 levels.

In flight mode, the FR performs its primary function of data recording in a CAN bus environment.

The details on configuring the CAN controller are covered in the CAN implementation section and will not be discussed here.

Finding the first available memory location in the Flash is necessary to allow the FR subsystem to recover from a WDT or other restart conditions. The microcontroller will walk through the Flash memory until only blank memory locations are encountered, at which point the memory address will be noted and used as the start of new data. In the event of discontinuities in the data, the FC.STAT message will be used during data processing to resynchronize the data.

The FR is connected to the CANbus via the standard CAN interface discussed in the CAN section of this document.

To minimize the amount of wasted space used in the Flash, the processor will strip the control, data length and CRC bits from the CAN header. The remaining message will consist of the message ID and data bytes only. This compacted message will have a 5 bit header prepended to it before it is written to the Flash. The 5 bit header rounds the message to a byte boundary and provides a mechanism to identify the beginning of each message.

The typical write time is specified in the data sheet as 7µs. This permits a maximum data throughput between the microcontroller and the Flash of about 140K Bytes/second, which is approximately twice the bandwidth required. Any variation in the Flash write time is further compensated by buffering all incoming data in an 800 byte First-In-First-Out (FIFO) buffer. The microcontroller will manage its register use to provide the FIFO functionally, including monitoring for overflow and other error conditions.

The Flash bank has special programming requirements that the processor must take into account when writing a byte. Specifically, programming the Flash is a four-bus-cycle operation. The program command sequence is initiated by writing two unlock write cycles, followed by the program's set-up command. The program address and data are then written. Once the Flash has completed its programming cycle, the RY/BY# signal is asserted. This process is highlighted in the diagram below.

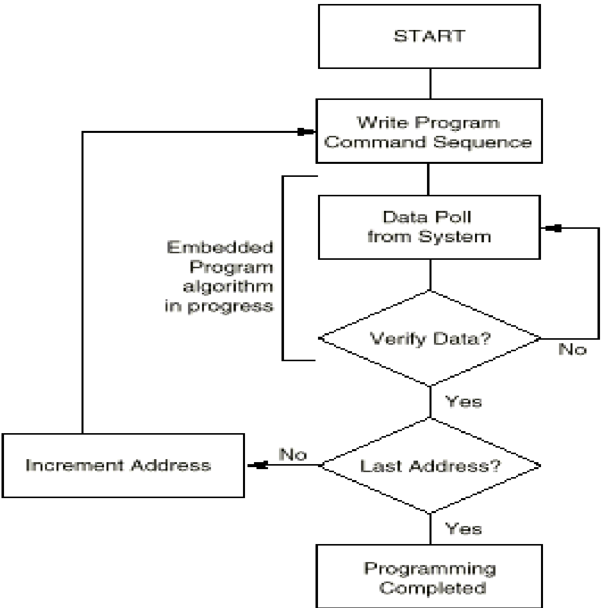


Figure 29: Programming the Flash RAM

The default mode for the Flash device is ‘read’ mode. In this mode the device is similar to any memory device. To read from the device, the system must drive the CE# and OE# pins low. CE# is the power control and selects the device. OE# is the output control and gates the data to the output pins. WE# should remain high. Reading will only be done on the ground.

Erasing the device is similar to writing to it, requiring a six-bus-cycle operation which erases the entire contents of the chip. The Flash’s chip-erase command sequence is initiated by writing two unlock cycles, followed by a set-up command. Two additional unlock write cycles are then followed by the chip erase command, which erases the chip. The Flash device verifies the erase process and asserts the RY/BY# pin when done.

As long as the FIFO has data, the main software loop will be going through the Flash write process. This consists of writing two unlock command sequences, followed by an initialization command, the address, and the data. The main loop cannot continue with the next write until the Flash chip has completed the write process, as signaled by the RY/BY# pin. The Flash address must also be incremented, preparing for the next data write.

Another task is to monitor the incoming messages for FR command messages. A command message can come from the FC or the CC and instructs the FR to perform a specific action. See the FR Messages section above.

Figure 28 shows a state flowchart of the FR program.

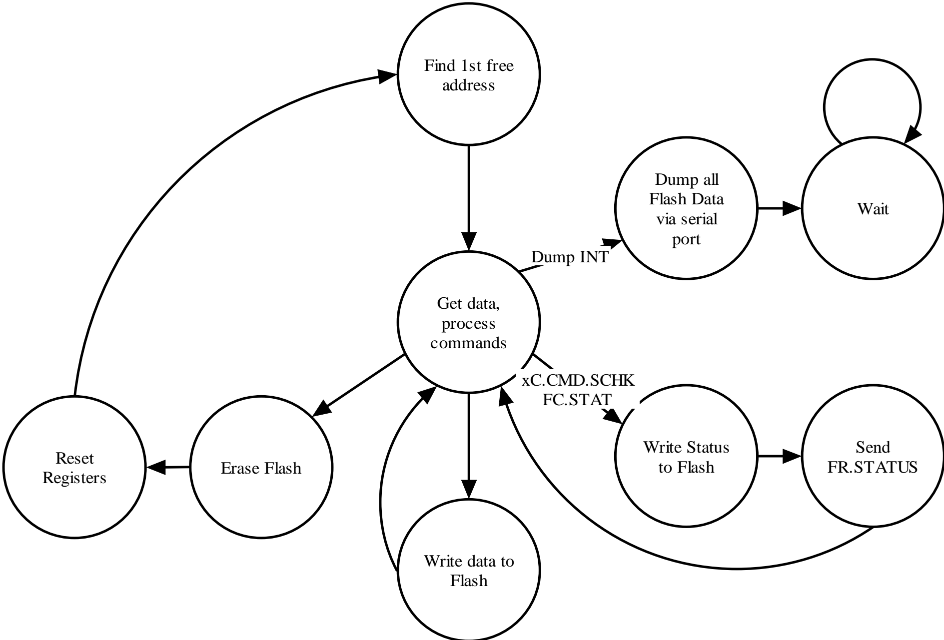


Figure 30: Program flow chart for the Flight Recorder

Igniters

The igniters are smart, self-contained devices connected to the CANbus. Igniter activation is accomplished by sending CAN messages from the FC or CC to an igniter that has received a valid authorization code (as discussed below).

This module's main tasks include:

- Igniter validation/testing
- Primary launch ignition
- Stage separation ignition

Hardware

The igniter modules consist of a PIC 16F873 connected to a pair of igniters via an ignition circuit and a separate test circuit. The igniters are embedded in either a separation charge or a launch ignition charge, depending on their application.

Due to the location of the igniters, which may often be physically distant from the avionics package or located in rocket stages that will be separated from the avionics package as the flight progresses, each IGN has its own internal power supply with linear voltage regulator and battery monitor.

The actual charges are currently Estes model rocket igniters which have been modified for high-stress environments and embedded in a black-powder capsule.

The IGN are connected to the CANbus via the standard CAN interface discussed in the CAN section of this document.

Complete schematics for the IGN can be found in Appendix A.

Messaging

The IGN accept only those valid messages that directly affect IGN operation:

- FC.CMD.POWU
- xC.CMD.POWD
  FC or CC-originated messages requesting power-up/down of all subsystems on the CANbus.
- FC.CMD.ARML
- FC.CMD.DISL
- FC.CMD.LAU
- xC.CMD.ARMn
- xC.CMD.DISn
- xC.CMD.IGNn
  FC or CC-originated messages requesting arming, disarming, or ignition of an igniter.
- xC.CMD.ICHK
  FC or CC-originated message requesting that an igniter module perform a validity check on its igniter pair.
- xC.CMD.SCHK
  FC or CC-originated message requesting status from all modules.
- FC.STAT
  FC-originated message showing internal status of FC as well as status bits for the rest of the system.
- CC.CMD.DIAG
  CC-originated message forcing all systems into diagnostic mode.
- CC.CMD.NORM
  CC-originated message forcing all systems into normal mode.

The IGN generate the following messages:

- IGN.ICHK.n (Igniter check #n, n = {0…7})
  Data resulting from an internal igniter validity check.
- IGN.EVENT.n (Igniter #n event)
  Event generated by an IGN when armed, disarmed, or fired.
- IGN.STAT.n (IGN status frame)
  IGN status, including whether the igniters are armed or disarmed, and in diagnostic or normal mode.

Software

The IGN are the most critical actuators in the system. If the stages of the rocket fail to separate, the flight will end in catastrophic failure; if the IGN trigger at an inappropriate time, there is a potential for damage to the rocket and a considerable safety risk. It is therefore necessary to ensure that the IGN trigger only at the desired times.

When the IGN first boot up, they will perform an internal status check to ensure that the actual igniters are operational and that the CANbus connection is sound. A status message will be generated that lets the FC know the condition of the IGN.

The IGN will then become idle, during which time it will respond to status requests by generating an IGN.STAT.n message. This status message contains information on the state (armed or disarmed) of the IGN and the mode the IGN is currently in (more on modes in the Operations section of this document).

IGN are unable to trigger their igniters until an authorization code is received via a valid xC.CMD.ARMx command from the FC or CC. Proper decoding of a launch code places the IGN into a ready state in which it will respond to an xC.CMD.IGNx trigger command. Upon proper receipt of an authorization code, the IGN generates and IGN.EVENT.n message to inform the FC and the CC that it is armed.

Each IGN is capable of testing its igniters to determine their condition, returning an open/short/valid status indicator. Tests can be initiated via the FC (in normal process flow) or the CC (manual intervention in emergency situations).

Operations

Messages

The efficiency and functionality of the CANbus are heavily dependent on the judicious allocation of message identifiers. These identifiers not only provide the message with a label, they also dictate the priority of the message and therefore determine the arbitration and timing the message will be subject to.

