CAN Based Flight Computer and Avionics System for Small Satellite Applications

A project presented to the Faculty of Aerospace Engineering
San Jose State University
in fulfillment of the requirements for the degree
Master of Science in Aerospace Engineering

by

Aric M. Garcia

December 2015

The Designated Project Committee Approves the Project Titled

CAN BASED FLIGHT COMPUTER AND AVIONICS SYSTEM FOR SMALL SATELLITE APPLICATIONS

by

Aric M. Garcia

APPROVED FOR THE DEPARTMENT OF AEROSPACE ENGINEERING

SAN JOSÉ STATE UNIVERSITY

December 2015

Dr. Nikos J. Mourtos, Committee Chair *Date*
Department of Aerospace Engineering

Matt Messana *Date*
Skybox Imaging + Google

ABSTRACT

CAN Based Flight Computer and Avionics System

for Small Satellite Applications

by

Aric M. Garcia

Spacecraft are nearly full autonomous systems and some are completely autonomous but either way a flight computer and avionics system is always required to maintain the health and safety of the spacecraft and to control all aspects of the various subsystems defined by SMAD. Currently most satellites use an inter-computer communication bus similar to most vehicles except that most of what is used is still not as good as your average sedan. The automotive industry uses controller automated network as the light weight and high speed communication system that carries information throughout the various systems. Also modern day aerospace engineering students and space-tech aspiring high school students don't always have a firm understanding of electronics and programming or how to combine the two. Therefore there is a need for a cheap, affordable, and robust spacecraft flight computer that is can be easily assimilated into any small sat project.

Table of Contents

1. Introduction

1.1 Motivation

I started working at Skybox Imaging in January 2014 and since have learned a lot about spacecraft hardware. Mechatronics and spacecraft have always been my interests and I have learned that the avionics, command, and data handling subsystems are very much what I enjoy about engineering. Other motivations are my fellow aerospace students who do not have a firm grasp on or do not understand programming and circuit design and how to combine the two. I have the ability to design a system to help improve the capabilities of aspiring students in spacecraft engineering the opportunity should be capitalized upon. The following project was done to simplify the task of designing a small satellite.

1.2 Background

Space flight and space exploration is one of the newest, most extreme, and least explored of the environments known to man. It introduces an environment that is a complete vacuum of extreme cold or intense heat where radiation is abundant. Having the ability to transverse such an expanse is not only a challenge but also a necessity for humanity. Catching a ride to this frontier and traversing it is incredibly expensive and dangerous and therefore requires accurate and precise dynamic control of all aspects of a space vehicle. So it is up to engineers to build systems that not only are resistant to the harsh environment but can maintain absolute control for the safety of the payload (both human and scientific) even in the event that a failure in one or more systems were to

occur. In this light it is desirable to design an autonomous and intelligent control system that can resolve any errors or issues that it comes across on its own to ensure complete and consistent operability of the spacecraft. This system is called the spacecraft flight computer and is the primary component of the command and data handling subsystem. Flight computers have been around in aircraft since the early 1930's and the first use of a computer for space system control was the lunar extraction module during the Apollo moon missions. Since then spacecraft flight computers have been used on nearly every space mission. Flight computers are very expensive because they must be very robust and reliable. If a computer were to fail on startup or anytime through the mission without recovery the mission would be a bust and the total cost of the mission in time and money would go to waste. Cube satellites that are built by college and high school students also require flight computers. However the cost of a reliable flight computer is far too great for most students to afford and is too big and bulky to fit in a cube sat frame. So small unreliable microcontrollers that are prone to failure are generally used for use aboard such spacecraft. Also in general the code complexity is too great for most young engineers, which cause them to tend towards simpler less reliable microcontrollers. This is reason for a cheap, simple, and reliable controller that a student learning about spacecraft and space systems could easily afford and program for a basic satellite project.

Flight computers have been in use since the early 1930's in the first of the fly by wire aircraft when the use of electricity to maintain control of a planes attitude and control surfaces became important. Later it became a crucial part of plane operation when control of supersonic aircraft became too difficult for pilot control alone. The soviet Tupolev ANT-20 [1] used the first fly by wire system but the first non-experimental

plane that used fly-by-wire was the Avro Canada CF-105 Arrow. Invert wing fighter jets like the Grumman X-29 would be uncontrollable if not for the flight computer [2]. Because the jet had many inherently unstable characteristics it needed to have a highly computerized fly by wire system to maintain stability.

In the late 1960's when the Apollo missions were beginning to put men on the moon the first spacecraft flight computers were used to maintain control of the spacecraft and the safety of the crew and payload. Many fault alert systems were built into the spacecraft computers, which in some cases saved the lives of the crew. The computers that were used at the time were iron core memory. Simply put, it was a system of copper wire that flowed in and out of iron donuts. An iron donut would act as a transistor in the memory and binary signals could be created that would be used to communicate to other systems and maintain control of the overall system. That kind of computer was very large and bulky but was also resistant to radiation since bit flips are not possible with the iron core memory. [3]

Iron core memory was used in other space missions since but only in part due to the fact that is was extraordinarily large and heavy with far too little computational capacity. The space shuttle used iron core memory in the AP-101B computer because of the early state of semi-conductor technology and the resistance it proved to radiation. However the space shuttle was constricted to approximately 400kB of memory. Modern spacecraft use integrated circuits that are resistant to radiation also known as radiation hardened. Radiation

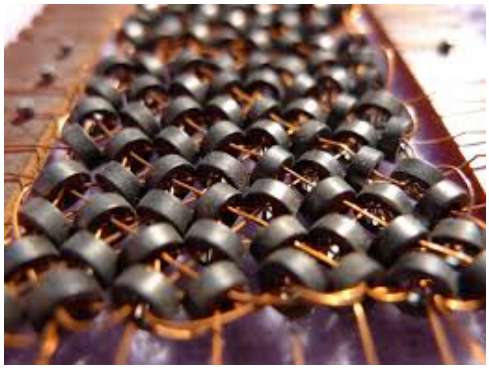


Figure 1.1 Iron core rope memory

hardened means that it is made with thicker and bulkier parts that increase radiation resistance or just a shield against the suns radiation with thick covering. [4]

Skybox Imaging of Mountain View California currently incredible hardware redundancy and powerful fault tolerance logic to combat radiation-induced problems. At the heart of these small satellites is a system of small non-rad hardened computers connected through a 1 Megabit CAN bus communication network to create a cheap but effective spacecraft flight computer and avionics system. It is a higher risk to include non-rad hardened parts but it is cheaper in the long run. This concept has been proven through the satellites at Skybox.

1.3 LEO Radiation Environment: Van Allen Belts

The radiation environment that is viewed by satellites and other spacecraft in orbit about earth is a dangerous hazard. Although usually predictable the radiation cannot be avoided and therefore must be designed for. The source of radiation about earth comes primarily from the Van Allen belt, which surrounds earth. The Van Allen belt is an area of highly charged ions or radiation that have accumulated and move very quickly around earth. The Van Allen belts are a result of the earth's magnetic field and its interaction with the sun. There are two belts that surround earth the first of which begins at an altitude of approximately 1000Km and goes to 6,000Km. The second begins at 36,000Km and extends to 84,000Km [5]. The existence

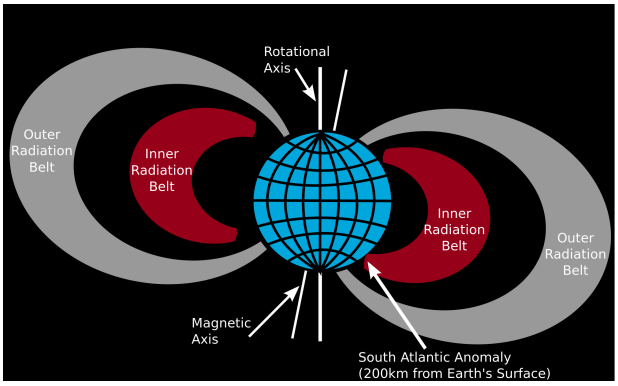


Figure 1.2 Van Allen belts showing SAA location [5]

of the belts was discovered in 1958 by the explorer 1 spacecraft and was further studied in 2012 by the Van Allen probes also known as the Radiation Belt Storm Probes (RBSP) [6]. It was discovered that high solar winds can cause the upper Van Allen belt to push up against the inner belt and for the lower belt to push lower into earth's plasmasphere. It was also discovered that there exists a region in low earth orbit where inconsistencies in earth's core cause the lower Van Allen belt to drop to around 200Km altitude. This region of space is called the South Atlantic Anomaly (SAA) and is a danger to small satellites that are usually covered in a radiation-unburdened bubble below the first belt. The SAA is located near the South Atlantic Ocean and is situated directly above the lower tip of South America. The Van Allen belt also dips at the north and south poles where earth's magnetic field flows through [6].

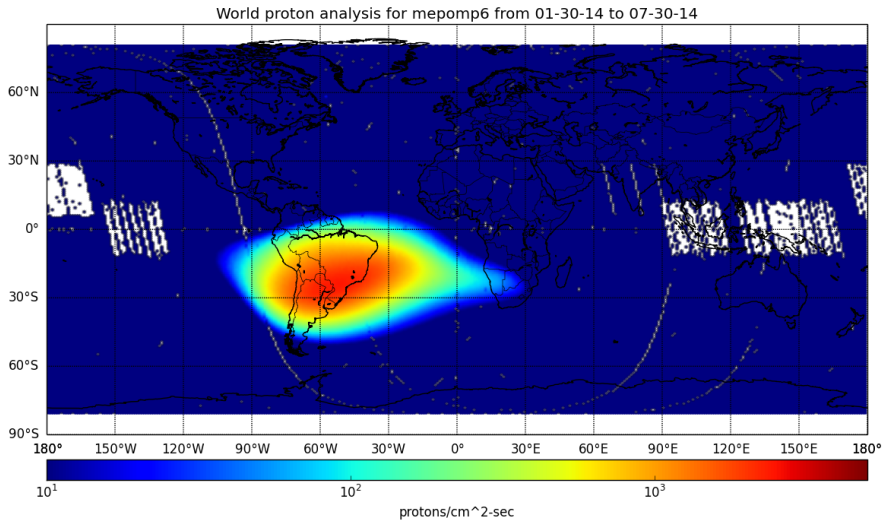


Figure 1.3 SAA as seen January 2014 [5]

The radiation environment can cause many problems in the electronics that make up the computation and avionics systems. The problems that can occur are but are not limited to: TID, LET, SEE, SEU, SEL, SEFI, SEB, and SEGR. Each is described in

Table 1. Most typical forms of radiation issues come in the form of an SEU which depending on where it occurred may require a power cycle of the device affected. [5]

Table 1.1 Radiation problem types [5]

Issue Type	Description
Total Ionizing Dose (TID)	Material damage caused by ionizing radiation sources. Quantified by deposited energy per mass for a given material with units of Gray (SI) or Rad. Total ionizing does is an accumulated property so the longer something is in a radiation environment, the more TID it will have accumulated. TID damage will eventually cause devices to fail or go out of spec. Most commercial components can withstand 10-20 kRad(Si) (that's 10,000-20,000 Rad(Si)). The (Si) indicates that we are talking about damage in silicon which is important because the same radiation source will not produce the same TID in different materials.
Linear Energy Transfer (LET)	Rate at which energy is deposited in matter as an ionizing particle travels through. Typical units are MeV/cm or scaled by material density as MeV-cm2/mg. You can think of LET as the derivative of TID, although a particles LET is also very important in how likely it is to create different SEE's.
Single Event Effects (SEE)	Disruption in function of electronic circuits due to single ionizing particle interaction. Table 2 contains types of SEE events.

Table 1.2 SEE problem types [5]

SEE Type	Description
Single Event Upset (SEU)	A "bit flip" or change in state of a digital logic element. These are, by definition, reversible. Generally refer to things like memory (SRAM,DRAM) or registers, etc.
Single Event Latchup (SEL)	Turn on of the parasitic SCR inherent in many CMOS processes. Typically will cause an increase in current draw, heating and potential burnout of the device. For "non-destructive" SEL's the only way to clear the SEL is to remove power from the device.
Single Event Functional Interrupt (SEFI)	Very much like an SEU but in an area of logic or memory that causes a reconfiguration of hardware. An example would be lockup of a microprocessor due to an SEU in a register that causes it to reconfigure some internal hardware (like a PLL) in an unrecoverable way.
Single Event Burnout (SEB)	An effect specific to power control devices, specifically MOSFET's. P-channel devices generally believed to be immune to this effect (but not N-channel!!). Voltage derating of a part helps prevent. Additionally a conducting part generally will not undergo SEB - generally happens when the part is "holding off" power.
Single Event Gate Rupture (SEGR)	Shoot-through of a MOSFET's gate dielectric due to charge injection from high energy particles. Generally exhibits itself as a short from gate to source/drain and is catastrophic for the device.

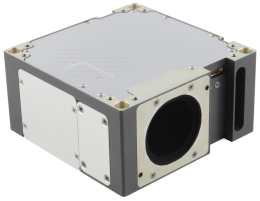
1.4 Past Small Satellite Flight Computer Designs

The idea of small sat in a box is not a new one. Many enterprises are developing integrated small satellites that can be used with a project. One system is from Blue Canyon Tech called the XB1 high performance 1-U cube sat. The system incorporates in parts made in house for GNC pointing and navigation. The overall design is built for high performance. The structure is made so that modules could be stacked for multiple satellite sizes and can be launched from any conventional cubesat deployment system. The computer has SPI, and 6 12bit analog telemetry channels and 6 discrete inputs. The SPI can be used to control experimental devices and the coding is a flight simulator that is built in house as well [7]. The entire system is built in house. This may be difficult for students trying to program their system since it is new unconventional software that they have likely never worked with.

