Design and Implementation of the Communications Subsystem for the Cal Poly CP2 Cubesat Project

Chris Noe
<cnoe@calpoly.edu>

Computer Engineering Department
California Polytechnic State University, San Luis Obispo
Submitted: June 11, 2004

Abstract

A primary mission requirement of any satellite is the ability to exchange information with a ground based command station. In this paper, the hardware and software design of the communications subsystem of the Cal Poly CP2 Cubesat is demonstrated. Hardware devices chosen for this design include two redundant Chipcon CC1000 ultra-low power FM radio transceivers, two RF power amplifiers, and two redundant PIC microcontrollers. The software written for the PIC processor is responsible for encoding telemetry and payload information into standard AX.25 format data packets as well as decoding commands transmitted from the ground station. The software can also detect different failure modes such as transceiver failure, main system bus failure and main processor failure and can recover gracefully from these situations. The design and implementation of the hardware and software to meet the stated mission requirements is discussed and problems, solutions and remaining issues are presented in detail.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A primary mission requirement of any satellite is the ability to exchange information with a ground based command station. Such information can include sensor data which detail the operating environment of the satellite, telemetry data which provide the orbital location of the satellite, or commands to instruct the satellite to perform a specific function such as running self-diagnostics.

Implementation of a communications subsystem for a Cubesat – a standard, small form factor satellite – is a difficult engineering problem. With their small size, limited area is available for solar cells and batteries and thus Cubesat operating power budgets are typically very small. In small satellite designs, it is not uncommon for the communications subsystem to be the most power hungry subsystem because of the need to transmit sensor and telemetry data back to earth with as much RF power as possible (in order to increase the ground station's ability to receive the data).

The goal of this paper is to detail the design and implementation of a complete communications subsystem for a Cubesat, using the most recent Cal Poly Cubesat, CP2, as a reference design. The design challenges posed by the small size of the satellite are met head-on and, at the conclusion of the paper, a working communications system prototype is presented.

1.1 Cubesat Project Overview

The Cubesat program is a project birthed and supported by Cal Poly and Stanford which aims to prove the feasibility of a new class of "standardized picosatellites" which can proceed from concept to finished design quickly in order to "reduce cost and development time [and] provide increased accessibility to space" [1].

The short development cycle is made possible by the small mass and simple volume specifications of a Cubesat (discussed in Section 2.1). With a simple, standardized satellite geometry, undergraduate university students have the opportunity to design, build, test, and actually see their satellite function in space, all within the time frame of an average college student (4-5 years).

The Cubesat project has been highly successful in the past few years, having launched multiple Cubesats from many countries including Canada (CANX1 [2]), Japan (CUTE [3], XI-IV [4]), Denmark (DTUSat [5]), the Netherlands (AAUSat [6]) and the United States (NarcisSat [7]).

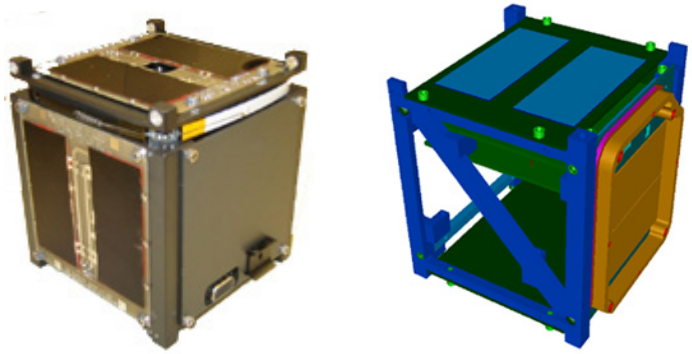
For more information on the Cubesat project, visit the Cubesat project website [1].

1.2 Polysat Project Overview

The Polysat project is a multi-disciplinary team of engineering students responsible for designing, building and qualifying Cubesats at Cal Poly [8]. Cubesats are an ideal student project for a school like Cal Poly, where emphasis is put on the "learn-by-doing" process, because they offer students a chance to participate in a project with hard deadlines, real world problems, and measurable product. Also, Cubesat design is a multi-disciplinary activity which gives students the chance to work within a team of people from other engineering disciplines, an experience which is key to future success in industry.