Each message identifier is an 11-bit field. When the CANbus becomes idle, all nodes requiring the use of the bus simultaneously transmit the identifiers for the messages that must be sent. Hardware-based arbitration causes any node with a lower priority message to detect this circumstance and switch to receive mode. As a result, it is necessary to allocate message identifiers with an eye toward the relative urgency associated with each message.

To this end, we subdivided the identifier field into three subfields:

- Message type ID[10:9]
- Message priority ID[8:7]
- Message ID ID[6:0]

These subfields were then further divided into priority levels.

(Note: in the following discussion all binary values are logical, not electrical.)

The Message Type field was prioritized as follows (from highest to lowest priority):

- Commands 11
- Events 10
- Responses 01
- Data 00

The Command type is used to send flight sequencing commands. Due to the extremely volatile and fast-acting nature of the rocket, commands must be given the highest Message Type priority to prevent them from being subsumed or delayed by the flood of data from the IMU and DAQ.

Events are generated by subsystems in response to some flight-critical occurrence such as apogee detection or stage separation. These messages are therefore of extreme importance in flight sequencing, and have as a result been assigned the next lower Message Type prioritization level.

Responses are Data messages that have been requested by the FC or the CC. To differentiate these from the flow of Data messages, they were therefore assigned the third Message Type priority level.

The final Message Type is Data. Data is not needed for flight sequencing, since the subsystems that generate the data also generate events whenever an interesting or important occurrence takes place. As a result, Data messages are assigned the lowest Message Type priority.

Within each of the message types, additional prioritization is accomplished with the use of the Message Priority field. This secondary field assigns relative priority to the messages within a specific type.

The Message Priority for the Command message type can take one of the following values:

- CmdHi Command High 11
- CmdLo Command Low 10
- ReqHi Request High 01
- RegLo Request Low 00

The CmdHi message priority is only assigned to Command type messages that originate from ground control via the CC. These represent critical manual intervention commands that must have immediate access to the CANbus.

The CmdLo message priority is assigned to Command type messages that originate from the FC. These represent normal flight sequencing commands that occur when the launch is proceeding as planned.

ReqHi denotes a request from the FC or CC. These messages are used to request data from a remote subsystem, such as when the FC request igniter check data from the IGN.

ReqLo is assigned to requests that have lower priority, such as status requests.

The Message Priority for all other message types can take one of the following values:

- Critical 11
- High 10
- Medium 01
- Low 00

These prioritizations have been assigned to specific messages to promote a continuous flow of messages on the CANbus. For instance, temperature and

pressure data from the DAQ could be easily drowned out by the flow of data from the IMU (which represents 95% of all bus traffic). The data messages from the DAQ are therefore assigned a High priority, while the IMU messages are allocated a Medium priority.

The remaining 7 bits of the original identifier field are then allocated as appropriate among the messages within each of the classifications outlined above. The four Message Types, each of which can have one of four Message Priorities, result in 16 priority levels that have been predefined by our design. Within each of these 16 priority levels, individual messages can be prioritized with one of the 2^7 = 128 remaining identifiers.

It should be noted for future reference that due to the presence of the bit-flip errors (discussed in the LV1 section) it is advisable that the Hamming distance between critical messages and any other system message within the same priority level be greater than one.

Modes

At any given time the avionics system will be in one of three defined modes. These include Diagnostic mode, Scrub mode, and Normal mode.

Diagnostic mode

Diagnostic mode allows ground support to safely examine the workings of each of the subsystems by placing them into a state in which critical messages (such as the triggering of an igniter) are acknowledged but not acted upon. This provides a method whereby communications and status can be checked, and a ‘dry run’ of the flight sequencing can be performed.

Scrub mode

Scrub mode is entered when an error occurs. This mode will be activated when a mission-critical event (such as an igniter failing a check) occurs while the rocket is on the launchpad. In Scrub mode, all launch authorization codes are revoked and the system is disabled. This provides ground support with maximal safety, rendering the rocket essentially harmless and inert to allow power-down and debug.

Normal mode

Normal mode encompasses all states in which the avionics package process flow is proceeding as expected. This mode can be further subdivided into five functionally discrete task phases: Initialization, Preflight, Launch, Flight, and Recovery. These task phases are discussed below.

Initialization:

When the system is first powered up, all modules perform an internal status check and update their status bits. A CC.REQ.POWU command causes the FC to assert an FC.CMD.POWU command which in turn wakes all subsystems up. The FC then enters a status loop in which the entire system continually sends status messages which are relayed to ground support. Initialization phase can only be exited by a command from the CC that either authorizes the next phase or changes the mode.

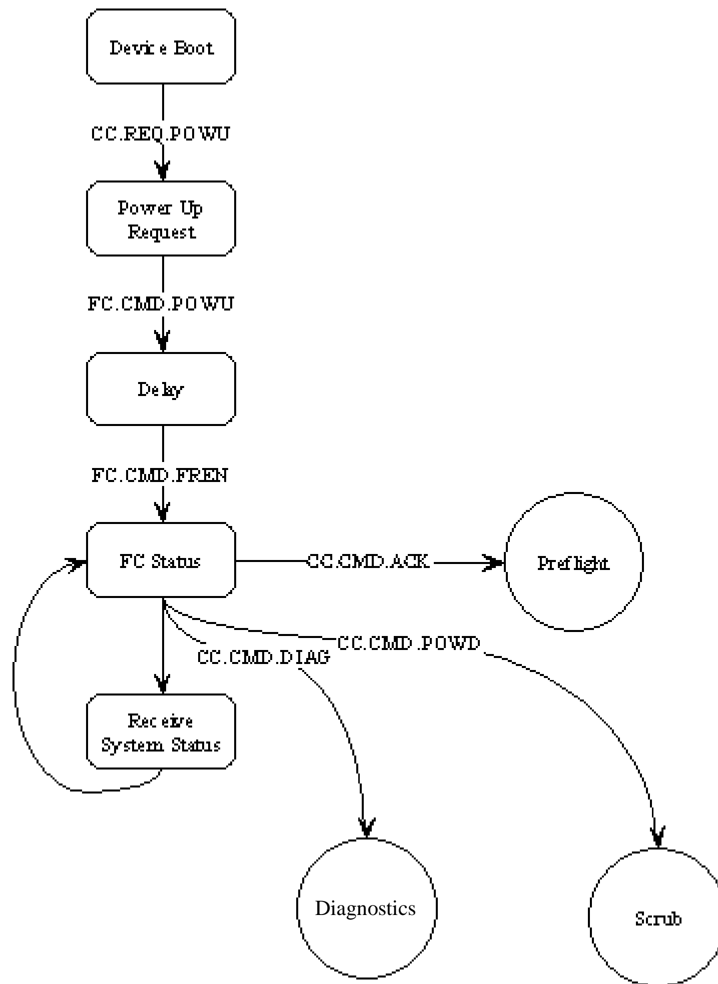


Figure 31: Initialization State Flow Diagram

Preflight:

In the Preflight phase, the system enters a stable status loop. During this phase ground support can authorize the arming or disarming of the primary launch igniter. This process causes an igniter check to be performed; if the igniter check or the communications channel fail the system will enter the Diagnostic or Scrub modes.

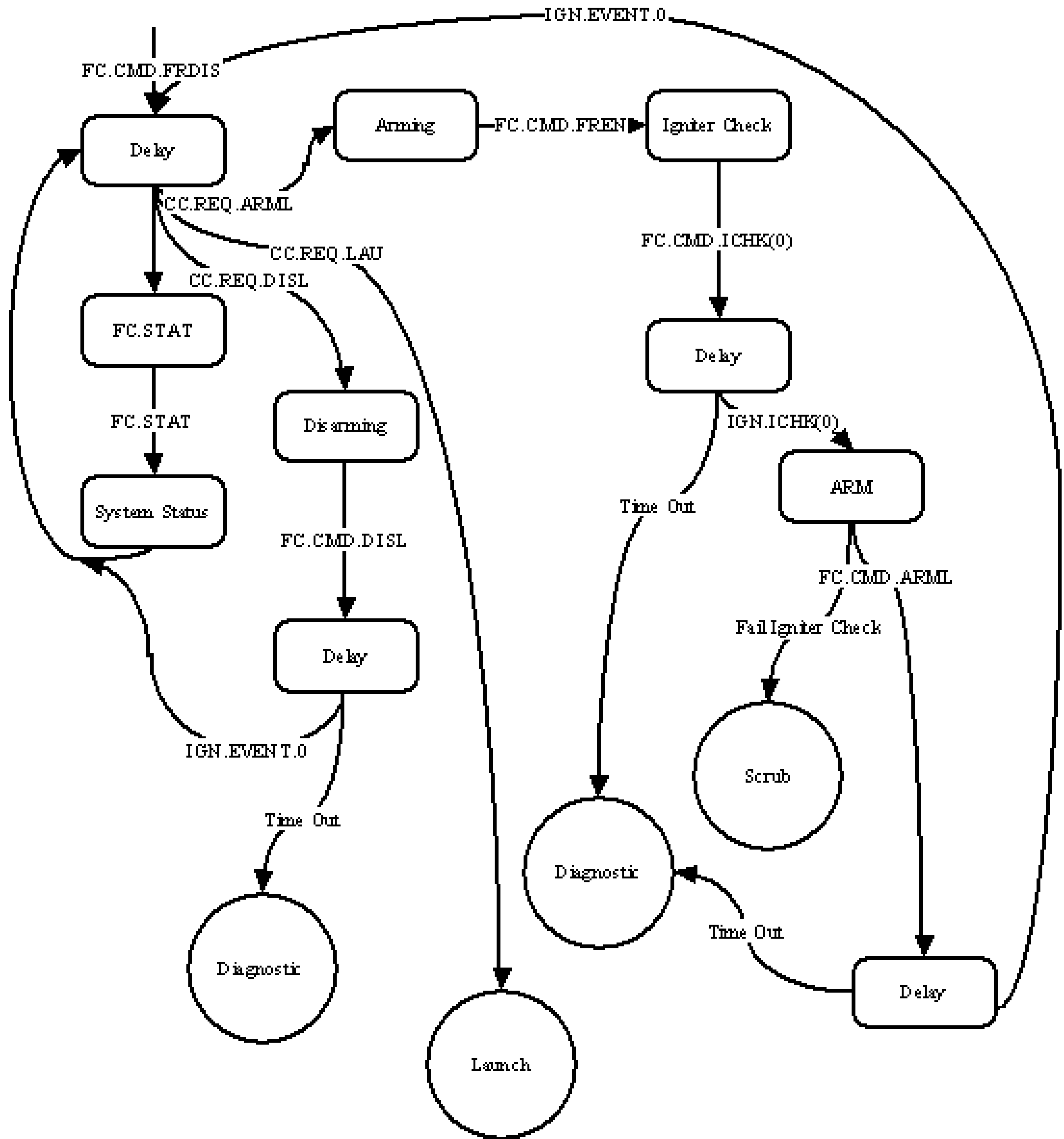


Figure 32: Preflight State Flow Diagram

Launch:

The Launch phase roughly corresponds to the '10-second countdown' phase. The Flight Recorder is enabled, the primary launch igniter is checked, and provided all system status is satisfactory the FC will start the countdown. Igniter failure will cause the system to enter scrub mode.

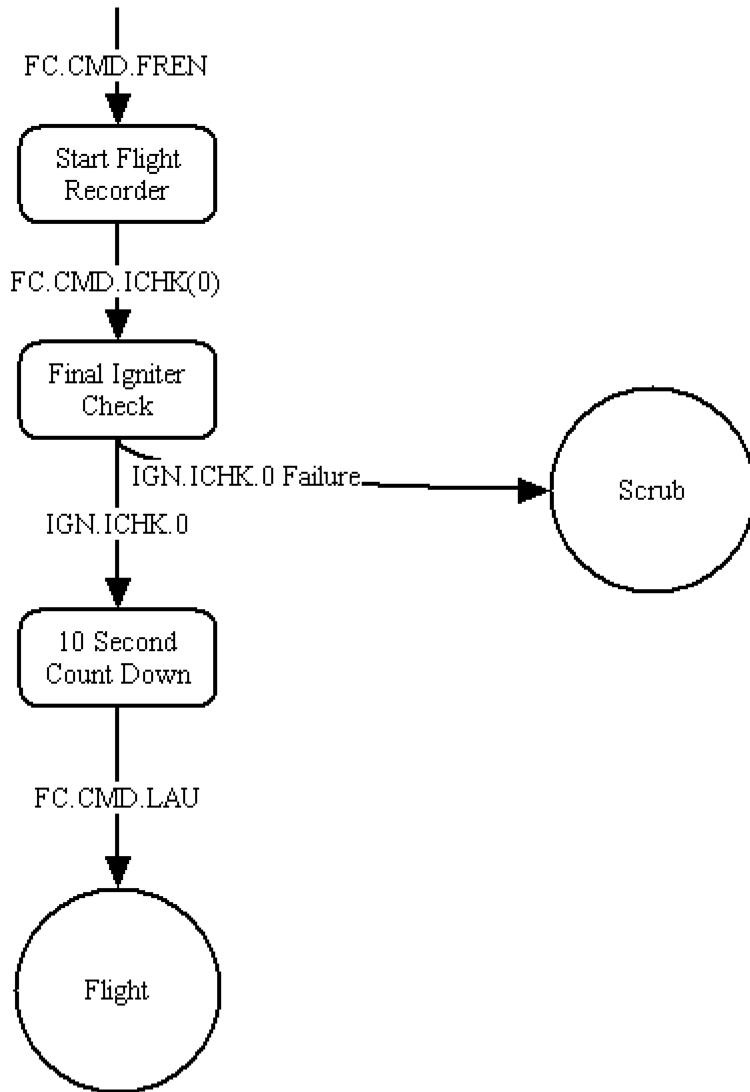


Figure 33: Launch State Flow Diagram

Flight:

The Flight phase involves the actual flight of the rocket from liftoff to chute deployment. After launch is detected by the internal sensors, flight sequencing begins. The diagram below shows the stateflow for a single-stage rocket; the process flow for a multi-stage launch vehicle would be correspondingly more complex.

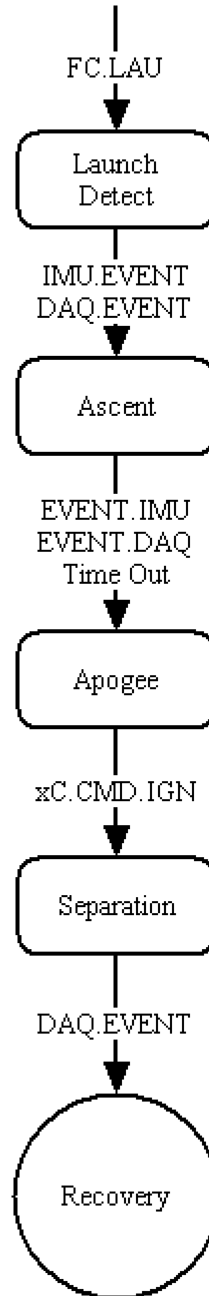


Figure 34: Flight State Flow Diagram

Recovery:

In the Recovery phase, unessential subsystems are disabled while the FC continues to transmit status frames which show the last known position of the launch vehicle. GPS systems, if used, will provide additional location information to provide ground support assistance in finding the launch vehicle when it lands.

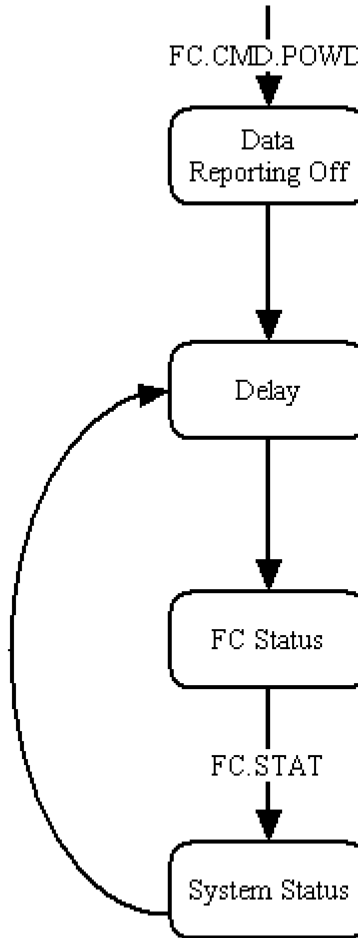


Figure 35: Recovery State Flow Diagram

Ground Support

A computer based ground support system that can seamlessly communicate with the rocket avionics must be developed to assist in the tracking and controlling of the rocket throughout its flight. The envisioned system will consist of a computer communicating with the rocket over the uplink/downlink channels. A GUI based interface for displaying flight information and data, as well as providing a means of activating commands, will need to be available to the flight operations team.

As the performance capabilities of the rocket increases, the practicalities of visually tracking its trajectory make it impossible to determine such parameters as altitude, position and velocity. With an intelligent ground based support system it will be possible to gather, display and record position, heading, velocity, flight staging, and error modes. In the event of a failure in the FR, flight data from the IMU and DAQ will be received and recorded (although at a reduced resolution).

To implement such a ground support system while assuring fault tolerant operation two computers operating with functionally identical software will be employed. The primary computer will display and record all data received from the rocket and function as the command originator, transmitting to the rocket when manual intervention or acknowledgement are required. The secondary unit will also display and record all data, but not be permitted to transmit and interfere with the primary computer. In the event of a failure in the primary ground support computer, the secondary computer will be allowed to continue where the primary left off, and send the appropriate commands to the rocket.

The software for the ground support system will utilize Labview as the software development platform. It has the capabilities to immediately display the data from the rocket in a variety of forms, and provides a familiar interface for control commands.

Plans for the future

The PSAS hopes to achieve much in the arena of high-powered amateur rocketry. The design we have created provides us with a platform that can be used as a basis for many future flight systems. To achieve our goals we plan to incrementally improve on our knowledge, systems, and processes.

A large portion of the engineering yet to be done will be an exercise in Control Engineering. The PSAS will use the lessons learned from the design of the Inertial Measurement Unit (IMU) to develop an Inertial Navigation System (INS) that realizes control of a rocket and actively guides and corrects the rocket's flight path. The ability to follow a pre-defined flight plan will provide valuable functionality for launches with scientific payloads having specific needs or that reach very high altitudes. The ultimate result of the development of an INS will be an actively guided, fin-less launch vehicle capable of maintaining stability throughout an extended flight.

Prior to implementation of the design outlined in this document there will be a number of launches (using the LV1 airframe) in which our design ideals will be incrementally qualified and airframe modifications will be tested.

The first launch of the modified LV1 will take place during September 1999 at the Black Rock, Nevada dry lakebed. Improvements will include a new communications system utilizing off the shelf spread-spectrum RF modems, more advanced and comprehensive IMU algorithms, and rollerons (gyroscopic stabilizers mounted in the fins) for improved spin rate control.

Subsequent flights of the modified LV1 will incorporate the modules designed in this document, allowing for validation of the avionics systems and exercise of the CANBus in a real-world environment.

In September 2000, the first launch of LV2 that incorporates a new and redesigned airframe and all of the avionics system specified in this design will occur.