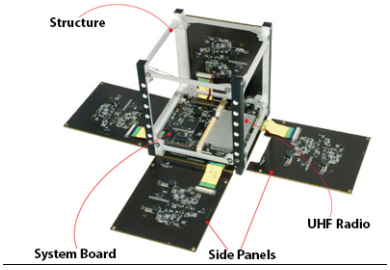
Another system that is on the market today is a Pumpkin product that incorporates a PIC microcontroller and a TI microcontroller with a Pumpkin cube satellite frame. The product is called CubeSat Kit. The coding is done in C code and Pumpkin provides libraries that can be included that incorporate specific spacecraft commands for simplicity. This system is more flexible than Blue Canyons XB1 and is programmed through a very fast and known form of coding. Pumpkin also offers libraries of code that makes it very simple to perform spacecraft specific commands [8]. However with only 2 unprotected controllers, this computation design might not fare well if it were hit by radiation.

One other system that incorporates small satellite in a box is Terran Orbital. The offered options for cube satellite are the intrepid and the endeavor modules. These

designs are a very integrated cube satellite that is already built to spec with power systems and computation systems included. Experimental devices can be included through SPI and other than that every sub system is already built and mounted. The system is even outfitted with an input for a CMOS image sensor like that of a cell phone camera. There is only one computer on board which is the ARM9 based AT91SAM9G20 from Atmel industries [9]. This system is very similar to the XB1 except that it takes more of the freedom of space system design away.



A.



B.

C.

Figure 1.4 A: XB1 integrated satellite designed for stackability. B: Terranorbital Endeavor general satellite body also designed with all systems attached similar to the XB1. C: Pumpkin cubesat kit, outfitted with a Pic microcontroller on a pumpkin frame and libraries that simplify programmability.

1.5 CAN Bus Background

Controller automated network (CAN) is a high-speed inter-computer communication system. It was designed so that all systems in a vehicle could be directly connected to one another and still be independent. However CAN also provides much lower weight in hardware as compared to traditional communication bus techniques. This lower weight came as a side effect due to the simplicity of CAN. Less need for a complicated wire system decreases weight on a vehicle. CAN was created in 1983 by

Robert Bosch and his company when it became evident that traditional inter computer communication techniques were far too slow for the automotive industry to effectively use in automobiles. In 1986 it was presented to the SAE congress in Detroit and in 1993 CAN became an international standard for cars. Today CAN can run at up to 1 Mbit/sec and is used to control all aspects of a vehicle. The abilities of CAN also allow many mechanics and citizens to diagnose problems with their cars. This system has also been used and implemented with many other vehicles including aircraft and boats. CAN is an international standard and all CAN integrated circuits must comply with the Bosch CAN reference model. Since then many international standardization organization codes (ISO) have been outlined regarding CAN specifications. [10]

In satellites inter-computer communication is an important aspect for the avionics and command and data handling subsystems. All satellites have a way of handling serial data flow and commanding between hardware. Small satellite projects from colleges or high schools will usually use I^2C or SPI communication with hardware and one microcontroller or two working in parallel as an error check. Small sat systems like these while cheap are more prone to error. Some current satellites use CAN as a solution to a cheap robust control system. One is the SkySat satellite designed and built by Skybox Imaging. The entire satellite except for the single non-volatile memory is not rad hardened making the satellite far cheaper than an average satellite of its size. The satellite flies through the south Atlantic anomaly approximately 6 times per day and sometimes experiences normal radiation induced problems in the different controllers but because of the modulation the CAN network provides among the error catching algorithms expertly

designed, the controllers are able to notify each other easily when one controller is off nominal and be able to quickly and efficiently restart it or replace it.

2. Theory

2.1 Basic CAN Design

When Robert Bosch set out to invent CAN in 1983 he intended it to be a fast system that did not require a vast network of cabling and wires as most controller networks were. The main idea is that every system can hear and talk to every other system [10]. The idea of CAN can be thought of as similar to a business conference table while traditional Controller networks can be thought of as a game of telephone. When a command is sent from the flight computer to a subsystem it must travel through a chain of other systems before reaching it similar to a boss giving an order to the second in command who passes it to the third and so on to the desired person down the chain. Other systems have direct cable attaching each system that needs to communicate directly to each other. Both of these systems can cause a serious mess of wires that could add excess weight and be too complicated to trouble shoot. Figure 2.1 is a visual representation of what this means.

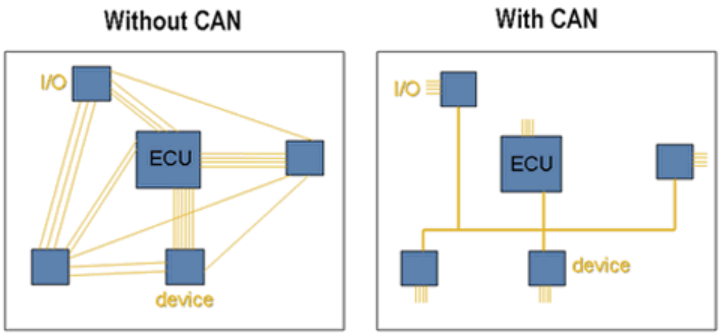


Figure 2.1 CAN networks significantly reduce wiring. [11]

Instead CAN bus operates on a single pair of wires that all controllers are attached to. Every controller can hear what every other controller transmits. The message gets sent directly to every single controller listening. However only the controllers that care about

the message will actually absorb the message, others will see the irrelevance of the message to itself and ignore it. The flight computer can be thought of as the boss at the head of the table while the people around the table are all the chains of command all listening at the same time. If the boss or any subordinate wants to talk to any other subordinate around the table all he needs to do is specify his message for that individual. Communication is direct and universal. And as a side effect results in much less need for complicated wiring. [12]

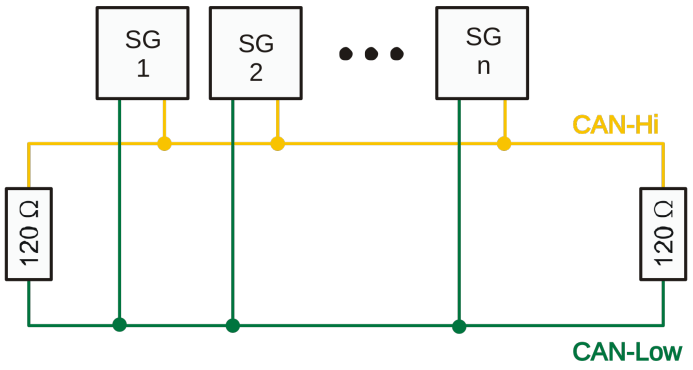


Figure 2.2 CAN Electrical Setup. Very simple wiring is required on a CAN system. The simplicity cuts down on both weight and cost of the system. [13]

2.1.1 CAN Node Configuration

CAN is a differential signal, this means that it works by taking the difference between two different signals, a CAN high (CH) and a CAN low (CL). This means that as long as the difference is clear then the signal can be read. The CAN controllers flip an electronic switch back and forth very quickly to send a message out across the wires. When there is a high logic voltage on the CL there is a low logic voltage or 0 volts on the CH and vice versa. The quick flipping of these two signal wires is what creates the binary CAN frame that carries the message. Although this can be done with a single wire, the use of two wires (CH and CL) creates a more robust system when noise is introduced;

noise omission will be explained in a later section. All controllers that are to be attached to the CAN have CH and CL terminals. All CH terminals are attached to the same node and all CL terminals are attached to the same node. All controllers can see the same message because all of their CH and CL terminals are attached. The CL and CH should always be separate but due to fast moving differential signals across the bus, the ends of the wires can cause reflections as the bus switches between high and low voltages very quickly. To prevent reflections on the bus it is important to put a resistance between the two wires on the far end of both nodes. Typically 120Ω of resistance is placed on either side of the bus wires as a standard. The size of the resistor is not too important as long as reflections are minimized in the wire. [13]

2.1.2 The CAN Frame

The computational mechanism that consists of the traffic in a CAN bus is called a CAN frame. The frame is the binary signal that encapsulates the message, prioritization, address, and error catching algorithms. When a message begins it sends a start of frame bit followed by the 11-bit arbitration field. The definition of arbitrate is to settle a dispute. The arbitration field is meant to settle the dispute between CAN frames sent at the same time. The arbitration field that makes up the highest binary number will win priority of the bus to send data. The other CAN nodes will back down and wait until after that frame is done before trying again. A node knows when a frame is done by observing the end of frame period signified by 7 consecutive bits being held high. The arbitration field also acts as the id or address of the message. A controller will always see every message but it can choose to ignore the message based upon its id or arbitration field. This way priority and destination of a message are inter-linked and have to be considered when designing

the control system of a CAN based spacecraft. Other areas of the CAN frame are the RTR, extension, and error bits such as the CRC field and the ACK field. [13]

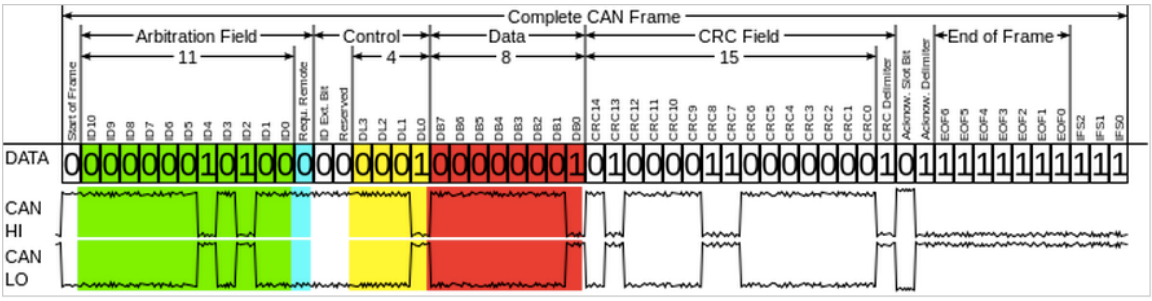


Figure 2.3 Standard CAN message frame. [13]

2.2 Noise Omission

As previously mentioned, CAN is a differential signal. This means that the difference between the two wires of the CAN are always 0 or whatever the logic voltage is, i.e. when the bus is in a dominant state the CL is drawn to ground and the CH is drawn to logic voltage otherwise both nodes sit at rest between zero volts and logic voltage. In most cases 5V is the logic voltage. The CAN controller only cares about the difference between the signals being 5 volts. The required difference can be compromised if a noise is introduced. Noise can be input in a multitude of ways but the problem with long transmission lines is what electromagnetic fields a transmission line can experience when spanning between controllers. Most of the disturbance will come from nearby wires and power sources that emit electric and magnetic fields. As current runs through a wire it produces a magnetic field that can create an electromotive force (EMF) on adjacent wires called electromagnetic interference (EMI) also called electrical cross talk. [14] This cross talk can come in three possible forms: Capacitive coupling, inductive coupling, and RF coupling. In this case RF will not be an issue since very little external radio frequencies typically interfere in a spacecraft. Capacitive coupling induces a current in the adjacent

wire due to fast changing voltage in the CL and CH and inductive coupling induces a voltage in the adjacent wire due to fast changing currents. This is how EMF will induce current and voltage in the adjacent wire that can create electric noise on the bus. Because of this it is important to reduce and manage magnetic flux, the magnetic field passing through the induced wire.

2.2.1 EMF Disturbances

The magnetic field B due to current I through a wire can be described as the current by the EMF emitting wire permeability μ_0 and inversely proportional to the distance from the wire. Therefore the magnetic field B changes with distance from the wire. [15]

$$B = \frac{\mu_0 I}{\Delta l} \tag{1}$$

The magnetic flux Φ_B through an adjacent wire assuming the wire is parallel to the EMF emitting wire (hence perpendicular to the magnetic field) and the magnetic field is uniform throughout the induced wire is defined as the integral of the magnetic field vector dotted with the differential cross sectional area dA of the induced wire.

$$\Phi_B = \int B \cdot dA \tag{2}$$

And the EMF induced in the adjacent wire is just the rate change of the magnetic flux and is equivalent to the resistance of the induced wire by the induced current i.

$$EMF = iR = \frac{\Delta \Phi_B}{\Delta t} \tag{3}$$

$$EMF = -A\frac{dB}{dt} \quad (4)$$

EMF does not always come from another wire running current but the concept is the same wherever the source is. In application of modern electronics, EMF is something that is unavoidable and is a cause of failure even if the conceptual design is flawless. In CAN bus the two transmission wires, CH and CL, depend on a clean difference between the signals but are separate lines. They can experience separate EMF strengths even if placed next to each other. This is true for all transmission lines and is the reason for the twisted pair wire. [16]

2.2.2 Twisted Pair Wiring

In 1881 Alexander Graham Bell invented and patented the twisted-pair wires in the midst of the invention of the telephone in response to the noise induced when operating the telephone.

The reason for this in a quote by Alexander Graham Bell:

"The several circuits are composed each of two wires, a direct and a return wire, forming a metallic circuit. Inductive disturbance in the telephone and in other electrical instruments connected with a metallic circuit when the later is placed in the neighborhood of other electrical circuits arises from the unequal inductive effect of the later upon the two wires, for it is obvious that if the direct and return wire were affected equally the current generated in one would neutralize and destroy that created in the other. The

disturbance can be avoided by placing the two wires in the same inductive relation to the disturbing currents, or, other conditions being the same, by placing them at equal distance from said circuits." –Alexander Graham Bell [14]

By placing the two transmission wires at equal distances from all sources of EMF it will be possible to not avoid or reduce the induced current but to create an equal disturbance in both wires. By having both wires take up the same spatial area they will experience the same disturbance and be able to cancel out each others noise in difference. The answer is to simply twist the wires about one another in equal loops that will receive equal noise from all directions.