Cal Poly has designed and built one Cubesat, CP1, and is in the final design stages of a second Cubesat, CP2 (Fig. 1.1). The concept driving both CP1 and CP2 is to provide an educational experience in designing a useful, generic platform for experimentation with individual components or subsystems in space. This platform must meet the size and mass constraints imposed by the Cubesat standard in order to keep development time and launch cost low.

The challenge addressed by Polysat is to provide the same degree of testing and qualification functionality as a larger satellite, but within the cost, size, and power constraints inherent in any Cubesat design.



(a) (b)

Figure 1.1: (a) CP1 Final Flight Model, (b) CP2 Preliminary Structure Model

CP1, Cal Poly's first Cubesat, was completed and qualified in January 2003, and was designed to provide a "a reliable bus system to allow for flight qualification of a wide variety of small sensors and attitude control devices" [9]. For its first mission CP1 will be qualifying an Optical Energy Technologies sun sensor and an experimental, Cal Poly designed magnetorquer.

CP2, the second Cubesat designed and built at Cal Poly, is a complete redesign of CP1. Its design reflects the lessons learned from CP1 and also provides additional functionality. We hope that the CP2 design is both powerful and flexible, so that it can be used as the reference design for future Cal Poly Cubesats. Currently in its final development stages, the mission of CP2 is to characterize the behavior of ultra capacitor energy storage devices in space.

Both CP1 and CP2 are scheduled for launch with Kosmotros on the Dnepr rocket in early 2005 [10].

Chapter 2

Specification

As a Cubesat, CP2 must meet the requirements and constraints imposed the Cubesat specification as well as those of its own mission. This chapter discusses the basic requirements and constraints of the CP2 project. A set of derived requirements necessary to satisfy the given requirements within the set constraints is formulated, and a set of design decisions made to achieve those derived requirements is presented.

2.1 Requirements

Cubesat Requirements

As mentioned above, the Cubesat specification sets strict requirements on mass and volume. These requirements are non-negotiable as they are required for the Cubesat to fit properly within, and launch correctly from inside of, the Cubesat deployer. The standard Cubesat deployer, used successfully in each Cubesat mission to date, is the Cal Poly designed Poly Picosatellite Orbital Deployer (P-POD) [11].

Figure 2.1, below, shows an external view of the current revision of the P-POD which is capable of simultaneously deploying three Cubesats.

(a) (b)

Figure 2.1: (a) P-POD External View, (b) P-POD Exploded View

The table below enumerates the requirements which directly affect our implementation of the CP2 communications subsystem. For the complete Cubesat specification, see [1].

Requirement	Value (US)	Value (metric)
Mass	$\approx$ 2.2 lb	1 kg
Volume	$\approx$ 61 in^3	1000 cm^3

Table 2.1: Cubesat Requirements

As shown in the table above, a Cubesat is allowed no more than 1 kg of total mass, including structure, power, electronics and payload, and no more than 100 cm^2 per outward face, not including the P-POD railing slides.

Polysat Requirements

A satellite is useless without the ability to communicate with the earth. Our first requirement, therefore, is to create a communication subsystem that can communicate with our earth-based command station, reliably, while in orbit. Communication with the earth can can be established using a wide range of radio frequencies, depending on the data rate requirements, earth station equipment costs, and FCC licensing restrictions.

In order to address the equipment cost and licensing aspect of our communications subsystem, we require that it operates using a frequency within in one of the available amateur radio bands[1].

Using amateur radio frequencies provides the bandwidth necessary to support our mission at a reasonable equipment cost, with the only licensing requirement being that a licensed amateur radio operator be present at the ground station when it is in use. Using amateur radio frequencies for satellite communication is hardly a new concept. Amateur radio enthusiasts launched their first satellite, OSCAR I, in 1961 and there are currently nearly 20 amateur radio satellites in orbit right now [12]. The Polysat project is also fortunate enough to retain one of the original OSCAR team members as a mentor today[2].

Operating over amateur radio frequencies also provides experience and involvement for radio operators around the world. In addition, the Polysat project hopes to harness the knowledge and experience of local ham radio mentors in order to design a practical communications subsystem. It is also hoped that radio operators across the world will decode our data telemetry packets and forward them to Cal Poly in order to assist in monitoring and commanding the satellite.