Appendix A: System Schematics

Figure A1: System Overview Schematic 86
Figure A2: CAN Interface Schematic 87
Figure A3: Flight Computer Schematic 88
Figure A4: Communication Computer Schematic 89
Figure A5: Inertial Measurement Unit Schematic 90
Figure A6: Data Acquisition Module Schematic 91
Figure A7: Flight Recorder Schematic 92
Figure A8: Flash RAM Unit Schematic 93
Figure A9: Igniter Module Schematic 94

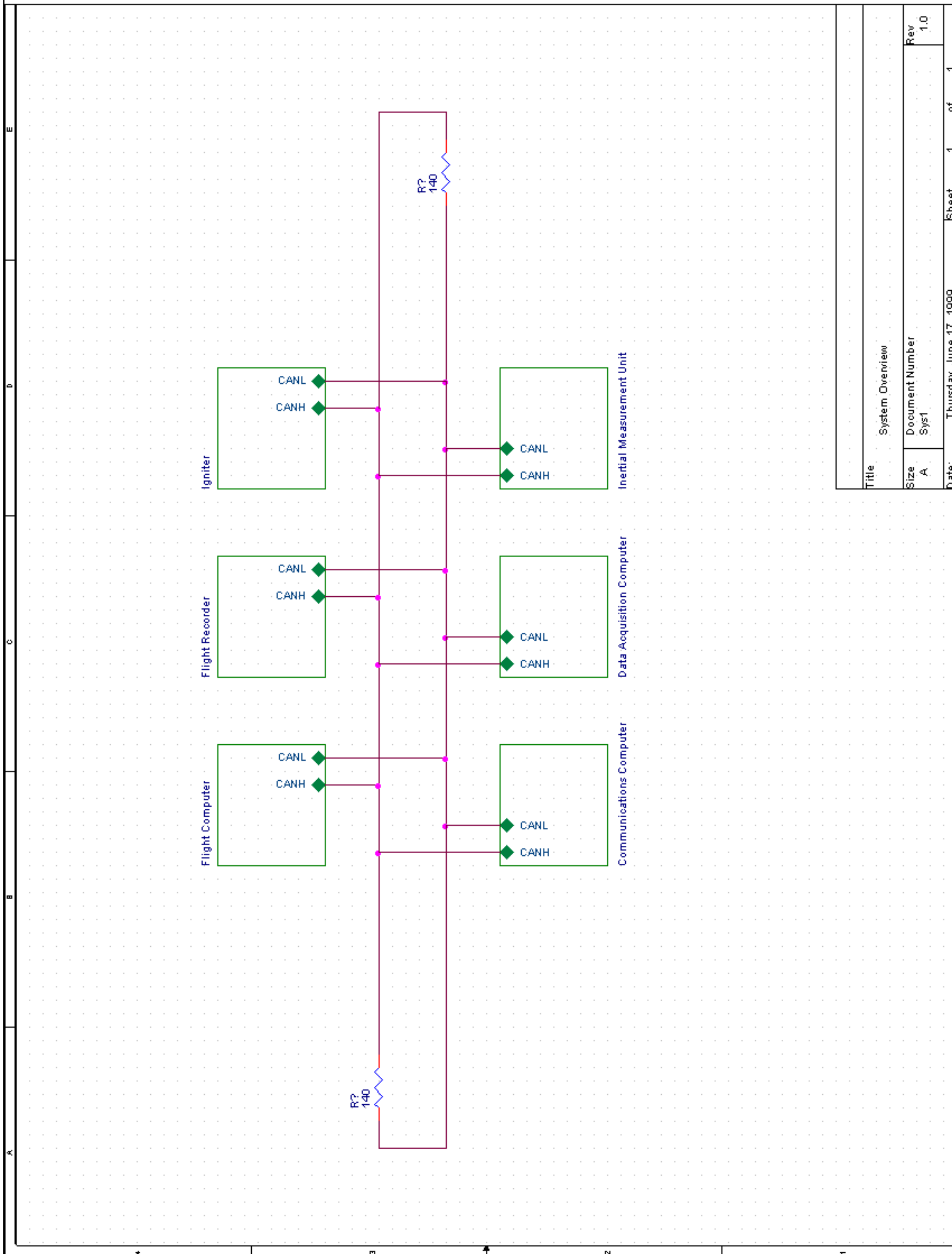


Figure A1: System Overview Schematic

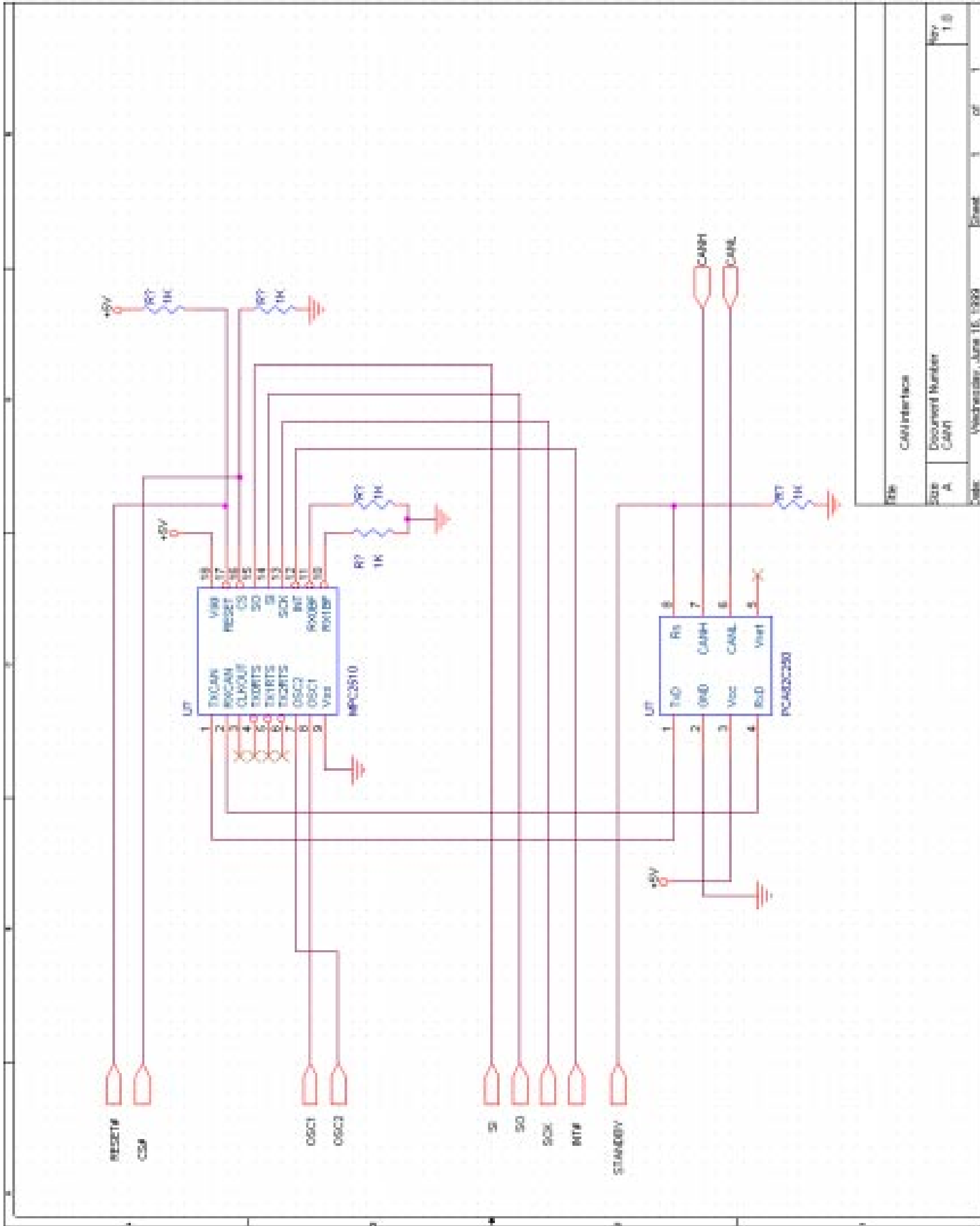


Figure A2: CAN Interface Schematic

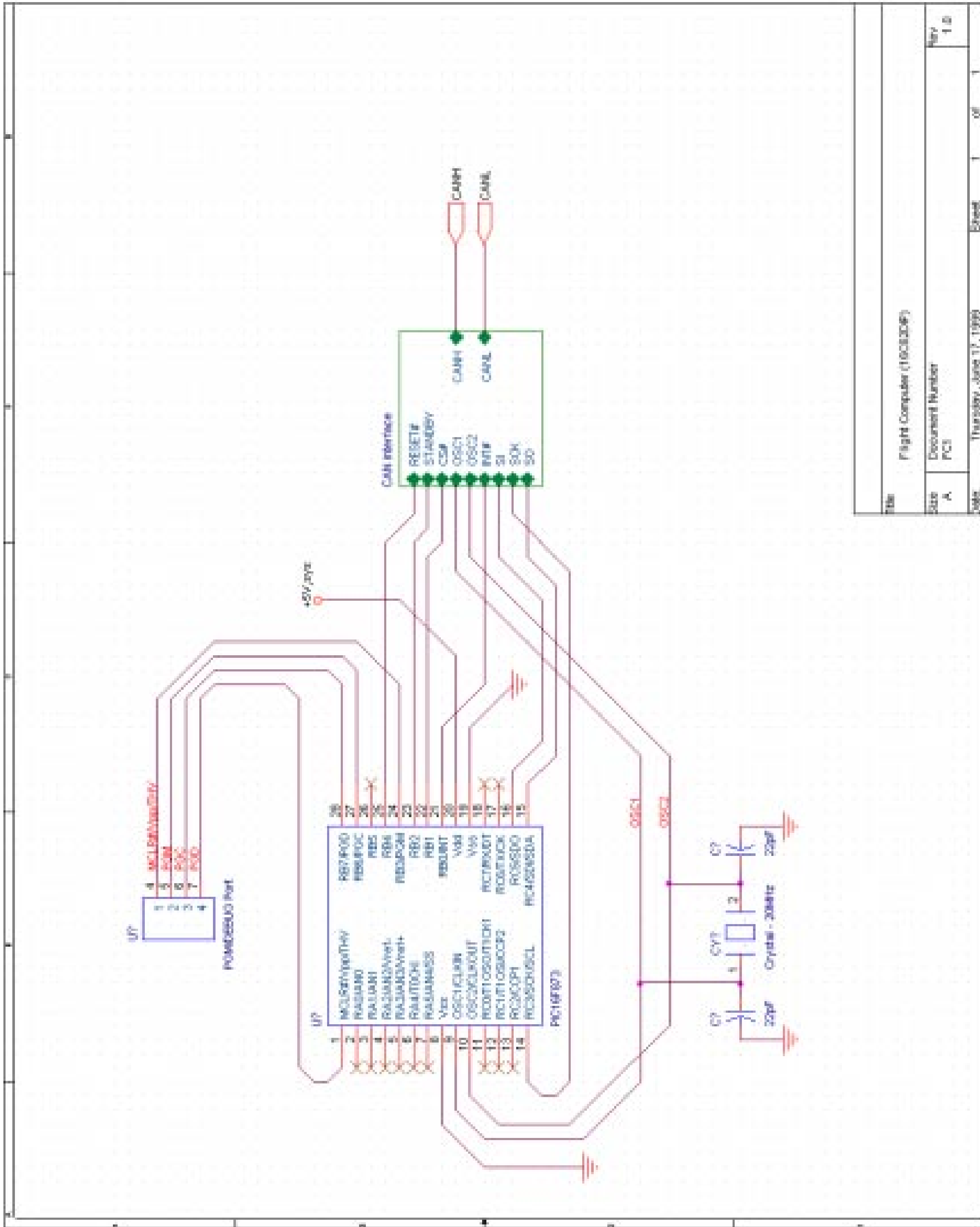


Figure A3: Flight Computer Schematic

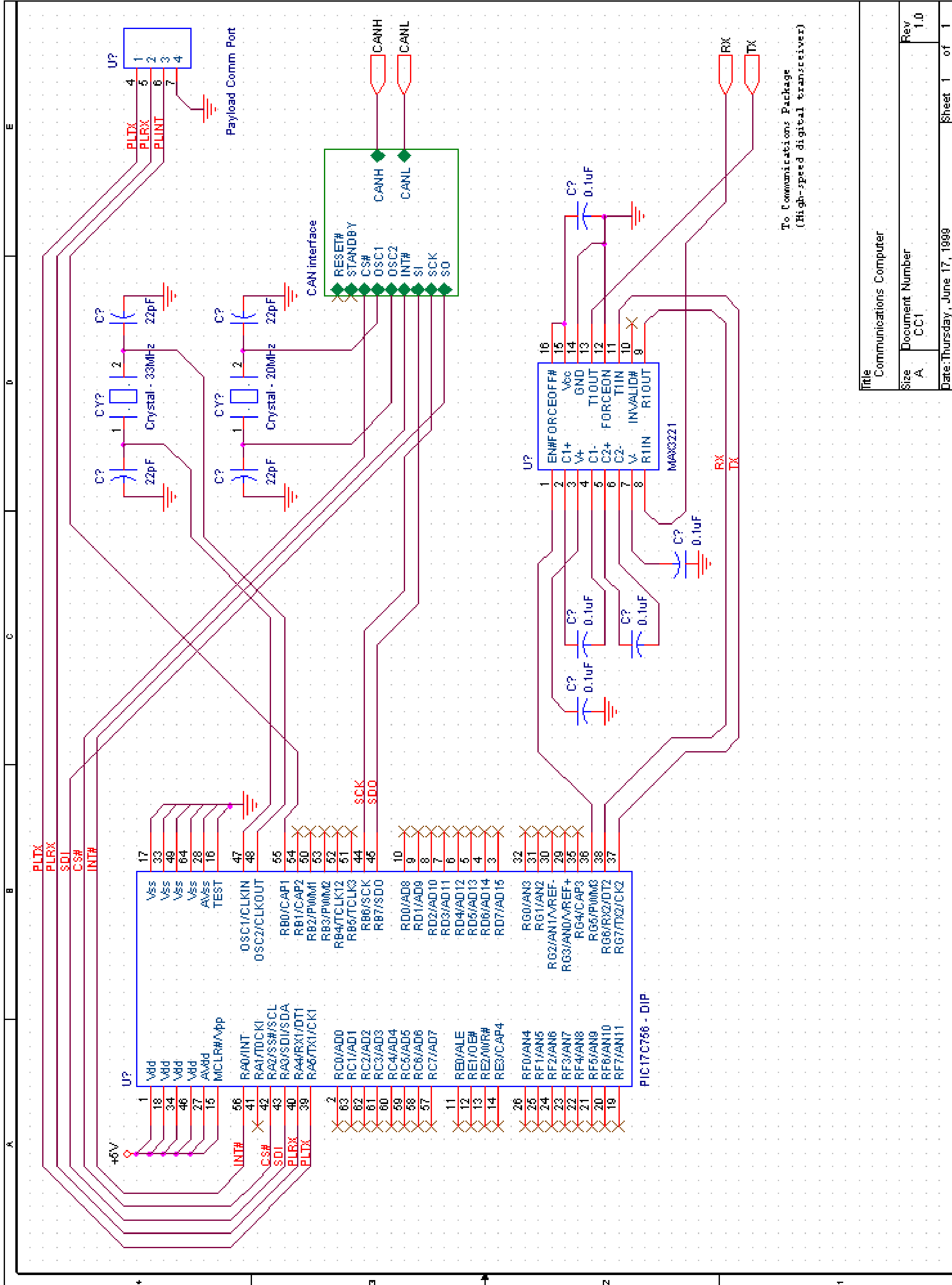


Figure A4: Communication Computer Schematic

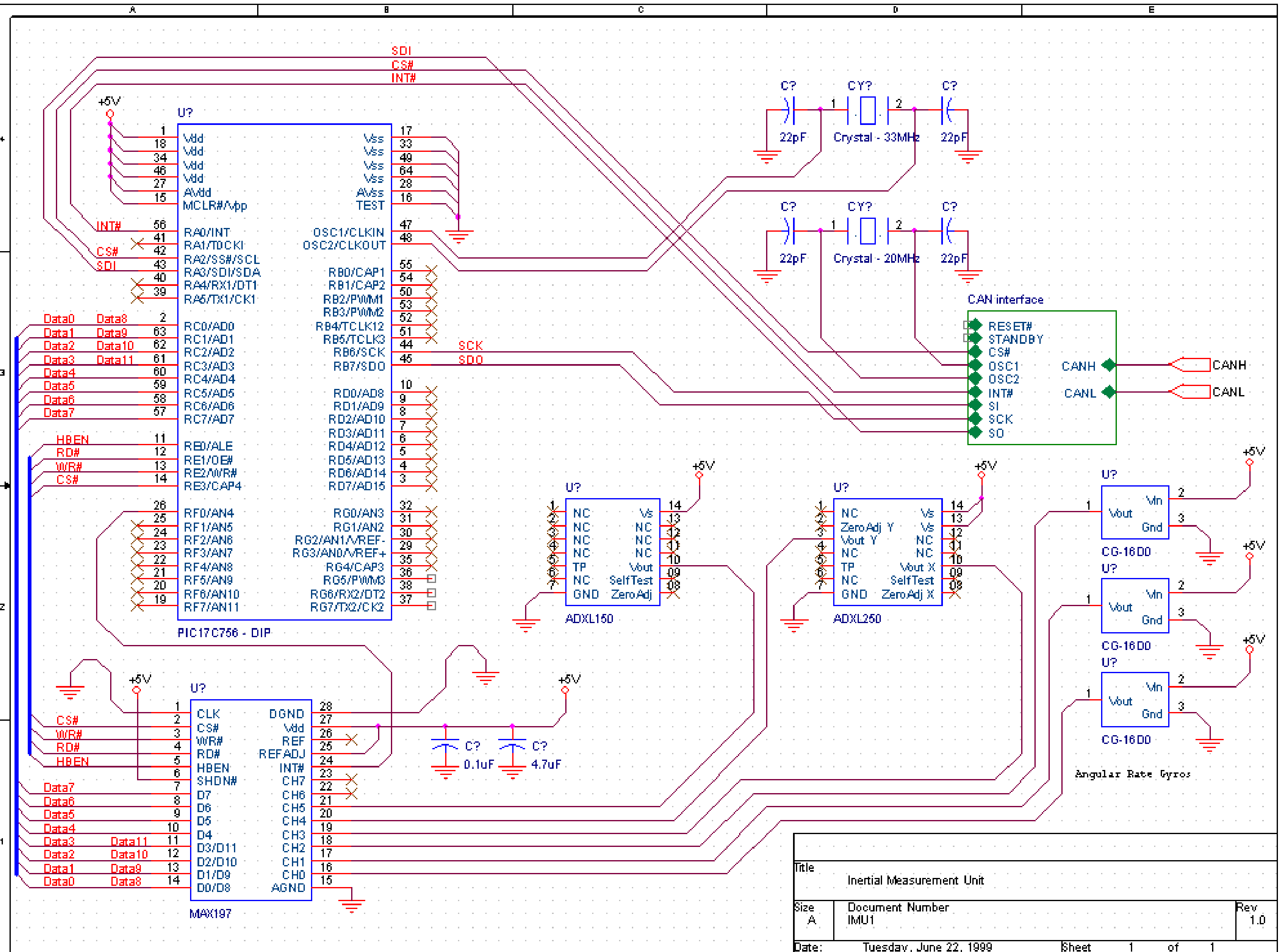


Figure A5: Inertial Measurement Unit Schematic

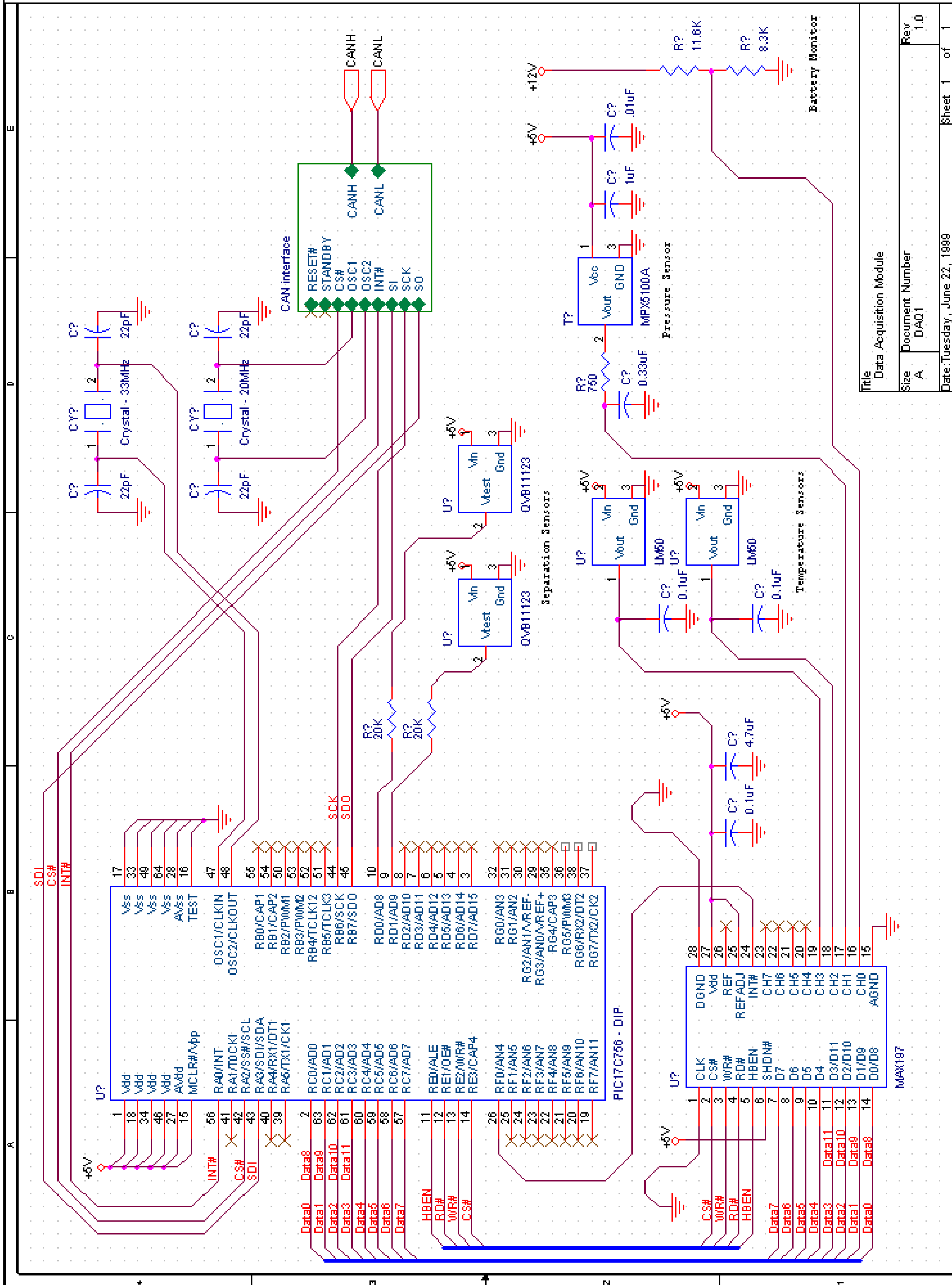


Figure A6: Data Acquisition Module Schematic

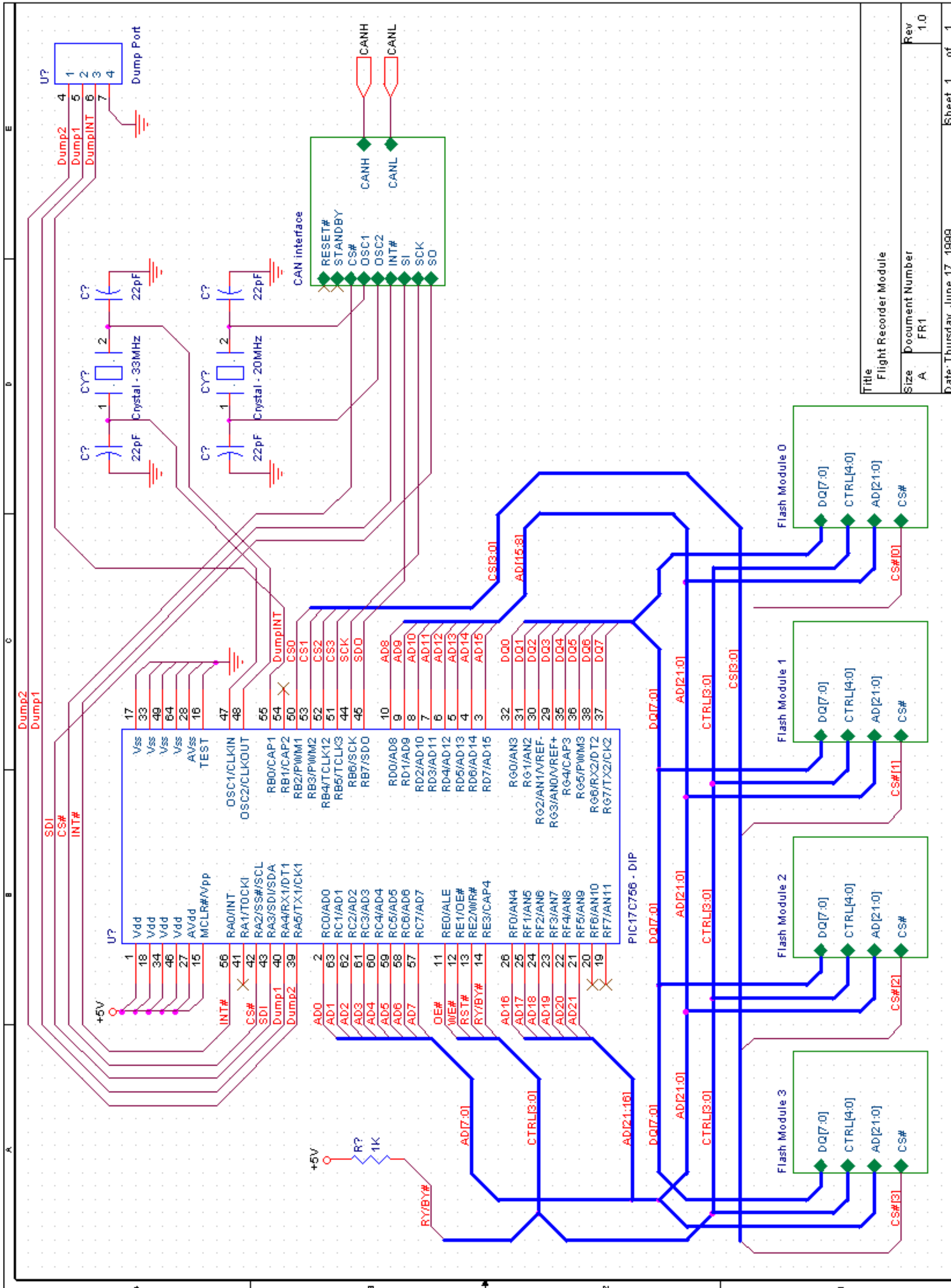


Figure A7: Flight Recorder Schematic

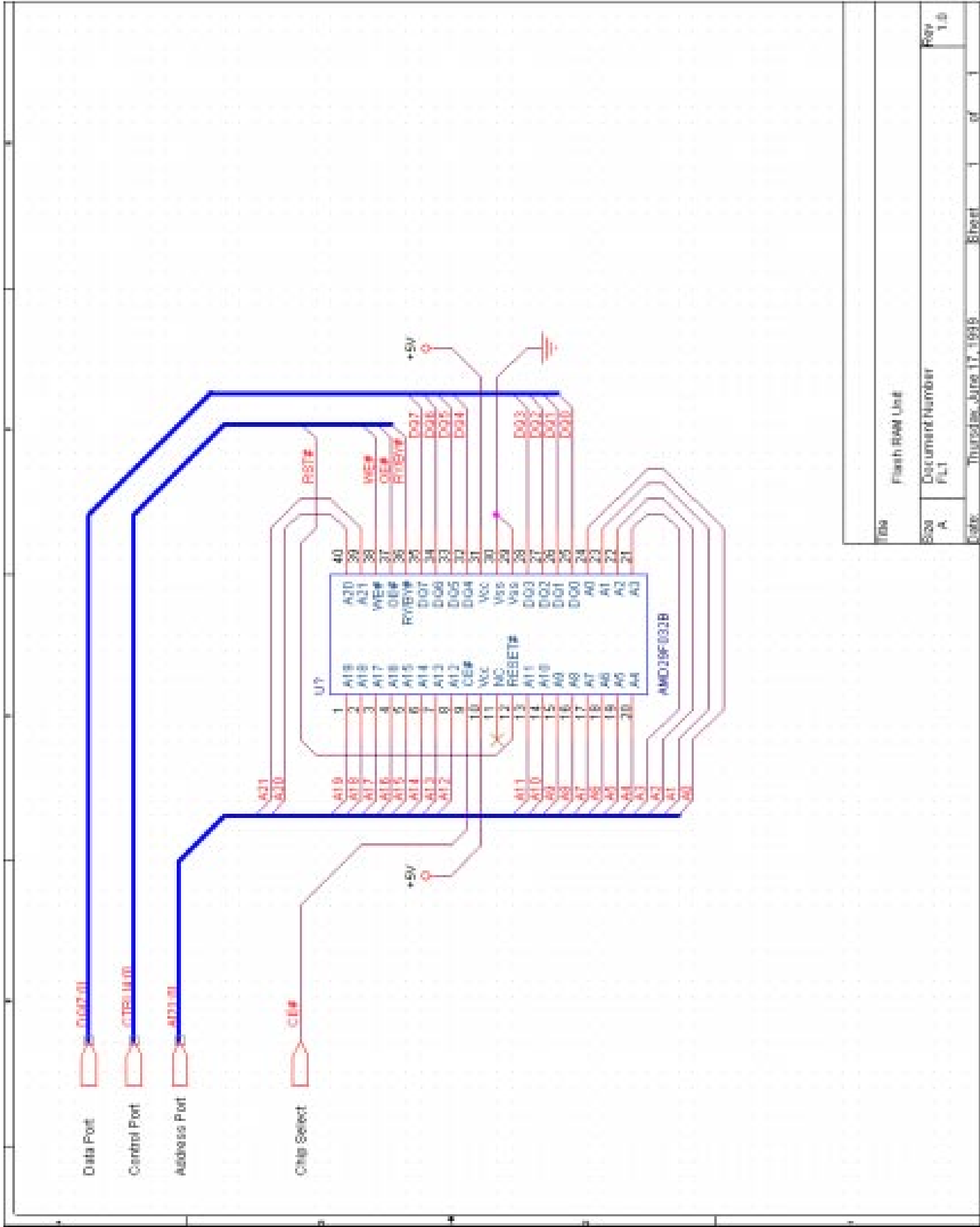


Figure A8: Flash RAM Unit Schematic

Figure A9: Igniter Module Schematic

Appendix B: Message Identifiers