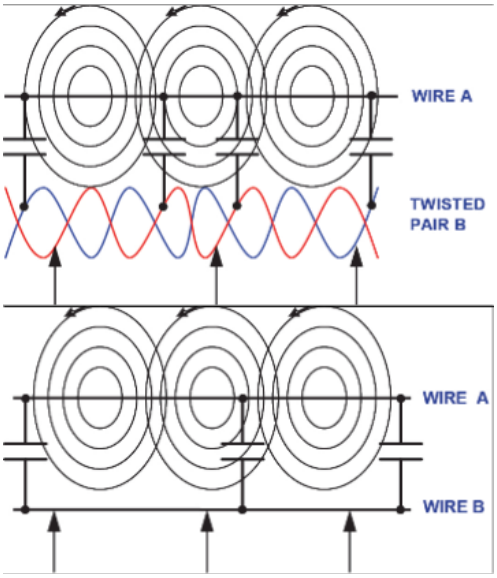


Figure 2.4 RFI and EFI signals due to EMF. The top shows equal induction on the twisted pair. The bottom shows normal induction on an untwisted wire. [14]

This concept has been adapted and used in nearly all electronics since it patent in 1881. It is the cheapest and simplest way to reduce noise in modern electronics. [14]

2.3 Error Catching

The CAN system has a lot of methods for finding and catching errors in both the hardware and the transmission/receiving methods. In the CAN frame there are two fields

that pertain to catching CAN errors, the CRC field and the ACK field. The CRC is a cyclic redundancy check also known as a check sum. In a CRC the total number of high bits from everything preceding the field is added up to a value and placed in the CRC field. The receiving node will take the same message and find the CRC using the same algorithm. If the receiving and transmitting numbers do not match then there has been a problem while transmitting. CAN uses a 16bit CRC but CRCs can be done in multiple bit sizes. For can a 16bit polynomial is divided among the entire bit length until a 16 digit remainder is left. This is the CRC that has encoded the CAN frame. Below is a work through of finding the CRC3 of a random data frame.

The 3rd degree polynomial that will be used:

$$1x^3 + 0x^2 + 1x + 1 \rightarrow 1011 \tag{5}$$

The message bits that will be encoded:

b11010011101100

Now the message will be bit divided using XOR by the polynomial. With XOR 1 and 1 and 0 and 0 equals 0. However 0 and 1 will equal 1. Also a 3 bit remainder will be used since this is CRC3.

```
11010011101100 000
1011
01100011101100 000
 1011
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000
       1011
```
(6)

```
00000000110100 000
    1011
-----
00000000011000 000
     1011
-----
00000000001110 000
      1011
-----
00000000000101 000
       101 1
-----
00000000000000 100
```

Once the quotient is 0 the process is done and the CRC3 is 100. This code can be used to check if there have been any bit errors afterwards by setting the remainder last 3 bits to 100 instead of 000 and repeating the process. If the remainder is 0 then there have been no bit errors. The longer the CRC is the more likely that the CRC will be able to catch an error if one exists.

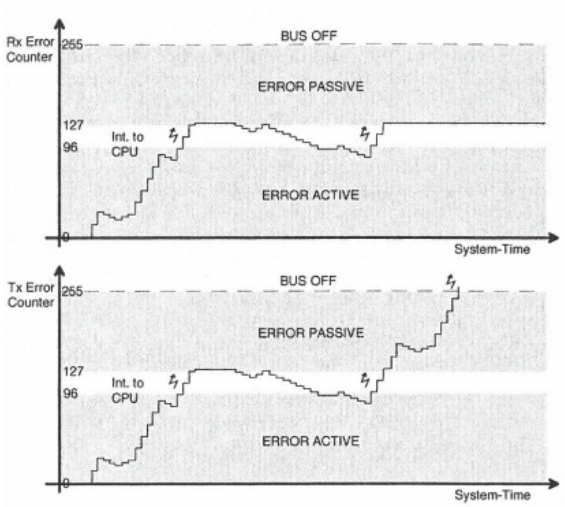


Figure 2.5. CAN Error Counters [13]

The ACK field is just a field that will be a one or a zero depending on if any other node has heard the message. If a dominant bit was not received from any other receiving node then there has either been an error with the received message or no nodes on the CAN were actually listening or exist. Other errors pertain to the hardware itself and will fault if there are errors that occur in the node itself. These errors are called TEC and REC errors, which stand for transmitting/receiving error counter. These errors have different levels of severity and if an error count becomes too high then interrupts will be generated which will shutdown all activity on that particular node. Usually the only way to recover from such errors is by power cycling the hardware [13]. The REC and TEC will increase if faults are detected in either transmission or reception and there are rules that are followed that govern this. They can be found in Appendix C. If the REC or TEC is between 0 and 127 the node is in a state of error active. Error active means that the node can be active on the bus and will throw an active error if an error is detected and nominal operation will occur. If either is above 127 and TEC is below 255 then the node is considered error passive. The node can still communicate on the bus but will throw only passive errors and the node will delay transmissions on the bus. If the TEC rises above 255 then the node moves to a state of Bus Off. This means that it can no longer participate in communication on the bus. In this

case the node needs to be power cycled or commanded into Normal Mode to zero out all REC and TEC values. This is shown in figure 2.5. [13]

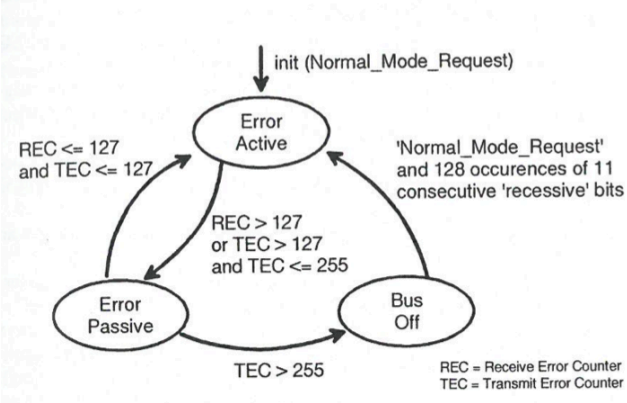


Figure 2.6. CAN Error States [13]

2.4 Basic CAN Operation

Setting up a CAN network is very simple. Basic commands and C code libraries already exist that simplify the use of a node. When initializing a CAN system the speed of the transmission must be the first thing to occur. CAN is capable of transmitting at up to 1 megabit per second but not all controllers on the market are up to the task. The clock or crystal oscillator on the CAN must be similar speeds but must at least be able to operate on the same or faster speed than the CAN bus. The CAN can be configured for lower speeds if necessary but computational speed is important when operating a vehicle network. Other initialization parameters involve setting the mode of the device to normal operation. This involves simply setting a few specific bits in the node and making sure that it is always in normal mode. When receiving messages code can be set up to always check for a pending message and if one is found it needs to be received to any available buffer in the node. Where it can then be read and evaluated if it's id is really one the particular controller cares about. Transmitting involves defining the id and other frame

parameters. The message at most can contain 8 bytes of data and each byte must be defined individually. Once the message has been successfully created all that remains is to simply send the message. [13]

2.5 System Clock and Advanced Message Prioritization

An option sometimes used in CAN systems with high amounts of traffic is setting a system clock. This is one of a few system designs where there is a master controller included. This controller can send out a start of session message that will tell all the other controllers what kind of messages can be sent, primarily aimed at the priority of a message. This is essentially splitting up available time on the bus into timed sections that can be used for different priorities of messages. CAN is already designed to allow for higher priority messages to get through first but if a system has a very high amount of traffic with a lot of very high priority messages then the lower priority messages may get pushed back continuously until there is time to send it. Delaying any message for too long can be bad for the system and may lead to the permanent loss of a message. For example if there is an error in the payload system, critical navigation telemetry may not get through because it is not as important as the payload error data being transferred. Therefore setting allotted time for different priority messages may be necessary. However doing this means taking away a part of the freedom and most useful abilities of the CAN system. Priority is already taken into account and setting a system clock would only slow down transmission of all messages. Doing this means creating caches of messages on each controller so that the messages can be stored until the time to send those priority messages arrives. This adds complication to all of the subsystems on the CAN system. A

smaller satellite may not require a lot of traffic to be sent around so it may be more beneficial to not use a system clock. This is one design parameter that must be carefully considered and if necessary implemented. [13]

2.6 Subsystem Hardware

A satellite's hardware like reaction wheels and gyros are all similar in the fact that they are hardware that can talk through an SPI or I^2C relationship with a controller. Most products on the market have similar startup and shutdown procedures and provide telemetry and receive commands for control and configuration. Therefore it may be possible to generalize the procedures that control any piece of hardware. [17] [18] This is an important aspect to the overall system and will be the backbone of the satellite subsystem architecture. Digging through data sheets to understand basic system operations and commands will become important for designing specific commands for configuration and control. Also CAN IDs need to be created for each controller individually that reflect message priority and priority of a certain system. No two IDs can be used to send different messages so in some cases it may be necessary to have generic messages that could emanate from any system carrying the same message. [12]

Since some systems critically depend on certain kinds of telemetry it will become critical to design an ID allocation protocol based on message importance and what kind of systems need to have the most fluid communication. For example the navigation system would need to have a strong communication with the attitude control system for critical pointing efficiency and accuracy since the control algorithms will depend on each other. The systems that need to be taken into account in a basic cube satellite would be

the basic subsystems defined by SMAD. Below is an example of ID allocations for the CANopen integrated system. [13]

Index (hex)	Object
0000	*not used*
0001 – 001F	Static Data Types (for reference only)
0020 – 003F	Complex Data Types (common to all devices)
0040 – 005F	Manufacturer Specific Data Types (device specific)
0060 – 1FFF	*reserved*
2000 – 5FFF	Manufacturer Specific Profile Area
6000 – 9FFF	Standardised Profile Area
A000 – FFFF	*reserved*

Figure 2.7. CAN open Object Dictionary Structure. Example list of ID allocations for the CAN open system. All ranges are in hexidecimal [13]

The end result of this project is supposed to have the ability to attach and control any hardware directly through any of the CAN controller nodes. Dynamic control theory combined with CAN bus operation will be required in all subsystems. For example when the attitude control controller receives navigation information from the navigation controller it needs to be able to decipher the information and make it useful. This is where dynamic control solutions come into play. Since the hardware for this system is supposed to be able to be moved into any desired position multiple types of dynamics system setups need to be considered as possibilities and checks on controllability of the system designed

Table 2.1. Considered allocation of ids for Spacecraft CAN bus.

Index (Hex)	Index (Dec.)	Object
000-00C	0-12	Hardware Ping
00D-1F4	13-500	Developer Telemetry
1F4-3E8	501-1000	Telemetry
3E9-5DC	1001-1500	Developer Commanding
5DD-7D0	1501-2000	Warnings
7D1-BB8	2001-3000	Faults
BB9-DAC	3001-3500	Navigation Data
DAD-FFF	3501-4095	ACS Commanding

ensured through checks in ACS system's code. The CAN index ids that could be used for this project could look something like in table 2.1.

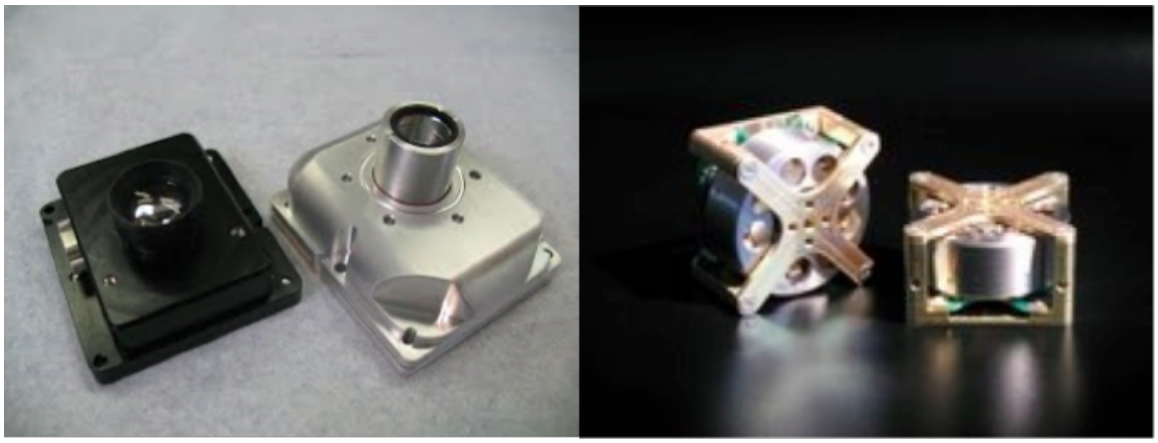


Figure 2.8 Sinclair Star trackers and Reaction wheels similar to what is flown on SkyBox's SkySat's. These devices contain many sensors for health and safety which is a desirable trait to have in a device.

Figure 2.8 shows examples of hardware that can be used in a spacecraft attitude determination and navigation subsystems. Hardware like this is usually designed by third party companies and is already designed with all algorithms necessary to operate. As mentioned previously they will usually have an SPI, I^2C, or some form of serial communication (ex. RS485, RS422, RS232) method of communication with the controller. Therefore it may be desired to create individual PCB boards that will be unique to each piece of hardware. For example a reaction wheel board can have the reaction wheel directly designed into the board, which would decrease required board size since excess connections would not be needed.

2.7 Subsystem Redundancy

Besides the primary function of the flight computers subsystem setup there should be algorithms that catch errors and respond with an appropriate resolution to right the

issue. Already CAN is designed to catch errors in transmission but other techniques must also be used to correct faulty hardware or a radiation hit. Typically the correct response to a radiation hit or an out of sync piece of hardware is to do a power cycle, which will clear out the memory of the system and may possibly fix the issue. If it persists it may have an uncorrectable error. In this case it must either use a radiation resistant Non-volatile memory to reload its own code or simply fail over to a duplicate system. Therefore it may be more desirable to have multiples of the same controllers for redundancy in the case of an SEB or SEGR (refer to table 1.2). Other responses that need to be designed for are in the event of communication loss, repetitive system failure even after reset, and safe orientation failsafe. All of these responses need to be automated so that in the event of a major event the health and safety of the satellite can still be ensured.

2.8 Integrated Development Environment

After the final design of the system is built it needs to have the ability for simplistic programming for the first time satellite builder. The best option for making a basic programmer is to base it off of Matlab since the program is already built for heavy mathematics and is capable of solving complex dynamics and optimization problems. However implementing Matlab into the design means integrating a computer with an operating system capable of utilizing it. Integrating this into the CAN might be difficult and an entire project of its own. This is why it might be more desirable to use another microcontroller as the main controller and using a C++ based system. While C++ is much faster it is also much more complicated to use. Also using Matlab would mean that the overall cost of the system could greatly increase since Matlab is very expensive and so is

a micro cpu that could run it. Matlab will first be attempted and if it proves too complicated for the project timeline it will be abandoned for a simpler and cheaper C++ controller. The programmer GUI called an IDE or integrated development environment will need to have functions that can be easily used by the programming engineer to easily use all of the hardware in a useful way. This should come out in libraries of code that will ideally simplify the way spacecraft commands are used in combination with a CAN bus. If a library is made that can operate the CAN bus and devices that use SPI/ I^2C such as gyros, GPS, and most every other device, then it should theoretically be very straight forward to create spacecraft commands integrated with CAN.