2.2 Constraints

2.2.1 Limited Communication Window

The geometry of a satellite's orbit dictates a schedule of when, and for how long, the satellite is able to communicate with a fixed ground station. Cubesats are typically launched in what is called a low-earth orbit (LEO). Low earth orbits are characterized by their short range, high orbital velocity and non-geosynchronous nature (Table 2.2, below).

Parameter	Value (US)	Value (metric)
Elevation	400-435 mi	650-700 km
Orbital Velocity	$\approx$ 17,000 mi/h	$\approx$ 27,000 km/h

Table 2.2: Low Earth Orbit Characteristics

[1] The data rate requirement mentioned above is addressed in Section 2.3.2

[2] Cliff Buttschardt, K7RR

The communication window for a satellite is the amount of time that a fixed ground command station can transmit to and receive signals from a satellite. The duration of this window is determined by the orbital parameters, and is defined as the length of time between AOS (acquisition of signal) and LOS (loss of signal) (Fig. 2.2, below).

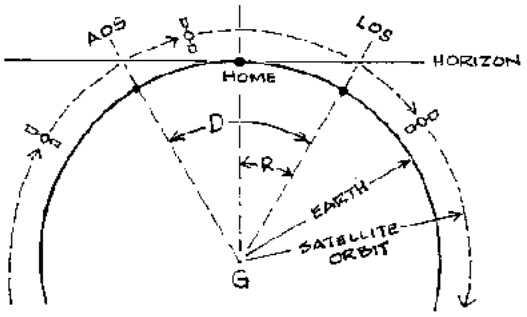


Figure 2.2: Communication Window in terms of AOS and LOS
[http://octopus.gma.org/surfing/satellites/orbit.html]

Based on the orbital parameters set by our launch provider [10], a communication window of approximately 5 to 10 minutes in duration should be available, with 4 to 6 passes above the ground command station per day. A store-and-forward type of communications architecture is key to the success of the communications subsystem due to the short-duration, multiple-pass, low-altitude characteristics of the orbit and the low-cost requirement of the mission [13].

2.2.2 Harsh Space Environment

The harsh environment of space will do its best to meddle with the inner workings of any satellite. Due to the lack of atmosphere in low-earth orbits, charged alpha particles (cosmic rays) can interact with our electronics and cause them to malfunction. Single Event Upset (SEU) and Single Event Latchup (SEL) are two conditions caused by cosmic rays that can alter the otherwise predictable state of our electronics by "flipping bits" in memory (changing 0 to 1 or vice versa) or causing transistors to latch into open- or short-circuit operation [13]. We must design our communications subsystem to prevent and recover gracefully from SEU's and SEL's.

Additionally, space presents an extreme environment in both temperature and pressure. We expect our external structure temperatures to range from approximately $-30°$ to $+70°$C [14] and, in addition, the vacuum of deep space may adversely affect the behavior of our electronics. We must design our communications subsystem to operate in these extreme conditions and test and quantify its behavior before final launch.

2.2.3 Limited Funds

The Polysat project operates under an limited budget. The majority of funding for our satellites is obtained through space technology grants, corporate donations, and sponsorships of payload electronics. The amount of funding is entirely dependent on the complexity of the payload and the generosity of our sponsor. Limited grants can be obtained to continue work on Cubesats without the assistance of corporate sponsors, however the development process slows considerably as the paperwork multiplies. Thus, the communications subsystem should be designed with minimized cost in mind.

2.2.4 Limited Power Budget

The primary power source of the CP2 satellite is a pack of on-board batteries that provide, at full charge, 8.8 W/hour, and are recharged by solar panels which capture a maximum of 1.1 W in direct sunlight [15]. These batteries must support the full functionality of the satellite, and thus power consumption must be strictly budgeted between satellite subsystems.

A preliminary power budget calculation estimates that our average power consumption per orbit will be between 200-400 mW, with the communications subsystem demanding the most power (between 150 and 350 mW, based on either 5 or 10 minutes of transmission per orbit) [16]. Because communications is responsible for consuming 75-88% of our available power budget, every effort should be made to reduce the power consumption of this subsystem.

2.3 Derived Requirements

In order to implement our requirements within the constraints above, the following derived requirements were formulated to help guide our design.