IMU, FC Messages …………………………………………………………………96
CC, FR, DAQ Messages…………………………………………………………97
IGN Messages ……………………………………………………………………98

Unit Name	MSG Type	MSG Priority	MSG ID	Packet Name	Packet Description	Data Size	Target System(s)
IMU	DATA	Low	00000000000	IMU.XYZ	X,Y, & Z-axis accelerometer data, raw	5	FR, CC
	DATA	Low	00000000001	IMU.ABC	α, β, & γ-axis gyrocompass data, raw	5	FR, CC
	DATA	Med	00010001001	IMU.ACC.X	X-axis acceleration data	4	FR, CC
	DATA	Med	00010001010	IMU.ACC.Y	Y-axis acceleration data	4	FR, CC
	DATA	Med	00010001011	IMU.ACC.Z	Z-axis acceleration data	4	FR, CC
	DATA	Med	00010010001	IMU.VEL.X	X-axis velocity data	4	FR, CC
	DATA	Med	00010010010	IMU.VEL.Y	Y-axis velocity data	4	FR, CC
	DATA	Med	00010010011	IMU.VEL.Z	Z-axis velocity data	4	FR, CC
	DATA	Med	00010011001	IMU.POS.X	X-axis position data	4	FR, CC
	DATA	Med	00010011010	IMU.POS.Y	Y-axis position data	4	FR, CC
	DATA	Med	00010011011	IMU.POS.Z	Z-axis position data	4	FR, CC
	DATA	Med	00010100001	IMU.ORI.a	α-axis orientation data	4	FR, CC
	DATA	Med	00010100010	IMU.ORI.b	β-axis orientation data	4	FR, CC
	DATA	Med	00010100011	IMU.ORI.c	γ-axis orientation data	4	FR, CC