3. Method

In the design of the overall system it is important to modulate components. This can make a system robust and easy to trouble shoot when working in either hardware or software. The CAN system is already designed with a robust hardware algorithm of error checks as previously discussed and because individual controllers are used independent of each other, CAN is already a modulated system. However it will be important to modulate the code as well. The method of code modulating and condensing is called libraries. This will be important for a number of reasons that will be further discussed later. Challenges will arise when it comes to designing a PCB with all controllers integrated into multiple electrical boards while maintaining the unique modulation of the CAN network. To facilitate a small satellite weight and size requirement while bringing beneficence will mean sacrificing weight and volume for computational power and efficiency. Before any hardware design can begin a test bed should be created to facilitate the concepts and get a first prototype of the final product working.

Although the system is meant to be a complete modulated spacecraft flight computer, I will only be focusing on the CAN protocol that will be used. If it can be proved that CAN is an efficient way to modulate spacecraft subsystems then this project will be a success.

3.1 Building the Test Bed

Before any hardware is created a test platform should be created that will represent the final design but should be simpler to produce. It will incorporate already existing boards that can supply Arduino with CAN capability. The boards that will be

used are the Arduino Uno rev3 and the Sparkfun CAN bus shield that gives an Arduino the capability to read messages off of a car's CAN bus network. Because this was designed for a car changes are needed to to make it work between other Arduino controllers. This simply consists of providing a 5V power supply and creating custom CAN functions that send and receive messages across the CAN network without the car codes. A basic prototyping breadboard can be used to house the power, ground, CAN high, and CAN low with two terminating 120Ω resistors at either end of the CAN high and CAN low. The power can come from a 5V DC power transformer that plugs into a 9V AC-DC power supply that plugs into standard house AC power. All of the wires connecting the bus will be twisted together to ensure that any noise is cancelled. The breadboard will not provide twists but it is not a large portion of the wire and should have little affect on the signal. This will also prove whether or not the noise over a short distance will have any real affect in the final design and the information from this may end up simplifying the final design as well. The test bed code is a great way to begin learning how to correctly code the CAN commands and bit banging to the CAN network. It will also help in the final library code design.

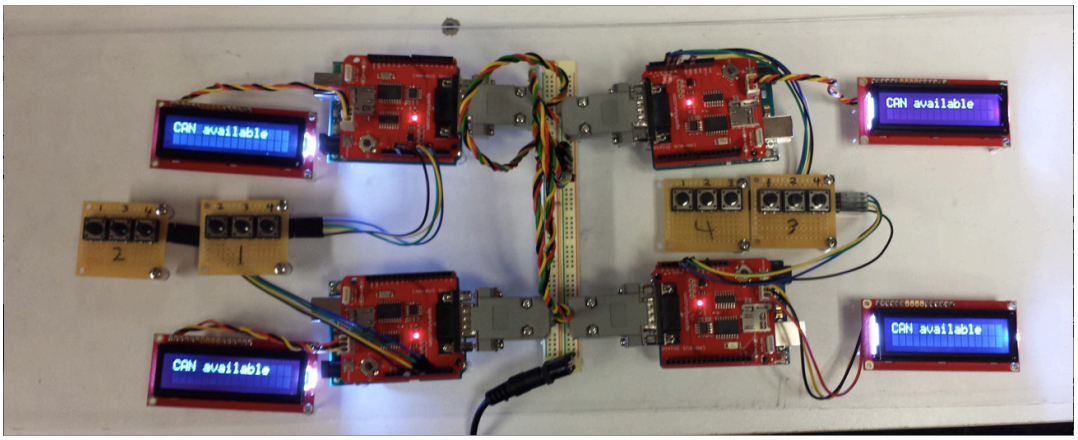


Figure 3.1. Testbed built and programmed

3.2 Coding Methods

Libraries are used in many types of styles of coding and are very useful tools. A library consists of multiple modulated codes called functions. Each function can be called in the main code if the library has been imported. Using these libraries saves space in the microcomputer's memory since the code only needs to be written once but can be called as many times as needed. In C++ code libraries are efficient but also very difficult to produce. It needs to be able to access all of the right source files while keeping many variables out of the hands of the main system; this is for simplicity and safety. Also in C++ many of the operations needed to operate CAN Bus from a coding perspective are not simple to use from a function or a library. This involves an advanced coding concept called pointing.

Pointing as a process is simple but the concept is confusing to many novice programmers. Normally in any code variables are directly handed over from location to location but sometimes when the variable is far too large this is not a simple task. For example the eight-byte array that makes up the entire CAN message must be transferred from the main to the function. Basically the whole point to the library in the first place. Pointing can be equated to sending a letter in the mail vs. and elephant through the mail. If you want to deliver a letter you can just directly send it and the reciever will just go receive it from the mailbox. They don't need to be pointed in the right direction to find their own mailbox. However if you want to send your relative an elephant you need to drop it off in a safe location that's easy for you and the reciever to access. So how does

your neighbor know where to go to retrieve the elephant? You send them a letter pointing them to the correct address to find the elephant. Seems simple but in practice can be rather tricky. In C++ the '&' symbol before a variable will point to the variable's address and the variable will equal to the byte location address and the '*' symbol is called the dereference symbol. This will take an address and convert it back to the value of what ever is stored there. In figure 3.1 myvar is equated to 25 then foo is pointed to the location of myvar while bar is equated to the value of myvar. Notice the '&' before myvar. The value of foo is then the memory location of myvar instead of the value. On the other hand bar holds the value of myvar and not the location.

1	myvar = 25
2	foo = &myvar
3	bar = myvar

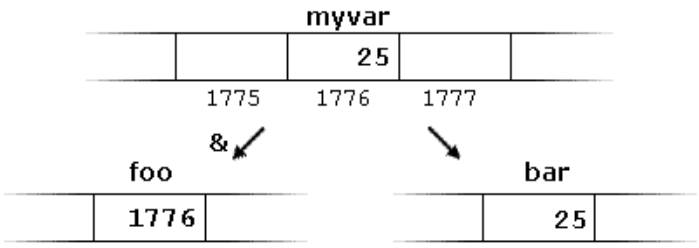


Figure 3.2. In this illustration 'myvar' is equated to 25 then the variable 'foo' is assigned the pointer of 'myvar' at memory location 1776 while 'bar' is assigned the same value as 'myvar'. Showing the difference between equating a variable to a value and a pointer. [19]

In figure 3.2 the variable 'baz' is equated to the reference of foo. This means that it will take the value that is stored at the location indicated by foo. Since the number stored at memory location 1776 is 25, the value of baz will equal to 25. The concepts shown in figure 3.1 and figure 3.2 are vital to getting information in and out of the CAN library. [19]

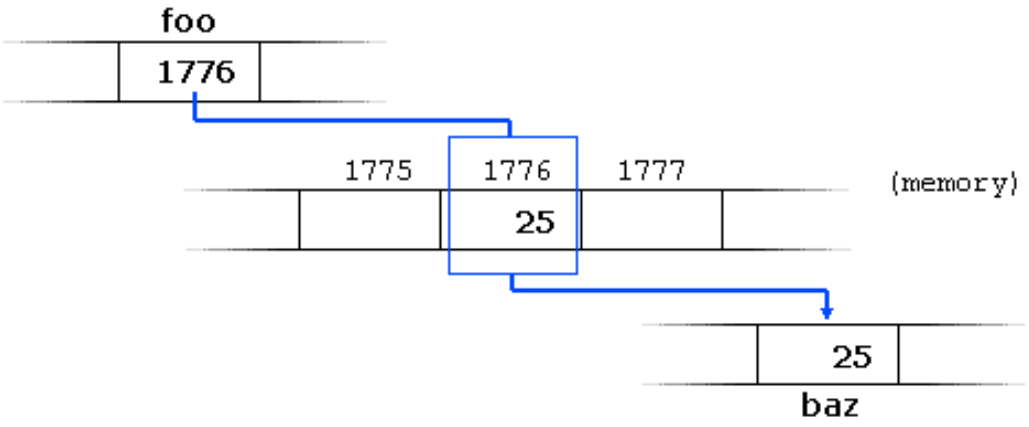


Figure 3.3. The dereference of foo becomes baz. [19]

Most hardware that is used in a spacecraft has many different channels that can be read for different data. Some data is sensor measurement data and some is error and trouble shooting data. All of this data needs to use the CAN library to take data from the hardware and to send the data across the bus. A typical GPS sensor will have over 30 channels that can be read. Some are more important than others and hence need to be read quicker. This concept is called inclusion and will be explained in the next section. All of the channels from all hardware need to be transferred into a CAN message with an ID of importance fitting to the data's importance. As a result each ID will hold telemetry that will provide critical information between systems and to the controllers on the ground that care. A typical small satellite will have less than 100 telemetry points but larger satellites can have thousands of telemetry channels all caring varying levels of important information. Basically the more complex a satellite's hardware, mission, and lifetime the more telemetry channels will be present. Considering this project is intended for a small satellite of minor complexity far fewer channels will likely be present. This will be a great help when designing the complexity of the CAN bus library. This means

that the system can get away with far fewer CAN IDs which means certain functionalities do not need to exist.

Vehicle codes are a concept designed into the CAN2.0 protocol for automobiles across the world and it is a great way of finding system errors and faults. This is another reason why CAN is an inherently great way of designing a spacecraft inter-computer communication bus system. A cars check engine light will turn on if certain CAN IDs come through the main computer. These specific IDs, called Diagnostic Trouble Codes (DTC), will carry information pertaining to an error or fault in the system that needs to be corrected. A mechanic can plug in a CAN decoder and easily find out exactly what is wrong with your car since the message is literally spelling out exactly what the problem is. For example a car can tell you based off the id and the message something like this: “02 Sensor Circuit Low Voltage (Bank I Sensor 2)”. This is essentially the same way the spacecraft can relay errors to operators on the ground. If the ground system designer so chose they could have all codes displayed in a proper sentence text format. However this is a ground system dilemma and is not within the scope of this project. For this project all codes will be transmitted and only some sent to the main display to be spelled out as a concept of functionality. However this does illustrate the ability CAN has to easily trouble shoot.

Another function of the CAN library will be to send information in SPI format. SPI is not as diverse as CAN but is very fast at communicating one on one with hardware. This is why this functionality is still needed. As aforementioned it will be

important to create an inclusion priority scheme. Or how often a telemetry channel will be included when reading off sensor data. This will prioritize which hardware data channels come down quicker than others. SPI will be the communication system that controls individual pieces of hardware (GPS, Gyro, thruster) but a coded algorithm will need to be created to control the inclusion of a certain telemetry point. A bank of priority numbers will likely need to be made with an algorithm that decides which data comes down more often. The algorithm will work by simply making a structure that assigns each telemetry point with a number between 0 to 10 including 0 and 10. The main code can then go through each entry of the structure and grab each point as many times as the number says before moving on to the next entry in the structure. A value of 0 means that the associated telemetry will not be retrieved at all.

3.3 Hardware

The whole point of putting together a CAN system is to make a more organized and efficient way of controlling multiple pieces of hardware simultaneously. However this project will not delve too deep into getting hardware to work well together in a controlled method. The idea is just to prove that hardware working on a CAN bus can be more efficient and add more computational capacity than a conventional microcontroller on its own. If it can be commanded and operated then an SME can design the ACS control algorithms, Telemetry, Tracking and Command (TTC) radios, and other specific subsystems. It is however important to design the backbone of the entire system. This requires iterations of circuit design that will integrate all of the controllers and possible

hardware power and data connections that would be required to customize the control laws and hardware complexity.

A printed circuit board (PCB) is a fiberglass board with layers of copper that act as the conductive paths. Covering the copper layer is a thin solder mask that insulates the copper and gives the board its colored look; it is usually a green look. A silkscreen is applied over the top and bottom of the board with words, logos, and symbols that help understand the board and component orientation. PCB first came around in the 1930's but became more relevant as circuitry moved away from vacuum tubes and relays. Modern PCB was designed as a result of the Silicon Valley semiconductor emergence and simplified a complex system of wires that made up early circuit boards (Figure 3.4) [20]. Due to the complexity and mess of the wires these early boards usually failed when wire insulation wore out and cracked resulting in shorts and trouble shooting problems became a nightmare. Modern circuit boards can contain up to 16 layers of copper, which means that many different wire layouts can be organized inside of the board without the mess or the chance of failure [20].

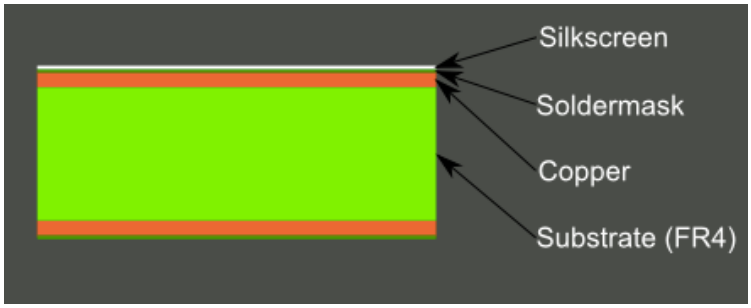


Figure 3.4. Basic PCB layering

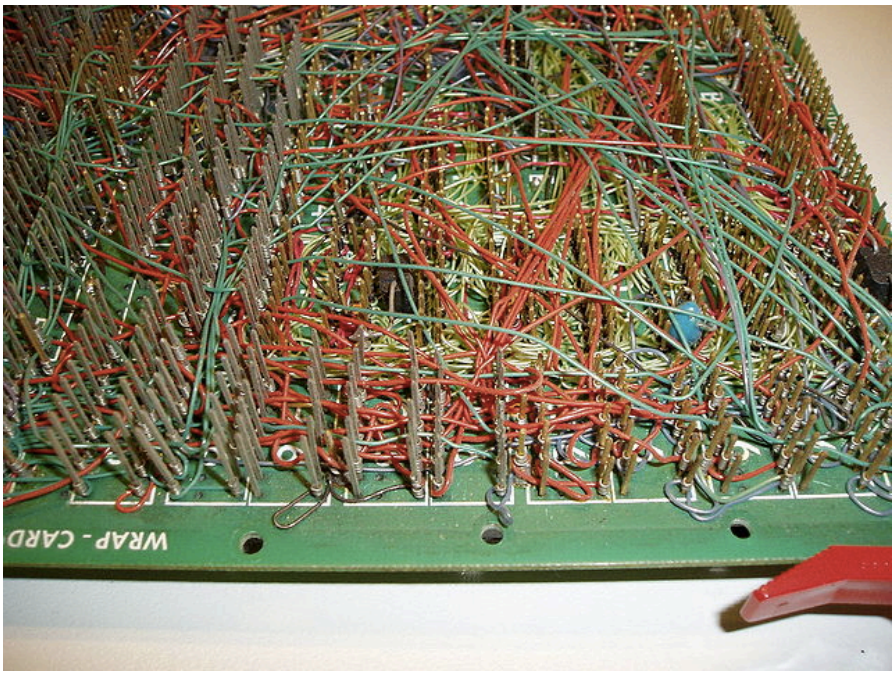


Figure 3.5. Early circuit board design was made up of a complex network of wires that resulted in shorts and headache inducing failures.