2.3.1 Implement a Simple, Easily Recognizable Beacon

Based on the experience of other Cubesat projects (most notably, CUTE-1 [3]) it was determined that Cubesats with audible beacons were much easier to locate and contact after deployment. After initial deployment, it would be extremely difficult to locate a satellite using only short bursts (less than one second) of AX.25 encoded satellite telemetry (as this design proposes to do). Thus, an audible beacon is required in addition to an AX.25 beacon in order to help audibly acquire the satellite quickly and therefore obtain more usable time from the limited communication window.

2.3.2 Use Short, Reliable Communications Bursts

The communications window given in Section 2.2.1 restricts telemetry and uplinking activities to many short bursts over the course of one day. Due to the short duration of availability, a high data rate is necessary.

Given the payload specification for CP2, a maximum data transmission size of approximately 40 Kbytes is expected[3]. This maximum data size, together with the limited number of passes and variable pass length, will determine a minimum baud rate required for data transmission. The possibilities are calculated below, assuming perfect transmission (Table 2.3).

Duration	Passes	Total Time	Min Data Rate
5 min.	1	300 sec.	1.093 Kbits/sec.
5 min.	3	900 sec.	0.365 Kbits/sec.
5 min.	5	1500 sec.	0.219 Kbits/sec.
10 min.	1	600 sec.	0.547 Kbits/sec.
10 min.	3	1800 sec.	0.183 Kbits/sec.
10 min.	5	3000 sec.	0.11 Kbits/sec.

Table 2.3: Minimum Baud Rate Calculation

In the worst case, a data rate just over 1 Kbps is necessary. Standard data transfer rates are based off of multiples of 300 bps, thus a minimum data rate of 1.2 Kbps should satisfy

[3]Due to non-disclosure agreement, the payload specification cannot be discussed in detail

the conditions above requirements as long as the higher-level protocol chosen has a relatively low overhead. Higher data rates are acceptable as long as they are equally reliable.

Because of the enormous distance between the satellite and the ground command station, the probability of data packet corruption is assumed to be non-negligible. The communications protocol chosen to transmit and receive data must have provisions for providing some sort of reliability. If necessary, a simple handshaking and verification protocol may be implemented on top of an existing, unreliable protocol.

2.3.3 Increase Reliability

In order to function in the harsh environment of space, special efforts must be taken to increase the reliability of the satellite through both hardware and software means. Fully redundant hardware should be exploited where necessary to avoid single point failures, as long as the cost, mass, or volume increase is not prohibitive. Increasing reliability through software should be implemented as well, using techniques such as self-diagnosis and contingency mode operation.

2.3.4 Minimize Hardware Cost

The cost of the satellite should be kept as low as possible, without violating the Cubesat standard or jeopardizing any mission objectives. Commercial-off-the-shelf (COTS) parts, instead of cost-prohibitive space grade parts, must be used when possible in order to keep the cost of electronics down. Standard, commodity parts (programmable microcontrollers, FLASH memory devices, etc) are preferable in order to increase availability and support. Additionally, software solutions should be used to keep the hardware costs down.

Engineering intuition reveals a "diminishing returns" behavior when discussing the relationship between hardware simplicity and software complexity. As the hardware becomes increasingly simple (and thus cheaper) the software becomes increasingly more complex, and therefore more difficult to write, debug and maintain, and thus incurs an increasing cost. A stated goal of the CP2 project is to simplify hardware as much as possible without becoming bogged down in software development time.

2.3.5 Minimize Power Consumption

The choice of communications subsystem electronics must be made in order to keep power consumption to an absolute minimum. Additionally, any software methods available to decrease power consumption should be exploited. Because communications is the primary consumer of power, optimizations made to decrease power consumption of this subsystem will be worth the time and effort.

2.4 Design Decisions

The design decisions below present the most attractive engineering solutions to realizing requirements within the given constraints. After each decision, the derived requirement satisfied or affected by the decision is given in parentheses.

- Utilize the AX.25 packet radio protocol in connectionless mode, with an additional, simple handshaking layer, in order to keep communications bursts short and reliable (2.3.2).

  AX.25 is a specification for transmission of packetized digital data over radio frequencies [17]. Developed by amateur radio operators, AX.25 was designed for simplicity and reliability. The protocol is easily able to handle the sort of data transmissions

required for this design, and do so within the limited communication window. In addition, the AX.25 protocol has a fairly low overhead of 8.2%, allowing a reasonable amount of data to be transmitted in a small time period[4].