	EVENT	High	10100000000	IMU.EVENT	Event flags	8	FC,FR,CC
	RESP	Med	01010000001	IMU.STAT	IMU status frame	2	FC,FR,CC
FC	CMD	CmdLo	11100000000	FC.CMD.POWU	Power up	0	ALL
	CMD	CmdLo	11100000001	FC.CMD.POWD	Power down	0	ALL
	CMD	CmdLo	11100000010	FC.CMD.ARML	Arm launch igniter	1	LIGN,FR,CC
	CMD	CmdLo	11100000011	FC.CMD.DISL	Disarm launch igniter	1	LIGN,FR,CC
	CMD	CmdLo	11100000100	FC.CMD.LAU	Launch the rocket	1	LIGN,FR,CC
	CMD	ReqHi	11010000000	FC.CMD.ICHK	Check igniters (masked)	1	IGN
	CMD	ReqLo	11000000001	FC.CMD.SCHK	Check subsystem status (masked)	2	ALL
	CMD	CmdLo	11100001001	FC.CMD.ARM1	Arm igniter #1	1	IGN,FR,CC
	CMD	CmdLo	11100001010	FC.CMD.ARM2	Arm igniter #2	1	IGN,FR,CC
	CMD	CmdLo	11100001011	FC.CMD.ARM3	Arm igniter #3	1	IGN,FR,CC
	CMD	CmdLo	11100001100	FC.CMD.ARM4	Arm igniter #4	1	IGN,FR,CC
	CMD	CmdLo	11100001101	FC.CMD.ARM5	Arm igniter #5	1	IGN,FR,CC
	CMD	CmdLo	11100001110	FC.CMD.ARM6	Arm igniter #6	1	IGN,FR,CC
	CMD	CmdLo	11100001111	FC.CMD.ARM7	Arm igniter #7	1	IGN,FR,CC
	CMD	CmdLo	11100010001	FC.CMD.DIS1	Disarm igniter #1	1	IGN,FR,CC
	CMD	CmdLo	11100010010	FC.CMD.DIS2	Disarm igniter #2	1	IGN,FR,CC
	CMD	CmdLo	11100010011	FC.CMD.DIS3	Disarm igniter #3	1	IGN,FR,CC