The hardware that will be required for the spacecraft flight computer will all piggy back off of a circuit board which is why creating a compact PCB design will be important to completing the project. Many programs exist that are designed specifically for the design of PCB circuit boards of multi-copper-layer boards. The one I will use in particular will be Eagle CAD. The limiting factor in the free version of Eagle is that the number of copper layers allowed is only 2. It is very difficult to design a compact PCB with only 2 layers of copper but if it can be done the cost will be much lower and the weight marginally decreased. With this software the size and shape of the board can be chosen. This will be important when trying to fit the overall design into a basic cube satellite frame. The goal is to be able to fit 4 - 5 CAN controllers inside of a frame so the size should be well under 10 centimeters. Eagle has libraries of premade components that already exist. Each component contains a symbol and connections designed by the library designer and a foot print file that contains the exact dimensions and copper shapes needed

to mount the physical part. For this project I will be using the Sparkfun eagle library, which contains parts that can be purchased on the Sparkfun site. When designing a PCB a board file and schematic file will be created and will connect automatically. When components on the schematic file are added or subtracted the board file will automatically add the footprint to its file. When connections between components are made the board file will create a connection between the footprints of each component as well. Figure 3.5 below shows an example of a schematic and board file. These files can then be sent to a manufacturer to be printed.

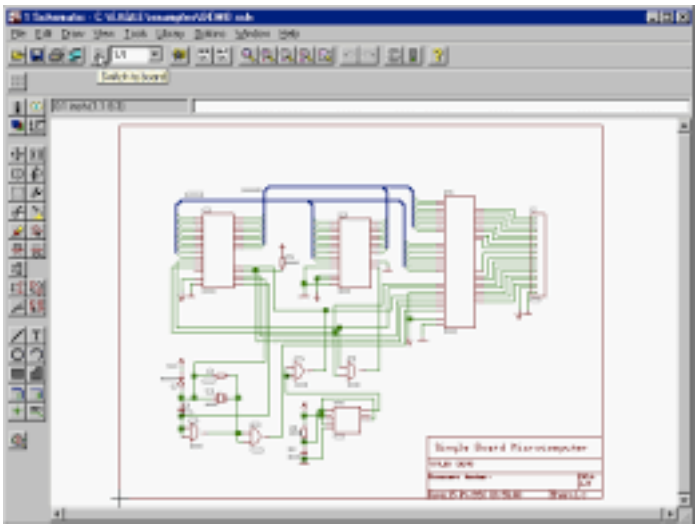


Figure 3.6. Circuit board designers can take a schematic and make an optimized PCB layout then later customized to fit the designers' needs.

3.3.1 Circuit Board Design Techniques

Designing a circuit board is a very difficult process that involves organization of components, wires, and physical space on the board. When putting together a PCB, guidelines should be followed to ensure that the interactions of wires and components proximity to each other do not affect the final electrical system. It can also ease the pain of trying to strategically place parts that make it easy to wire, easy to solder, and easy to

access ports after soldering. The following points have been will help shorten the process and simplify the final design. These are taken from a list of 10 points but only a few should be needed for the complexity of this project

- **Keep trace lengths as short and direct as possible:** This applies particularly in analog and high-speed digital circuitry where impedance and parasitic effects will always play a part in limiting your system performance.
- **Use power planes for distribution of power and ground:** Using pours on the power plane is a quick and easy option in most PCB design software. It applies plenty of copper to common connections and helps ensure power flows as effectively as possible with minimal impedance or voltage drop, and that ground return paths are adequate. If possible, run multiple supply lines in the same area of the board and remember that if the ground plane is run over a large section of one layer, it can have a positive impact on cross-talk between lines running above it on an adjacent layer.
- **Group related components:** Grouping components that are directly connected will help keep a board organized and help shorten wire lengths in a previous rule.
- **Replicating the board you need several times on a larger board:** Start by laying out the board as one panel. Then, after your design rules have been corrected, do your best to step and repeat your design multiple times within the preferred panel size.
- **Design rule check:** Check as you go. The DRC function on PCB software takes a little time, but checking as you go can save hours on more complex designs.

- **Use capacitors everywhere:** Decoupling power lines is good practice. Use capacitors between power and ground wherever possible.

All of the rules listed will be considered when designing the circuit boards. [21]

3.3.2 Creating the Bus

After the PCBs are made they need a bus to make all of the connections between controllers. As mentioned before the best way of doing this is to create a twisted-pair wire for noise omission. The wires that need to be distributed and interconnect all components in the bus are power, ground, CAN high, and CAN low. All four of these wires need to be included in the bus twisted pair wiring. To do this I will use a slightly larger gauge wire for all four connections and use a drill to create the twist. A longer than needed twisted wire should be created in case of mistakes. Afterwards I will tape the wires into a single clean section. The 4 twisted wires can then be cut and divided into separate sections to create each leg that will connect CAN controller nodes. The sections will need to be spliced into the sides of the main bus section. This process may cause mess and result in noise due to poor connections but it is necessary. The terminating resistors of 120Ω need to be soldered between the ends of the CH and CL lines for signal reflection and the ends of the power and ground sealed off. Instead of capping off the power and ground, capacitors between the ends of power and ground may increase signal integrity. Finally DB9 connection headers can be attached to the ends of each leg to make the final connections needed between the controllers and the bus.

3.4 Hardware to PCB Integration

To build the circuit boards a very close eye on the electronics will be required. Most of the components and the schematic are already decided due to the testbed but further action may be required to ensure that the stitching of circuits will work properly. Many issues can be present when switching from a wire layout to a single integrated PCB like an unexpected inductance due to the lengthening or shortening of wires, which can result in lagging of signal and/or voltage build ups. Both of these problems can affect the signal integrity of the CAN bus or result in over voltages that may be potentially harmful to certain components. However for the most part the schematics of the multiple already existing components being used in the testbed can simple be stitched together. One way to resolve signal issues on a board is to incorporate a low-pass filter as explained in section 2 of this report. Capacitors can absorb excess signal and reduce noise. Without adding a lot of excess weight.

3.5 Signal Integrity

As mentioned earlier the difference between the signals is what is important when operating CAN. An important analysis to consider after construction of the system is the integrity of the signal when under nominal operations. This kind of testing should be done by using an oscilloscope to analyze the signal over the two transmission wires, CH and CL, and taking the difference between the signals. If the difference is clean then there should be no problems with receiving on the bus. Figure 3.6 is an example of what it should look like to test signal integrity.

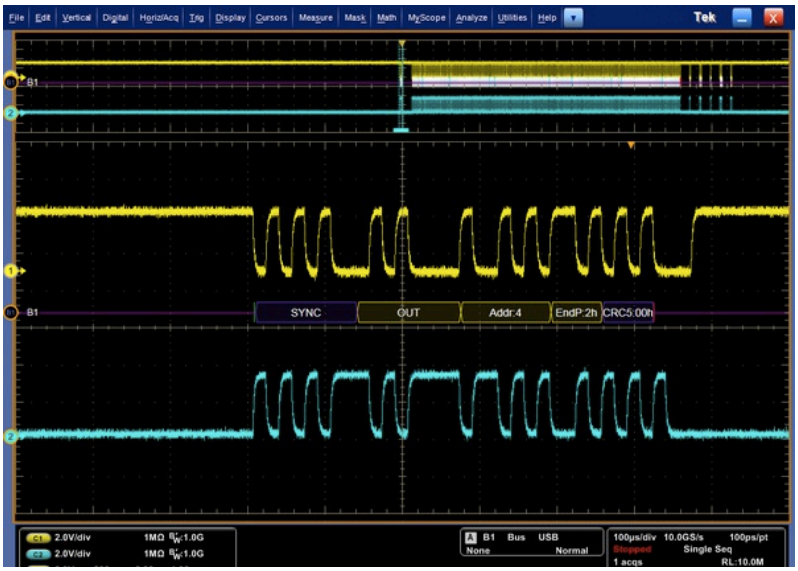


Figure 3.7. Signal integrity of a CAN frame. Although noise is seen it can be seen that it is evenly cancelled out by the difference.

4. Design and Testing

The actual design and implementation of the hardware and integration of the software proved to be a challenge. The free Eagle software was limiting and made creation of the board difficult. Some parts had to be excluded for simplicity of the first design and problems with the programming process were experienced. Fortunately a work around was possible that allowed the first iteration of the final design to be functional.

4.1 Test Bed Final Design and Results

After the test bed was put together as described in the methods, it proved difficult to get functioning. Many issues seemed to plague the system and nothing was being sent. After debugging and looking into the component data sheets it was discovered that it was receiving thousands of REC and TEC errors which shut down the transmitting node. Looking into the issues for this resulted in a large amount of possible outcomes. These outcomes can be seen in Appendix C. After testing multiple scenarios the errors continued to persist. However after more careful inspection of the CAN shield schematic, it was realized that the input to the board was from the car battery originally which was a 12V source. The 12V was not stepped down at all in the schematic but was being fed directly to the Arduino Uno rev3 board. After closer inspection at the Arduino Uno rev3 schematic, it was discovered that there was a step down that took an input of 7-12V DC and converted it to 5V for the Atmega328 controller and CAN controller to use. Since I was stepping down the 9V to 5V I simply just bypassed the 5V transformer. This completely solved the problem. The lesson learned here is to always have a complete

understanding of the schematics and the component data sheets before modifying circuitry. Despite the set back the test bed and code worked and was ready to be transformed into a reduced and simplified form.

4.2 Final Circuit Design

After multiple iterations the final board designs were completed. Two boards were designed. Based off of testing and validation techniques. First the programmer board was completed.

4.2.1 Programmer Requirements

The programmer board was only required to interface with the computer and the microcontroller. It is a tool to program the microcontroller and does not need to be included in the final spacecraft flight computer. This is done to save both size and weight on the final design. Since this will not be included as mass in the final design there are no size or weight requirements on this board. The only requirement is functionality.

4.2.2 Programmer Design

Given the requirements the design of the programmer was straightforward. No excess components were needed, however many components were removed. This was done because many of the parts are considered to be unneeded and are just safety measures that will save weight on the final design of both boards. Since the final design of the programmer has no size or weight requirements this may be a negative design decision but will give a slight benefit when decreasing complexity of the microcontroller board. The portion of the circuit that was removed is designed to protect the board when

given two different power sources simultaneously. One is a 5V from the USB and the other is 7-12V from the input source pin. If a voltage of 7 or greater is supplied to the positive port of the operational amplifier the amplifier will supply a 5V source on the mosfet gate. If a voltage of greater than 4.5 is supplied to the gate then the mosfet will not allow the power from the USB to pass through. However if a voltage of less than 7V is supplied to the input power pin then there will be 3.3 volts on the mosfet gate and the mosfet will allow USB to power the board safely. If there is an input from the source pin the USB power is disabled else the USB source is what powers the board. However without this portion it is possible to damage the programmer or cause a fault on a computer USB port. But if care is taken then the resulting weight decrease will be worthwhile. The circuit portion is shown below in figure 4.1.

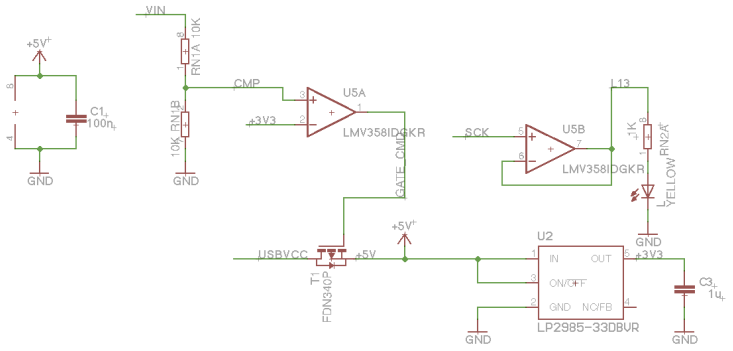


Figure 4.1. Removed power protection circuit. Protects the controllers when power is supplied from both a USB and direct input source of 7-12V.

The final iteration of the programmer board is simple and is highly based off the original Arduino board on purpose. It can be seen in figure 4.2 below.

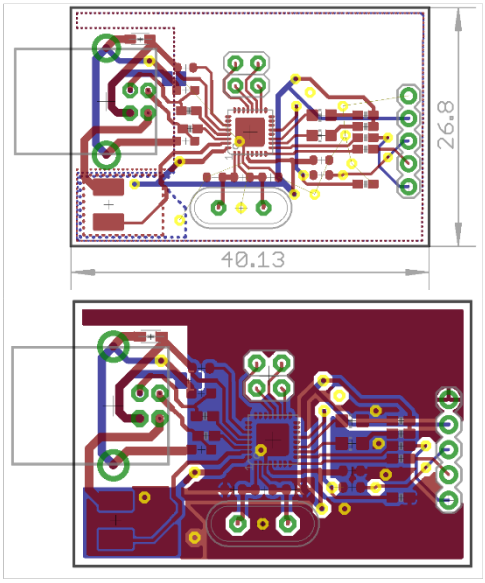


Figure 4.2. Final iteration of the programmer board. The top image is the wire map with dimensions in millimeters. The bottom image is the final version of the board including the top and bottom copper layers.