By using the amateur radio frequencies we also obtain redundant ground command stations from the ham radio community. Rather than having only a single dedicated ground command station at the Cal Poly campus, we have a network of thousands of amateur radio enthusiasts with similar equipment and capabilities that can be called upon to help track and command our satellite.

- Implement a software TNC, rather than use dedicated hardware, in order to minimize hardware part count and decrease power consumption (2.3.4, 2.3.5).

  A TNC is a device used to decode AX.25 packet radio data. Typically provided as a dedicated hardware device, the TNC functionality will instead be programmed into the microcontroller that controls operation of the communications subsystem. Doing so reduces the number of electronic devices necessary for the communications system, directly addressing cost and power consumption. Since the TNC functionality is implemented completely in software, it weighs nothing at all, which helps the design meet the mass requirement imposed by the Cubesat standard.

- Apply redundant hardware principles to the communications hardware in order to increase reliability (2.3.3).

  Given the requirement to use COTS parts, the issue of increased SEU/SEL rates (compared to radiation hardened parts [13]) must be addressed. A fully redundant communications subsystem can easily become cost prohibitive in traditional satellite designs. However, due to the decreased cost of COTS parts, we have the ability to build in a fully redundant communications subsystem at a modest overall cost. The reliability gained through the use of a fully redundant system was decided to be worth the additional cost[5]. The power requirements remain the same, as the second system is kept powered down unless the first fails. The satellites mass is increased slightly, but even with a fully redundant system, communications electronics account for a very small fraction of the overall mass.

- Implement a contingency mode of operation that takes effect in the event of main processor or main bus failure, in order to provide a method of completing the mission in the event of hardware failure (2.3.3).

  In the case of a main processor or bus failure, either of the redundant communications processors can assume control of the satellite and continue to operate in a contingency mode. This mode electrically isolates the power, payload, and communications subsystems from the main bus and attempts to continue to collect and transmit telemetry to earth. Contingency mode addresses the reliability requirement by allowing the satellite to function even when its main processor has ceased to function properly.

- Research a software controlled, polled receive mechanism in order to minimize receive mode power consumption, a non-negligible consumer of power. (2.3.5)

  A polled receive mechanism aims to reduce the power consumption of the communications receiver devices by reducing their duty cycle. Rather than constantly listening for a command from the ground station, the satellite remains in a low power hibernation state, would wake up to poll its receiver at a given interval and, if no signal is heard, immediately return to hibernation. If a signal is detected, the communications electronics remain powered up for the duration of the reception and then return to hibernation.

[4] Assuming 21 protocol bytes per 256 data byte packet, which is the header size for an AX.25 packet with zero digipeaters

[5] Many single point failures still exist within the satellite, but the failures most easily corrected through redundancy are in the communications system

Requirements Cross-Reference Tables

Tables outlining the requirements, constraints, derived requirements and design decisions are given below for quick reference.

Constraint	Description	Source
2.2.1	Limited Communication Window	Cubesat
2.2.2	Harsh Space Environment	Cubesat
2.2.3	Limited Funds	Polysat
2.2.4	Limited Power Budget	Polysat

Table 2.4: Constraints

Derived Req.	Description
2.3.1	Implement a Recognizable Beacon
2.3.2	Use Short, Reliable Data Bursts
2.3.3	Increase System Reliability
2.3.4	Minimize Cost
2.3.5	Minimize Power Consumption

Table 2.5: Derived Requirements

Derived Req.	Met by Design Decision
2.3.1	Implement a simple morse code beacon to assist tracking
2.3.2	Use AX.25 and a simple handshaking scheme
2.3.3	Use hardware redundancy to increase reliability
2.3.3	Implement a contingency mode in software to increase reliability
2.3.4	Use commodity, COTS parts
2.3.5	Use low power electronics
2.3.5	Monitor and control power consumption using software

Table 2.6: Design Decisions

2.5 Design Evaluation Matrix

In order to determine whether the design satisfies the stated requirements, an evaluation plan must be constructed. Table 2.7, below, lists a set of definitive tests which shall be run upon completion of the design in order to ensure its functionality meets the specified requirements.