	CMD	CmdLo	11100010100	FC.CMD.DIS4	Disarm igniter #4	1	IGN,FR,CC
	CMD	CmdLo	11100010101	FC.CMD.DIS5	Disarm igniter #5	1	IGN,FR,CC
	CMD	CmdLo	11100010110	FC.CMD.DIS6	Disarm igniter #6	1	IGN,FR,CC
	CMD	CmdLo	11100010111	FC.CMD.DIS7	Disarm igniter #7	1	IGN,FR,CC
	CMD	CmdLo	11100011001	FC.CMD.IGN1	Trigger igniter #1	1	IGN,FR,CC
	CMD	CmdLo	11100011010	FC.CMD.IGN2	Trigger igniter #2	1	IGN,FR,CC
	CMD	CmdLo	11100011011	FC.CMD.IGN3	Trigger igniter #3	1	IGN,FR,CC
	CMD	CmdLo	11100011100	FC.CMD.IGN4	Trigger igniter #4	1	IGN,FR,CC
	CMD	CmdLo	11100011101	FC.CMD.IGN5	Trigger igniter #5	1	IGN,FR,CC
	CMD	CmdLo	11100011110	FC.CMD.IGN6	Trigger igniter #6	1	IGN,FR,CC
	CMD	CmdLo	11100011111	FC.CMD.IGN7	Trigger igniter #7	1	IGN,FR,CC
	CMD	CmdLo	11100001000	FC.CMD.FREN	Enable FR recording	0	FR,CC
	CMD	CmdLo	11100001001	FC.CMD.FRDIS	Disable FR recording	0	FR,CC
	CMD	CmdLo	11100001010	FC.CMD.IMUON	Enable IMU output	0	IMU,CC
	CMD	CmdLo	11100001011	FC.CMD.IMUOFF	Disable IMU output	0	IMU,CC
	CMD	CmdLo	11100001100	FC.CMD.DAQON	Enable DAQ output	0	DAQ,CC
	CMD	CmdLo	11100001101	FC.CMD.DAQOFF	Disable DAQ output	0	DAQ,CC
	RESP	Critical	01110000000	FC.STAT	FC status frame	5	ALL