4.2.3 Microcontroller Board Requirements

The microcontroller board houses the controller that will make up the CAN network nodes and the Atmega328 controller that holds the program that will define the purpose of the unit (ex. RW control law and motors). This board needs to be small enough to fit multiple boards snuggly within a cube satellite frame but large enough house all necessary components. Also the number of components and the size of each component used will dictate how much space and weight is required. For this mostly surface mount components will be used. However the Atmega328 version that will be used will be the DIP (dual-inline-package) to ensure that it will be simple to solder and easy to wire. This is one component that will dictate the size and weight of the final controller. The other option would be the SMD (surface mount device) version of the controller, which would be multiple times smaller and lighter. The comparison can be viewed in figure 4.3. This can be utilized in later versions of the controller but for the

first iteration the DIP will be easier to utilize. Many components that existed on the original Arduino will be excluded in the final design. The header pins that made up connections to every port will be reduced to only include enough to connect very few extra sensors. Therefore two digital pins, two analog pins and the communication pins, RX and TX, will be the only data pins included as headers along with the power pins, reset, 5V, and GND. The only reason for including the power pins is due to the required connection to the programming board. This is also why the pins are organized in the order that they are. This should ensure a clean connection to the other boards. Other pins included are the SPI data connections. The CAN controller is already connected through the controllers SPI connection but some sensors such as GPS, Accelerometer, Gyro, magnetometer, etc. all need SPI connections as well. These connections could have all been excluded on the first version of the board but it may be desirable have the ability to test complex hardware if desired.

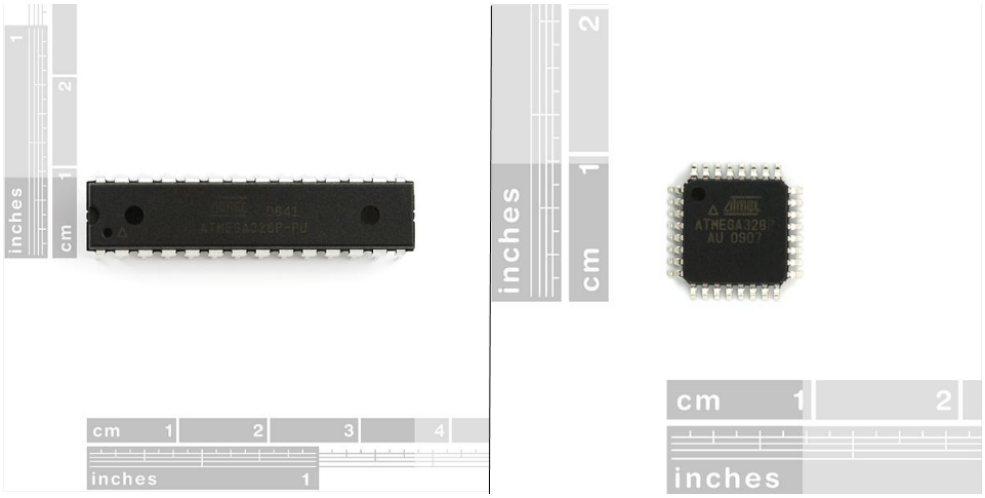


Figure 4.3 The left is the Atmega328 DIP and on the right is the Atmega328 SMD. They are equivalent in capability but differ in size. The SMD has the same width and length as the width of the DIP. The DIP however is easier to solder and can be replaced easily if damaged.

4.2.4 Microcontroller Board Design

When designing the board the guidelines of circuit board design was followed with closer care than with the programmer. This proved to be a very difficult task but in the end a final design was achieved. All spare space on the board used a grounding plane to ensure that a ground source was nearby at all times and the closest paths to ground were observed. The design was first started on a much larger board than desired, 15x15 centimeters. All parts were grouped together according to their relation to other components on the schematic so that wire paths could be kept as short as possible to avoid inductance over longer transmission lines. After the component groups were grouped and placed, the boards were shrunk to allow as much as possible while fitting the components comfortably. Then the challenge became wiring the boards. The first thing that was done was to make a grounding plane over the entire board except on one corner that was left to a 5V plane for the components that supplied board power. Finally the wire connections can be made. The important thing to note about wiring is that wires should be kept short to minimize inductance over long expanses and all wires should avoid making sharp turns. Unless not possible all 90 degree turns should be made with at least a 45 degree portion to ensure signal degradation and power is not lost. Depending on wire current it is possible for electrons to escape the wire when trying to make the tight turns like a car exiting the freeway at high speeds. This process had to be done several times from the 15x15 centimeter board to wiring because the organization of the components was difficult to address. Three revolutions of the board were designed before coming to the final design for the rev1 of this microcontroller CAN board prototype. The final iteration can be seen in figure 4.4. The previous prototypes can be found in appendix B.

Each rev resulted in shorter data and power lines and required unorthodox header pin locations.

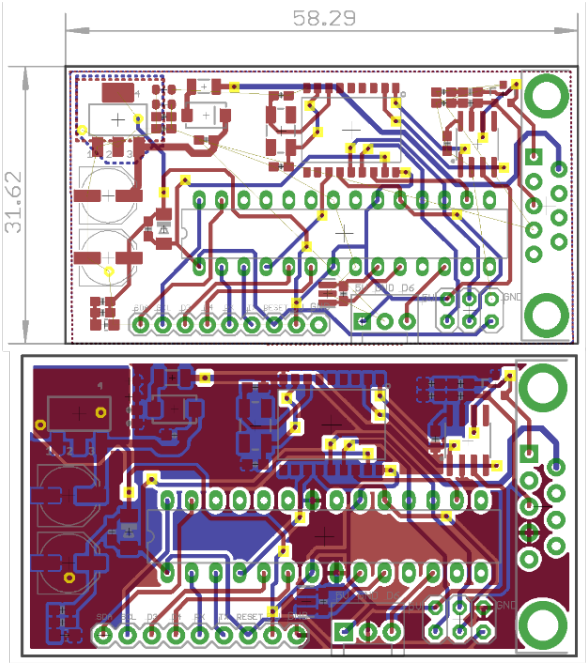


Figure 4.4. Final iteration of the microcontroller board. The top image is the wire map with dimensions in millimeters. The bottom image is the final version of the board including the top and bottom copper layers.

4.3 Validity Concerns

After the first iteration of the board there are concerns with the functionality of the board after design. All because a circuit diagram is accurate does not mean that the board will behave as expected. Due to wiring geometry there may be various reasons for concern regarding how well the prototype will function and how its electrical characteristics will behave. Other concerns involve possible factory presets in SMD components.

4.3.1 Wire Lags and Inductance

Due to the size of the board some lengths of wire had to be extended further than desired. Some lengths of wire work around the outside of the board to move a short distance due to other data/power lines being in the way. This is not desirable but should not affect most of the lines. What is concerning is the connection and location of the crystal oscillator. The crystal oscillator is a chip that literally contains a quarts crystal tuning fork that resonates between two electrodes and creates an electric field as it moves back and forth. In practice a crystal oscillator acts as an RLC circuit. An RLC circuit is a circuit containing a resistor, capacitor and inductor in series as shown in figure 4.5.

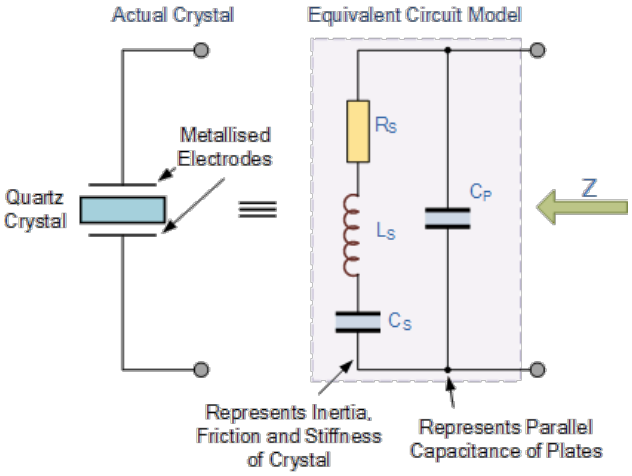


Figure 4.5. A crystal oscillator circuit comparison to an RLC timing circuit. [23]

The capacitor Cp represents the parallel capacitance of the electrodes and the RLC series circuit represents the stiffness and vibrational frequency of the crystal.

The reactance (resistive component in a non resistor component) of each component in the circuit can be described as shown:

$$R_s = R \tag{7}$$

$$X_{L_s} = 2\pi f L_s \tag{8}$$

$$X_{C_s} = \frac{1}{2\pi f C_s} \tag{9}$$

$$X_{C_P} = \frac{1}{2\pi f C_P} \tag{10}$$

Where the total impedance (combined resistance of a system) of the crystal stiffness is given by the following equation.

$$Z_s = \sqrt{R_s^2 + \left(Z_{L_s} + Z_{C_s}\right)^2} \tag{11}$$

And the total impedance of the crystal oscillator is given by the following.

$$Z_P = \frac{Z_s \times X_{C_P}}{Z_s + X_{C_P}} \tag{12}$$

Plotting reactance, figure 4.6, against frequency shows how the impedance acts over frequency and frequency can be deduced from this.

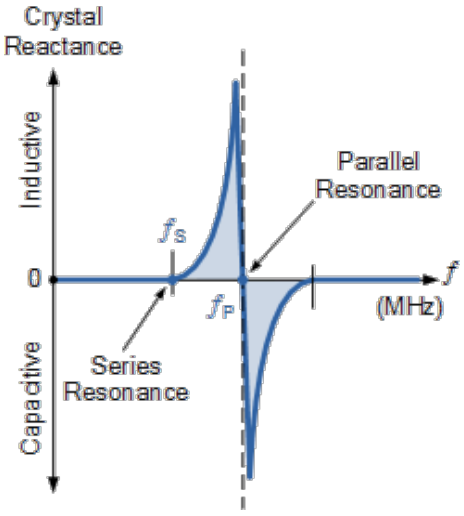


Figure 4.6. Crystal oscillator reactance vs. frequency. Resonance points can be seen at fs and fp. [23]

$$f_s = \frac{1}{2\pi\sqrt{L_s C_P}} \tag{13}$$

Total parallel frequency as,

$$f_P = \frac{1}{2\pi\sqrt{L_s\left(\frac{C_P C_S}{C_P + C_s}\right)}} \tag{14}$$

This shows that given a change in inductance or capacitance over the line can cause the frequency to change drastically and negatively affect the crystal oscillator. If the frequency is changed then the clock on the Atmega328 will not count the proper time increments and will cause errors on the SPI lines that depend on clock accuracy. This is why the crystal oscillator is normally placed as close to the microcontroller chip as possible with even length lines. The crystal oscillator on the

microcontroller board is placed near the clock ports but is at an awkward angle that puts both sides of the oscillator at different line lengths. Since the distance is so small this may not have any affect but it is not ideal and may cause the CAN bus to fail since it is operated through SPI from the microcontroller. [23]

4.3.2 USB Boot Loader

One concern with the final design is the USB boot loader. The final code in the project was functional but was built on an Arduino integrated development environment for simplicity. This meant that the USB boot loader needed to be functional as well. It is possible that the boot loader may not work if the programmer controller (Atmega16U2) is not configured correctly. The information covering this issue is not readily available and may harm the final product. However there is a work around. Since the version of the Atmega328 microcontroller being used is the DIP it will be possible to use an already existing Arduino to program each individual microcontroller and plug it into the DIP socket on the microcontroller board. This can only be discovered after the completion of both boards.

5. Conclusions

CAN bus is an incredible vehicle computer communication and control network that decreases weight and complexity and was designed for exactly that reason on vehicles. Terrestrial vehicles all over the world use it and it is beginning to catch up with the space age. Small satellites built by students usually use a cheap controller that allows for a single microcontroller for total control over the spacecraft. A single controller is usually sufficient for most small satellites but with growing complexity of student projects and the emergence of more daring missions planned for small satellites, the effectiveness and endurance of the onboard flight computer must be enhanced.

The use of CAN bus in a spacecraft has been proved to be effective through satellite missions like SkyBox Imaging's SkySat. However the question becomes: Should CAN bus be used in satellite missions that involve small satellites such as cube satellites? After design of a miniaturized CAN network the answer is dependent on the mission. Duration, required subsystems, and subsystem complexity are the deciding factors. Table 5.1 contains compares relevance of multiple controllers in a CAN network verse a single MCU.

Table 5.1. Multiple Vs. Single MCU

Dependency	Multiple (CAN)	Single (Arduino)
Duration	1 month +	1 month
# of Subsytems	3+	2-3
GNC Required	Yes	No
Prop Required	Yes	No
Radiation Exposure	Preferred	Not Preferred
Satellite Bus Size	3U +	1U

Due to the size of each CAN enabled controller board a 1U cube satellite bus may not be adequate for the size and complexity of the system. This is acceptable since a complex mission using GNC, Prop, or both would likely use a 3U bus or larger anyway. The size of a single Arduino board compared to a CAN controller board can is defined in table 5.2 below.

Table 5.2. Dimension differences between boards.

	CAN PCB	Arduino PCB
Length (mm)	53.34	68.34
Width (mm)	31.62	58.29
Total Square Area (mm^2)	1858.13	3658.06
Area Percent Difference	50.8	

The overall areal comparison between both boards shows that a 50 percent decrease in size was acquired. However one CAN controller board will retain the same

computational capacity as one Arduino. This means that the area to computational capacity was doubled. This can further be improved if the Atmega328 DIP is exchanged for the SMD version.

The final design was completely designed but due to lack of funds and time a complete physical design was not completed. If the final hardware had been completed it would be possible to conduct signal integrity analysis. This would have concluded whether or not the CAN controller boards were designed effectively. If this is the case then the project can be concluded as a success. If not then there is room for improvement and more future work.