#	**Test**	**Status**	**Req't**	**Date**
1	Transmit simple morse code beacon	—	2.3.1	—
2	Transmit status encoded morse code beacon	—	2.3.1	—
3	Encode and transmit AX.25 data packet	—	2.3.2	—
4	Receive and decode AX.25 data packet	—	2.3.2	—
5	Parse basic commands from ground station	—	2.3.2	—
6	Respond to CDH request for subsystem status	—	2.3.2	—
7	Respond to CDH request for data transmission	—	2.3.2	—
8	Detect main bus (or processor) failure	—	2.3.3	—
9	Implement contingency mode	—	2.3.3	—
10	Evaluate component costs	—	2.3.4	—
11	Measure power consumption in TX mode	—	2.3.5	—
12	Measure power consumption in RX mode	—	2.3.5	—

Table 2.7: Design Evaluation Matrix

This table is revisited in Chapter 4 (Evaluation), where the results of each test are presented.

Chapter 3

Development

Cubesat developers have different requirements for their communications subsystems. In this chapter, the designs of other university Cubesats are categorized and some of the factors influencing their design decisions are analyzed. Then, the design of the communications subsystem for CP2 is presented, and the specifics of how it satisfies the stated mission requirements and constraints are explained.

3.1 Existing Solutions

Cubesat communications systems to date can be categorized into three major groups, based on the type of hardware used: (1) modified handhelds; (2) Commercial-off-the-shelf transceiver devices; or (3) custom hardware.

In the first group, handheld amateur radios (such as the Yaesu VX-1R [18]) are stripped down to bare circuit boards and modified to interface with the power supply and main processor of the Cubesat. This approach simplifies implementation of the communications subsystem since the amplifier, transceiver, and sometimes even the TNC are already integrated into a single board and known to function correctly. On the other hand, even the smallest handheld radios are large enough that fitting them into a confined space, such as the inside of a Cubesat, is no easy task. Additionally, no new knowledge of the issues involved with implementing a communications subsystem is gained by using a solution that is already a known good. Finally, the use of a modified amateur radio limits the ability to tackle key issues such as power consumption and configurability.

The second group addresses these concerns by using commercial-off-the-shelf (COTS) transceivers and amplifiers that are available as single-chip devices. These devices can be placed virtually anywhere on a PC board and choosing the parts individually forces the design team to deeply understand the issues involved in designing a functional RF interface board. The obvious disadvantage of this approach is an increased design time, since the devices must be individually tested and then integrated together, laid out on a PC board with suitable RF characteristics, and tested again. In the end, however, the total hardware requirement (and thus power consumption) is lowered, since only up to three RF devices (amplifier, transceiver, TNC) are required and, optimally, each can be individually enabled or disabled to save power.

The third group obtains the highest flexibility at the expense of design time and cost. The University of Tokyo is a prime example of this category, as its design uses a completely custom RF transceiver.

A table comparing the various components chosen by other universities for their communications subsystem is shown below (Table 3.1).

Project	FM Band	TNC	Transceiver	Output Power
CANX-1 [2]	900 MHz	—	CMX469	0.5 W
Cal Poly, CP1 [19]	440 MHz	—	Alinco DJ-5C	0.3 W
Cal Poly, CP2 [15]	440 MHz	PIC18	CC1000	1 W
DTUSat [5]	440 MHz	—	CMX469	1 W
Hawaii [20]	440 MHz	PIC16+MX614	VX-1R	0.5 W
Montana State [21]	144/440 MHz	PicoPacket	VX-1R	1 W
Stanford [7]	2.4 GHz	MHX 2400	MHX 2400	1 W
Tokyo [4]	144/440 MHz	PIC16	Nishi RF Lab.	0.8 W

Table 3.1: Communications Subsystem Comparison

3.2 A New Solution

Based on the derived requirement of using AX.25 (2.3.2), a communications system capable of transmitting and receiving AX.25 formatted data is necessary. Of those designs surveyed above which used AX.25, most were either in class (1) or class (3) (discussed above, see 3.1), which are unacceptable due to cost and power concerns: custom designs are outside of our limited budget and handheld transceivers consume too much power and require higher power supply voltages than the CP2 power subsystem can provide.

The final design is most similar to that of the University of Tokyo's Cubesat, XI-IV [4], except a low power, commercial-off-the-shelf transceiver is used rather than a custom built one, in order to keep power consumption low (compared to traditional high power transceivers and amplifiers) and cost minimized (relative to space rated devices). The final power consumption and cost figures for the design are given in Chapter 4 (Evaluation).