Unit Name	MSG Type	MSG Priority	MSG ID	Packet Name	Packet Description	Data Size	Target System(s)
CC	CMD	CmdHi	11110000000	CC.REQ.POWU	Power up (request to FC)	0	FC
	CMD	CmdHi	11110000001	CC.CMD.POWD	Power down (request to FC)	0	ALL
	CMD	CmdHi	11110000010	CC.CMD.ACK	Acknowledge boot status	1	FC,FR
	CMD	CmdHi	11110000011	CC.REQ.ARML	Arm launch igniter (request to FC)	1	FC
	CMD	CmdHi	11110000100	CC.REQ.DISL	Disarm launch igniter (request to FC)	1	FC
	CMD	CmdHi	11110000101	CC.REQ.LAU	Launch the rocket (request to FC)	1	FC
	CMD	ReqHi	11010000000	CC.CMD.ICHK	Check igniters (masked)	1	IGN
	CMD	ReqLo	11000000001	CC.CMD.SCHK	Check subsystem status (masked)	2	ALL
	CMD	CmdHi	11110100001	CC.CMD.ARMn	Arm igniter #1	1	FC,IGN,FR
	CMD	CmdHi	11110100010	CC.CMD.ARMn	Arm igniter #2	1	FC,IGN,FR
	CMD	CmdHi	11110100011	CC.CMD.ARMn	Arm igniter #3	1	FC,IGN,FR
	CMD	CmdHi	11110100100	CC.CMD.ARMn	Arm igniter #4	1	FC,IGN,FR
	CMD	CmdHi	11110100101	CC.CMD.ARMn	Arm igniter #5	1	FC,IGN,FR
	CMD	CmdHi	11110100110	CC.CMD.ARMn	Arm igniter #6	1	FC,IGN,FR
	CMD	CmdHi	11110100111	CC.CMD.ARMn	Arm igniter #7	1	FC,IGN,FR
	CMD	CmdHi	11110101001	CC.CMD.DISn	Disarm igniter #1	1	FC,IGN,FR
	CMD	CmdHi	11110101010	CC.CMD.DISn	Disarm igniter #2	1	FC,IGN,FR
	CMD	CmdHi	11110101011	CC.CMD.DISn	Disarm igniter #3	1	FC,IGN,FR
	CMD	CmdHi	11110101100	CC.CMD.DISn	Disarm igniter #4	1	FC,IGN,FR
	CMD	CmdHi	11110101101	CC.CMD.DISn	Disarm igniter #5	1	FC,IGN,FR
	CMD	CmdHi	11110101110	CC.CMD.DISn	Disarm igniter #6	1	FC,IGN,FR
	CMD	CmdHi	11110101111	CC.CMD.DISn	Disarm igniter #7	1	FC,IGN,FR
	CMD	CmdHi	11110110001	CC.CMD.IGNn	Trigger igniter #1	1	FC,IGN,FR
	CMD	CmdHi	11110110010	CC.CMD.IGNn	Trigger igniter #2	1	FC,IGN,FR
	CMD	CmdHi	11110110011	CC.CMD.IGNn	Trigger igniter #3	1	FC,IGN,FR
	CMD	CmdHi	11110110100	CC.CMD.IGNn	Trigger igniter #4	1	FC,IGN,FR
	CMD	CmdHi	11110110101	CC.CMD.IGNn	Trigger igniter #5	1	FC,IGN,FR
	CMD	CmdHi	11110110110	CC.CMD.IGNn	Trigger igniter #6	1	FC,IGN,FR
	CMD	CmdHi	11110110111	CC.CMD.IGNn	Trigger igniter #7	1	FC,IGN,FR
	CMD	CmdHi	11110001001	CC.CMD.FCON	Enable FC status output	0	FC
	CMD	CmdHi	11110001010	CC.CMD.FCOFF	Disable FC status output	0	FC
	CMD	CmdHi	11110001011	CC.CMD.FREN	Enable FR recording	0	FR
	CMD	CmdHi	11110001100	CC.CMD.FRDIS	Disable FR recording	0	FR
	CMD	CmdHi	11110001101	CC.CMD.FRERA	Enable FR erase	0	FR
	CMD	CmdHi	11110001110	CC.CMD.IMUON	Enable IMU output	0	IMU
	CMD	CmdHi	11110001111	CC.CMD.IMUOFF	Disable IMU output	0	IMU
	CMD	CmdHi	11110010000	CC.CMD.DAQON	Enable DAQ output	0	DAQ
	CMD	CmdHi	11110010001	CC.CMD.DAQOFF	Disable DAQ output	0	DAQ
	CMD	CmdHi	11110010010	CC.CMD.FKS	Force FC state	2	FC,FR
	CMD	CmdHi	11110010011	CC.CMD.DIAG	Enter diagnostic mode	0	ALL
	CMD	CmdHi	11110010100	CC.CMD.NORM	Enter normal mode	0	ALL
	RESP	Med	01010000001	CC.STAT	CC status (+status for UL, DL, PL)	4	FC,FR
FR	EVENT	High	10100000001	FR.EVENT	FR events (recording on/off)	1	FC,CC
	RESP	Med	01010000010	FR.STAT	FR status frame	2	FC,CC
DAQ	DATA	High	00100000010	DAQ.TEMP	Temperature sensor data	2	FR,CC
	DATA	High	00100000011	DAQ.PRESS	Pressure sensor data	2	FR,CC
	EVENT	High	10100000010	DAQ.EVENT	Event flags	8	FC,FR,CC
	RESP	Med	01010000011	DAQ.STAT	DAQ status frame +battery monitor	2	FC,FR,CC

Unit Name	MSG Type	MSG Priority	MSG ID	Packet Name	Packet Description	Data Size	Target System(s)
IGN	RESP	Critical	01111100000	IGN.ICHK.0	Igniter check #0	1	FC,FR,CC
	RESP	Critical	01111100001	IGN.ICHK.1	Igniter check #1	2	FC,FR,CC
	RESP	Critical	01111100010	IGN.ICHK.2	Igniter check #2	3	FC,FR,CC
	RESP	Critical	01111100011	IGN.ICHK.3	Igniter check #3	4	FC,FR,CC
	RESP	Critical	01111100100	IGN.ICHK.4	Igniter check #4	5	FC,FR,CC
	RESP	Critical	01111100101	IGN.ICHK.5	Igniter check #5	6	FC,FR,CC
	RESP	Critical	01111100110	IGN.ICHK.6	Igniter check #6	7	FC,FR,CC
	RESP	Critical	01111100111	IGN.ICHK.7	Igniter check #7	8	FC,FR,CC
	EVENT	High	10100000000	IGN.EVENT.0	Igniter event #0	2	FC,FR,CC
	EVENT	High	10100000001	IGN.EVENT.1	Igniter event #1	3	FC,FR,CC
	EVENT	High	10100000010	IGN.EVENT.2	Igniter event #2	4	FC,FR,CC
	EVENT	High	10100000011	IGN.EVENT.3	Igniter event #3	5	FC,FR,CC
	EVENT	High	10100000100	IGN.EVENT.4	Igniter event #4	6	FC,FR,CC
	EVENT	High	10100000101	IGN.EVENT.5	Igniter event #5	7	FC,FR,CC
	EVENT	High	10100000110	IGN.EVENT.6	Igniter event #6	8	FC,FR,CC
	EVENT	High	10100000111	IGN.EVENT.7	Igniter event #7	9	FC,FR,CC
	RESP	Med	01010000000	IGN.STAT.0	IGN status frame #0	4	FC,FR,CC
	RESP	Med	01010000001	IGN.STAT.1	IGN status frame #1	5	FC,FR,CC
	RESP	Med	01010000010	IGN.STAT.2	IGN status frame #2	6	FC,FR,CC
	RESP	Med	01010000011	IGN.STAT.3	IGN status frame #3	7	FC,FR,CC
	RESP	Med	01010000100	IGN.STAT.4	IGN status frame #4	8	FC,FR,CC
	RESP	Med	01010000101	IGN.STAT.5	IGN status frame #5	9	FC,FR,CC
	RESP	Med	01010000110	IGN.STAT.6	IGN status frame #6	10	FC,FR,CC
	RESP	Med	01010000111	IGN.STAT.7	IGN status frame #7	11	FC,FR,CC

Appendix C: References

Microcontrollers:

"PIC 17C75X – High-Performance 8-Bit CMOS EPROM Microcontrollers"
Microchip Technology Inc., 1997
http://www.microchip.com/10/Lit/PICmicro/17C75X/index.htm

"PIC 16F87X - 8-Bit CMOS Flash Microcontroller"
Microchip Technology Inc., 1997
http://www.microchip.com/10/Lit/PICmicro/16F87X/index.htm

CAN:

"PCA82C250 - CAN controller interface"
Philips Semiconductor Co., 1994
http://www-us.semiconductors.philips.com/pip/PCA82C250_3

"Stand-Alone CAN Controller with SPI Interface"
Microchip Technology Inc., 1999
http://www.ee.pdx.edu/~aess/DataSheets/CAN/controllers/Microchip_MCP2510.pdf

"CAN Specification, version 2.0"
Bosch Industries, 1994
http://www.ee.pdx.edu/~aess/DataSheets/CAN/CAN/CAN_Spec_2.0b.pdf

"A perspective to the design of distributed real–time control applications based on CAN"
Martin Törngren, 1995
http://www.ee.pdx.edu/~aess/DataSheets/CAN/CAN/CANvsFIP_Paper.pdf

Components:

"Am29F032B 32 - Megabit CMOS 5.0 Volt Uniform Sector Flash Memory"
Advanced Micro Devices, Inc., 1998
http://www.amd.com/products/nvd/techdocs/21610.pdf

"LM50 – 3 -Terminal Adjustable Current Source"
National Semiconductor Corporation, 1998
http://www.national.com/pf/LM/LM50.html

"MPX5100 – Integrated Silicon Pressure Sensor"
Motorola, Inc. 1998
http://www.mot-sps.com/cgi-bin/get?/books/apnotes/pdf/an1653*.pdf

"QVB11123 - Slotted Optical Switch"
QT Optoelectronics,
http://www.qtopto.com/ir/i0901.htm

"TLC2543I - 12bit A/D converters with serial control and 11 analog inputs"
Texas Instruments Incorporated, 1997
http://www.ti.com/sc/docs/folders/analog/tlc2543.html

"ADXL150/ADXL250 Low-Noise/Power, Single/Dual Axis Accelerometers"
Analog Devices, 1998
http://www.analog.com/pdf/ADXL150_250_0.pdf

"CG -16D0 - Ceramic Gyro"
TOKIN SENSORS,
http://www.intercraft.co.jp/tokin/ENGLISH/t_file/data/gl_07e.pd