The final idea of this design is to create an easy to use, general purpose, satellite bus. Future work to be done is to modify boards for specific functions. A micro reaction wheel can be directly wired into a board. Although this would add size due to the added hardware you would eliminate the need for excess connections and functionality. This could be done for multiple subsystems and effectively make it a plug and play satellite bus where excess controllers can be added by simply plugging in a new controller to the CAN bus. Another improvement would be to replace the Atmega328 DIP for the Atmega328 SMD. If this outcome can be reached then building a small satellite would be similar to building a spacecraft out of Legos and effectively increase all possibilities. Even after this projects conclusion, this goal will still be worked toward outside of the classroom.

Despite the failure of producing final hardware, the outcome of this project is a successful feasibility study and proof of concept of what can be possible if CAN is used in a small satellite. The results of this project can be built upon in the future and work towards increased ability and range for small satellites including possible interplanetary small satellite missions. The modular design can be used by future students to piece together a spacecraft without too much extra complication.

References

[1] Spearman, M. (1985). *Some comparisons of US and USSR aircraft design developments*.*Ntrs.nasa.gov*. Retrieved 4 May 2015, from http://ntrs.nasa.gov/search.jsp?R=1986000673

[2] NASA,. (2015). *NASA Armstrong Fact Sheet: X-29 Advanced Technology Demonstrator Aircraft*. Retrieved 4 May 2015, from http://www.nasa.gov/centers/dryden/news/FactSheets/FS-008-DFRC.html#.VUbsdDvF8gA

[3] Tomayko, J. (1985). NASA's Manned Spacecraft Computers. *IEEE Annals Hist. Comput.*, *7*(1), 7-18. doi:10.1109/mahc.1985.10009

[4] Cpushack.com,. 'CPU History - Computers And Cpus In Space'. N.p., 2015. Web. 2 May 2015.

[5] Dyer, J. (2014, January 1). Background on Radiation Effects. Retrieved February 1, 2015.

[6] NASA,. 'NASA's Van Allen Probes Spot An Impenetrable Barrier In Space'. N.p., 2015. Web. 2 May 2015.

[7] bluecanyontech,. 'Precise 3-Axis Stellar Attitude Determination In A Micro-Package'. N.p., 2015. Web. 2 May 2015.

[8] Cubesatkit.com,. 'Cubesat Kit Home'. N.p., 2015. Web. 2 May 2015.

[9] Terran Orbital Corporation,. 'Intrepid Platform - Terran Orbital Corporation'. N.p., 2015. Web. 2 May 2015.

[10] Can-cia.org,. 'CAN In Automation (Cia): CAN History'. N.p., 2015. Web. 2 May 2015.

[11] http://www.ni.com/white-paper/2732/en/

[12] Can-cia.org,. 'CAN In Automation (Cia): CAN Protocol'. N.p., 2015. Web. 2 May 2015.

[13] Lawrenz, W. (1997). *CAN System Engineering From Theory to Practical Applications*. New York, NY: Springer Science Business Media.

[14] EETimes,. (2015). *Use a twist (and other popular wires) to reduce EMI/RFI | EE Times*. Retrieved 30 November 2015, from http://www.eetimes.com/document.asp?doc_id=1279624

[15] Hyperphysics.phy-astr.gsu.edu,. (2015).*Magnetic fields of currents*. Retrieved 30 November 2015, from http://hyperphysics.phy-astr.gsu.edu/hbase/magnetic/magcur.html

[16] Rpi.edu,. (2015). *Introduction to Magnetism and Induced Currents*. Retrieved 30 November 2015, from http://www.rpi.edu/dept/phys/ScIT/InformationStorage/faraday/magnetism_a.html

[17] Sinclairinterplanetary.com,. 'Reactionwheels - Www'. N.p., 2015. Web. 2 May 2015.

[18] Sinclairinterplanetary.com,. 'Star Trackers - Www'. N.p., 2015. Web. 2 May 2015.

[19] Cplusplus.com,. (2015). *Pointers - C++ Tutorials*. Retrieved 30 November 2015, from http://www.cplusplus.com/doc/tutorial/pointers/

[20] Learn.sparkfun.com,. (2015). *PCB Basics - learn.sparkfun.com*. Retrieved 30 November 2015, from https://learn.sparkfun.com/tutorials/pcb-basics

[21] Robledo, E., & Toth, M. (2015). *Ten best practices of PCB design*. *EDN*. Retrieved 30 November 2015, from http://www.edn.com/electronics-blogs/all-aboard-/4429390/2/Ten-best-practices-of-PCB-design

[22] Storr, W. (2013). *Quartz Crystal Oscillator and Quartz Crystals*. *Basic Electronics Tutorials*. Retrieved 30 November 2015, from http://www.electronics-tutorials.ws/oscillator/crystal.html

Appendix A

The following code is the original code that operated the basic CAN bus without the CAN library interaction.

```
// Libraries included
#include <Canbus.h>
#include <mcp2515.h>
#include <mcp2515_defs.h>
#include <SoftwareSerial.h>


// Attach the serial display's RX line to digital pin 3 and TX to difital pin 6
SoftwareSerial mySerial(3, 6);
// Basic LCD screen control commands
#define COMMAND 0xFE // Begin a command (the following LCD write will be the
        actual command)
#define CLEAR   0x01 // Command to clear screen
#define LINE0   0x80 // Command to move cursor to top line
#define LINE1   0xC0 // Command to move cursor to second/bottom line
#define PWMCOMMAND 0x7C // Begin a PWM command (the following LCD write
        will be the actual PWM command)
#define BRIGHT100 0x9D // PWM command to set backlighting to 100%
#define SPARKFUN 0x09 // PWM command to toggle on/off sparkfun startup screen

// definition for random number
long randnum;

// Controller IDs
#define ID1 0xA // Cotroller 1 ID 10
#define ID2 0x14 // Cotroller 2 ID 20
#define ID3 0x1E // Cotroller 3 ID 30
#define ID4 0x28 // Cotroller 4 ID 40

// 3 pushbuttons for each of the arduinos involved
const byte CLICK1 = A0;
const byte CLICK2 = A1;
const byte CLICK3 = A2;

void setup()
{
  pinMode(CLICK1, INPUT_PULLUP); //Set as input and activate the ATmega328's
        pullup resistor for this pin
  pinMode(CLICK2, INPUT_PULLUP); //Set as input and activate the ATmega328's
        pullup resistor for this pin
```

```
  pinMode(CLICK3, INPUT_PULLUP); //Set as input and activate the ATmega328's
          pullup resistor for this pin
  mySerial.begin(9600); // set up LCD soft serial port for 9600 baud
  clear_lcd();
  mySerial.write(PWMCOMMAND);
  mySerial.write(BRIGHT100);
  //  mySerial.write(PWMCOMMAND);
  //  mySerial.write(SPARKFUN);
  delay(500); // wait for display to boot up
  if (Canbus.init(CANSPEED_500)) /* Initialise MCP2515 CAN controller at the
          specified speed */
  {
   mySerial.print("CAN Init ok");
  }
  else
  {
   mySerial.print("Can't init CAN");
  }

  delay(1000); // Pause for effect...
  //  debug_msg();
}

void loop()
{
  clear_lcd();
  mySerial.write("Hello, Aric"); // Conceited much?
  delay(1000); // Pause for effect...
  clear_lcd();
  mcp2515_bit_modify(CANCTRL, (1 << REQOP2) | (1 << REQOP1) | (1 <<
          REQOP0), 0); // Make very sure CAN starts in normal operation mode
  mySerial.print("CAN available ");
  while (1) { // unnecesary infinite loop but needed for the above conceitedness
   can_rx(); // check for message and read if available
   //      debug_msg();



   if (digitalRead(CLICK1) == 0)
   {
    clear_lcd();
    randnum = random(1001);
    if (can_tx(randnum, ID1) == 1)
    {
     mySerial.print(randnum);
     mySerial.write(COMMAND);
```

```
    mySerial.write(LINE1);
    mySerial.print("TX");
    //        debug_msg();
    delay(1000);
    clear_lcd();
    mySerial.print("CAN available");
   }
   else
   {
    clear_lcd();
    mySerial.print("CAN available");
   }
}



if (digitalRead(CLICK2) == 0)
{
  clear_lcd();
  randnum = random(1001);
  if (can_tx(randnum, ID2) == 1)
  {
    mySerial.print(randnum);
    mySerial.write(COMMAND);
    mySerial.write(LINE1);
    mySerial.print("TX");
    //        debug_msg();
    delay(1000);
    clear_lcd();
    mySerial.print("CAN available");
  }
  else
  {
    clear_lcd();
    mySerial.print("CAN available");
  }
}



if (digitalRead(CLICK3) == 0)
{
  clear_lcd();
  randnum = random(1001);
  if (can_tx(randnum, ID3) == 1)
  {
```

```
        mySerial.print(randnum);
        mySerial.write(COMMAND);
        mySerial.write(LINE1);
        mySerial.print("TX");
        //        debug_msg();
        delay(1000);
        clear_lcd();
        mySerial.print("CAN available");
      }
      else
      {
        clear_lcd();
        mySerial.print("CAN available");
      }
    }


  }
}




void clear_lcd()
{
  mySerial.write(COMMAND);
  mySerial.write(CLEAR);
  mySerial.write(COMMAND);
  mySerial.write(LINE0);
}

//void debug_msg()
//{
//  mySerial.write(COMMAND);
//  mySerial.write(LINE1);
//  //mySerial.print("TX");
////  mySerial.print(mcp2515_read_register(TXB0CTRL));
////  mySerial.print(" ");
//  mySerial.print(mcp2515_read_register(CANINTF));
//  mySerial.print(" ");
////  mySerial.print(mcp2515_read_register(CANINTE));
////  mySerial.print(" ");
//  mySerial.print(mcp2515_read_register(REC));
//  mySerial.print(" ");
//  mySerial.print(mcp2515_read_register(EFLG));
//  mySerial.print(" ");
```

```
//  mySerial.print(mcp2515_read_register(TEC));
//}

void can_rx()
{
  if (mcp2515_check_message())
  {
    //    mySerial.print("message checked");
    //    delay(1000);
    tCAN message;
    int buffer[512];
    if (mcp2515_get_message(&message))
    {
      if (message.id == ID4)
      {
        buffer[0] = message.data[0];
        buffer[1] = message.data[1];
        buffer[2] = message.data[2];
        buffer[3] = message.data[3];
        buffer[4] = message.data[4];
        buffer[5] = message.data[5];
        buffer[6] = message.data[6];
        buffer[7] = message.data[7];
        clear_lcd();
        mySerial.print((buffer[0] << 8) | (buffer[1]));
        mySerial.write(COMMAND);
        mySerial.write(LINE1);
        mySerial.print("RX");
        delay(1000);
        clear_lcd();
        mySerial.print("CAN available");
        /*mySerial.write(COMMAND);
         mySerial.write(LINE1);
         mySerial.print("RX");*/
      }
    }
    else
    {
      mySerial.println("Receive Failed");
    }
  }
}

int can_tx(long randNumber, int ID)
{
  if (mcp2515_check_free_buffer() == false)
```

```
  {
   clear_lcd();
   mySerial.print("No free buffer");
   delay(1000);
   return 0;
  }
  tCAN message;
  message.id = ID;
  message.header.rtr = 0;
  message.header.length = 8;
  message.data[0] = randNumber >> 8;
  message.data[1] = randNumber & 0xFF;
  message.data[2] = 0x00;
  message.data[3] = 0x00;
  message.data[4] = 0x00;
  message.data[5] = 0x00;
  message.data[6] = 0x00;
  message.data[7] = 0x00;

  mcp2515_bit_modify(CANCTRL, (1 << REQOP2) | (1 << REQOP1) | (1 <<
          REQOP0), 0);
  if (mcp2515_send_message(&message))
  {
   /*mySerial.print("1");
    delay(1000);
    clear_lcd();*/
   return 1;
  }
  else
  {
   /*mySerial.print("0");
    delay(1000);
    clear_lcd();*/
   mySerial.print("Transmit failed");
   delay(1000);
   return 0;
  }
}
```

This is the CAN library that was created to facilitate the CAN bus and consolidate memory.

The .h file

```
#ifndef AnubisCommBus__h
#define AnubisCommBus__h
#include "Arduino.h"

class AnubisCommBus
{
  public:
    char CAN_Init(unsigned char speed);
    char SPI_Init();
    unsigned char CAN_CanTx(long ID, uint8_t * Data);
    unsigned char CAN_CanRx(long msg_id);
    unsigned char SPI_SpiTx(uint8_t spi_data);
  private:
    unsigned char _speed;
    long _ID;
    long _msg_id;
};

#endif
```

The .CPP file

```
#include "mcp2515.h"
#include "Arduino.h"
#include "AnubisCommBus.h"

/**************DEFINITIONS*******************/
//#define  P_MOSI     B,3
//#define  P_MISO     B,4
//#define  P_SCK B,5
//#define  MCP2515_CS B,2


/********************************************/

char AnubisCommBus::CAN_Init(unsigned char speed)
{
    _speed = speed;
    return mcp2515_init(_speed);
}

char AnubisCommBus::SPI_Init()
{
    pinMode(SCK, OUTPUT);
```

```
    pinMode(MOSI, OUTPUT);
    pinMode(SS, OUTPUT);

    digitalWrite(SCK, LOW);
    digitalWrite(MOSI, LOW);
    digitalWrite(SS, HIGH);

    SPCR |= _BV(MSTR);
    SPCR |= _BV(SPE);
}

//char AnubisCommBus::SPI_StartTransmission()
//{
//
//}
//
//char AnubisCommBus::SPI_EndTransmission()
//{
//
//}

unsigned char AnubisCommBus::CAN_CanTx(long ID, uint8_t * Data)
// Include pointer here!!!!!!
{
    _ID = ID;
    int num = 0, i;
    if (mcp2515_check_free_buffer() == false)
    {
        boolean tx_buff = false;
        while (tx_buff == false)
        {
            if (num == 2)
            {
                return 0; // If transmit failed return 0
            }
            tx_buff = mcp2515_check_free_buffer();
            num++;
        }
    }
    tCAN message;
    message.id = _ID;
    message.header.rtr = 0;
    message.header.length = 8;
    for(i=0;i<=7;i++)
    {
        message.data[i] = *Data;
        Data++;
    }
    // Ensure CAN controller is in Normal mode
    mcp2515_bit_modify(CANCTRL, (1 << REQOP2) | (1 << REQOP1) |
```

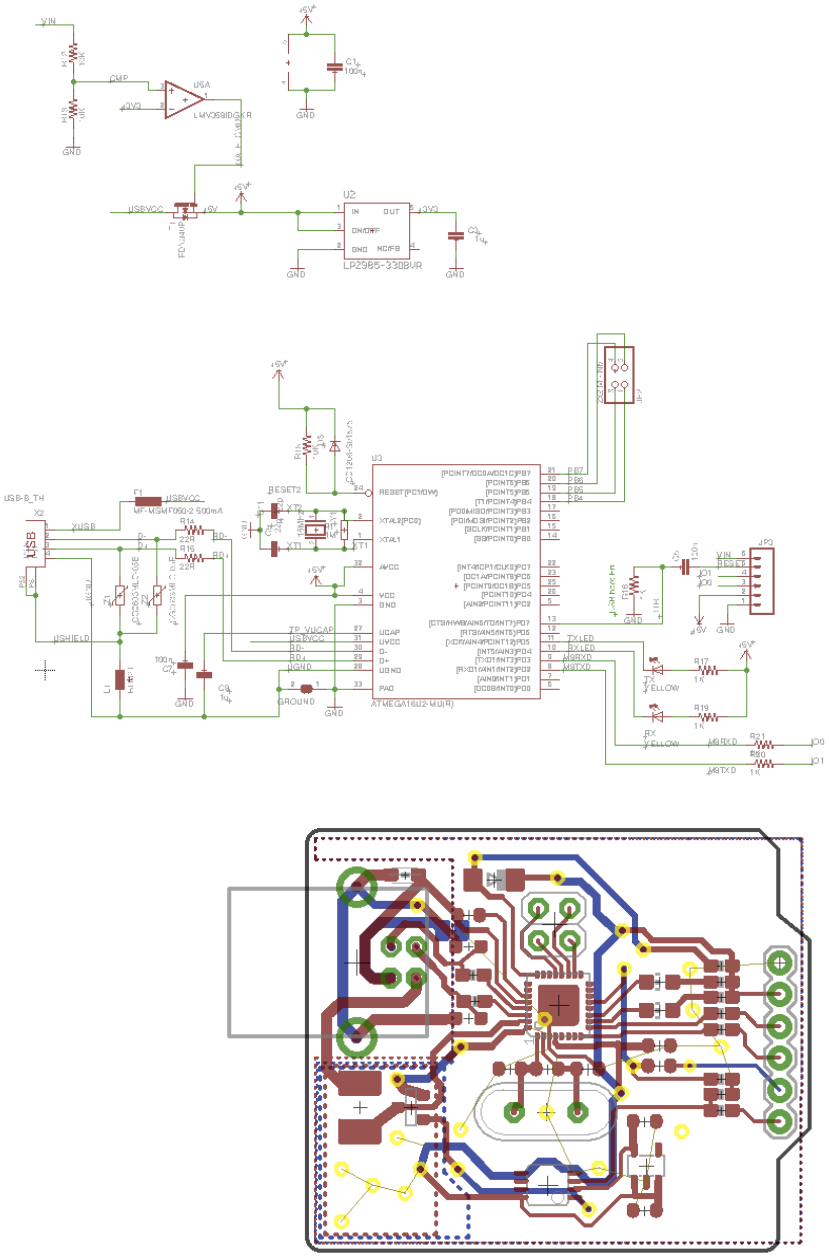
```
(1 << REQOP0), 0);
    if (mcp2515_send_message(&message))
    {
        return 1; // If tramsmit was successful return 1
    }
    else
    {
        int tx_confirmation = 0;
        num = 0;
        while (tx_confirmation == 0)
        {
            if (num == 2)
            {
                return 0; // If transmit failed return 0
            }
            tx_confirmation = mcp2515_send_message(&message);
            num++;
        }
    }
}

unsigned char AnubisCommBus::CAN_CanRx(long msg_id)
{
    _msg_id = msg_id;
    if (mcp2515_check_message())
    {
        tCAN message;
        uint8_t * buffer;
        if (mcp2515_get_message(&message))
        {
            if (message.id == _msg_id)
            {
                for(int i = 0; i <=7; i++)
                {
                *buffer = message.data[i];
                buffer++;
                }
                return 1; // Message available and stored in
buffer
            }
            else
            {
                return 2; // No relevant message currently
available
            }
        }
        else
        {
            return 0; // Error has occurred while retrieving
message
```

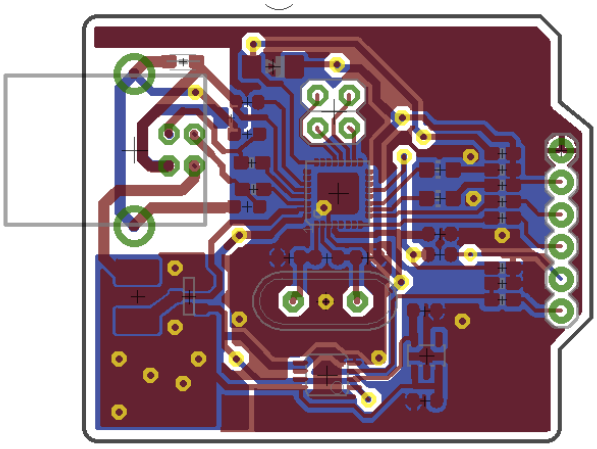
```
        }
    }
    else
    {
        return 2; // No message currently available
    }
}


unsigned char AnubisCommBus::SPI_SpiTx(unsigned char spi_data)
{
    return spi_putc(spi_data);
}
```

Appendix B

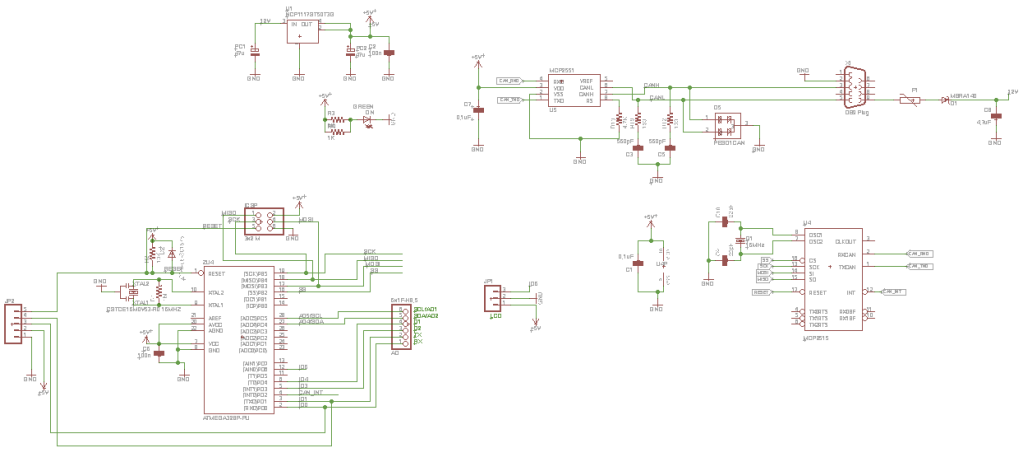
The following are other revisions of the same microcontroller boards before the final revisions were decided upon.

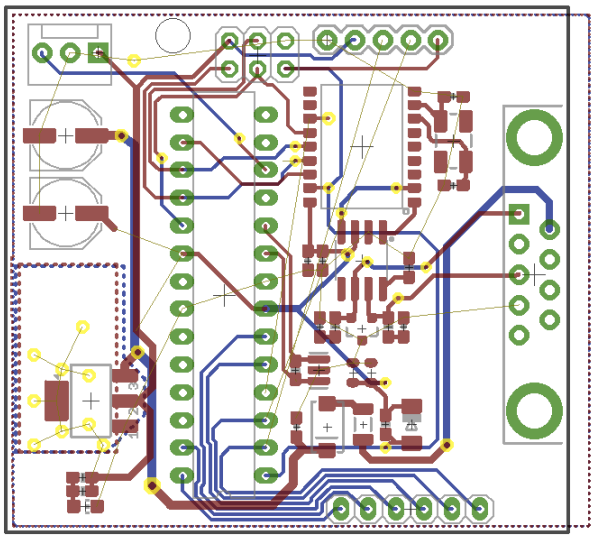
Programmer Revision 1



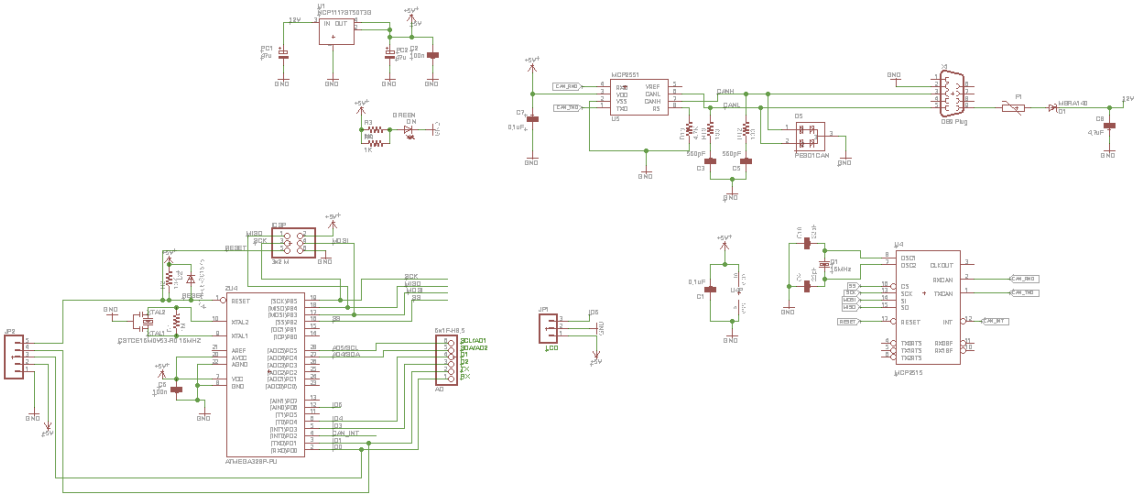


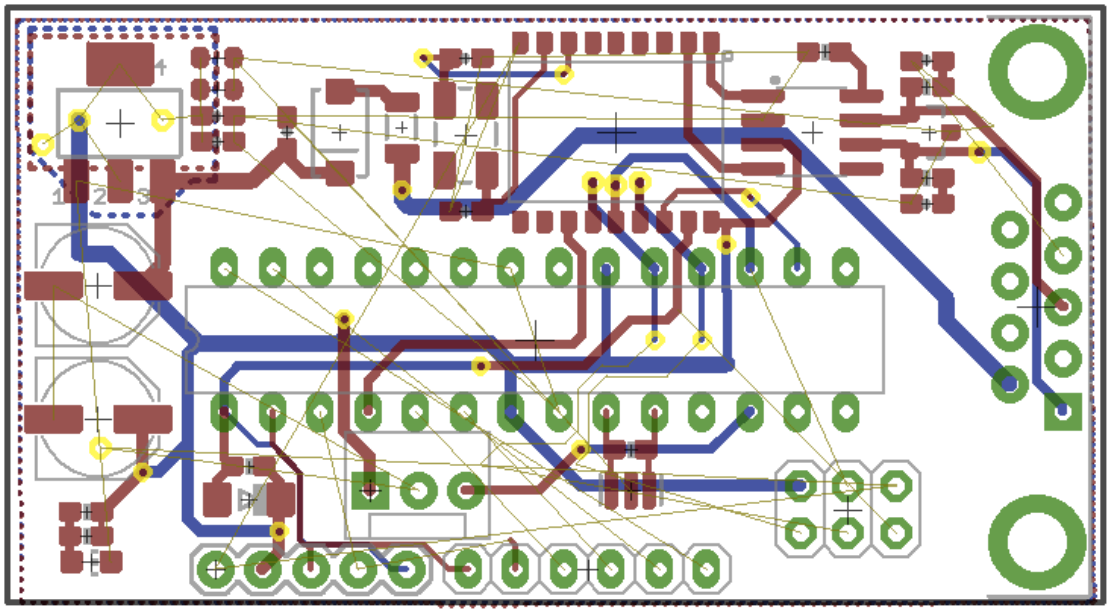
MCU Revision 1





MCU Revision 2





Appendix C

The following is a list of rules that govern the REC and TEC values.[13]

a) When a transceiver detects an error, the REC will be increased by 1, except when the detected error is a bit error during the sending of an active error flag.
b) When a receiver detects a dominant bit as the first bit after sending an error flag, the REC is increased by 8.
c) When a transmitter sends an error flag, the TEC is increased by 8. The TEC remains unchanged, with the following exceptions:
   Exception 1: If the transmitter is error passive and detects an acknowledgment error because of not detecting a dominant ACK slot and does not detect a dominant bit while sending its passive error flag.
   Exception 2: If the transmitter sends an error flag because a stuff error occurred during arbitration, which should have been recessive, and was sent as recessive but monitored as dominant.
d) If a transmitter detects a bit error while sending an active error flag or an overload flag, the TEC is increased by 8.
e) If a receiver detects a bit error while sending an active error flag or an overload flag, the REC is increased by 8.
f) Any node tolerates up to 7 consecutive dominant bits after sending an active error flag, passive error flag, or overload flag. After detecting the 14th consecutive dominant bit (in the case of an active error flag or an overload flag), or after detecting the 8th consecutive dominant bit following the passive error flag, and after each additional sequence of 8 consecutive dominant bits every transmitter increases its TEC by 8 and every receiver increases its REC by 8.
g) After a successful transmission of a frame (obtaining ACK and no error detected until EOF is finished) the TEC is decreased by 1 unless it was already 0.
h) After a successful reception of a frame (reception without error up to the ACK slot and the successful sending of the ACK bit), the REC is decreased by 1 if it was between 1 and 127. If the receive error counter was 0, it remains 0; if it was greater than 127, then it will be set to a value between 119 and 127.