Additionally, fully redundant transceiver and amplifier hardware will be used in order to increase reliability. This decision was made with the understanding that hardware redundancy was the most straightforward way to decrease the probability of mission failure. The small size, low power consumption and low cost of the communications subsystem gave us the opportunity implement it in a fully redundant configuration, with identical communications controllers, amplifiers and transceivers. Even with the size, power and cost of the subsystem doubled, the design still meets our requirements.

A block diagram of the communications subsystem hardware is shown below (Fig. 3.1).

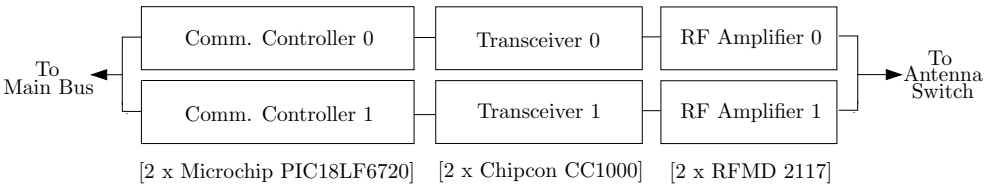


Figure 3.1: Communications Subsystem Hardware Block Diagram

3.3 Hardware Development

This section details the hardware components chosen for this communications subsystem design and the reasons behind those decisions. For more detail on the electrical characteristics, see the CP2 full electrical schematic [15].

Communications Controller: Microchip PIC18LF6720

The Microchip PIC18 family of microcontrollers are highly integrated parts, available with on-chip FLASH memory and static RAM as well as built-in serial and parallel peripheral

interfaces. The PIC18LF6720 was chosen specifically for its large amount of FLASH memory (256 Kbyte) for program storage, large static RAM (4 Kbyte) for run-time variables, support for the Inter-IC Communication (I^2C) bus (the protocol used to communicate with the main satellite bus) and its extremely low power requirements and power management abilities[1]. For more information, see the PIC18FXX20 family data sheet [22].

Transceiver: Chipcon CC1000

The Chipcon CC1000 provides, in a single device IC, the RF modulation necessary to transmit data using AX.25. The part is also a commodity, COTS part that is cheap enough that full redundancy is feasible[2]. In addition, the CC1000 is a low-power part, drawing approximately 25 mA at full transmit power [23].

The CC1000 transceiver exchanges data with the PIC microcontroller via a two-wire serial data bus with clock (DCLK) and data (DIO) lines. The CC1000 is configured to send a clock pulse to the PIC at each bit-time (e.g. the time between a single bit and the next). At 1200 baud, this leads to a clock period of 833 μsec. On each clock edge, an interrupt is triggered on the PIC, causing the software TNC to process the incoming bitstream. Additionally, the CC1000 has a three-wire programming bus consisting of PALE, PCLK and PDATA signals. This bus is responsible for setting the frequency, output power, baud rate and encoding of the CC1000.

Fig. 3.2, below, shows the data and programming bus connections between the PIC and CC1000.

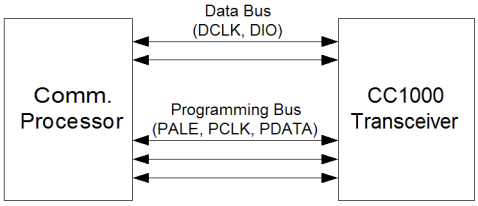


Figure 3.2: PIC-CC1000 data and programming buses

RF Amplifier: RF Microdevices RF2117

The RFMD RF2117 is a COTS amplifier designed for use with RF signals between 400 and 500 MHz. The part is also capable of operation at a supply voltage of 3V, which provides a significant reduction in power consumption over the typical 5V supply: the RF2117 sinks a maximum of 1,100 mA of current at either supply level, thus 3V operation consumes 3.3 W while 5V consumes 5.5 W. The RF2117 is also a single chip device and is available in the small, surface mount packages necessary for this design.

3.4 Software Development

The communications software is implemented in C using the MCC18 compiler and MPLAB IDE, both available from Microchip, Inc. The source code is structured as a set of files corresponding to the major modules of functionality required for the communications subsystem. A brief description of the functionality provided by each file is given below (Table 3.2).

[1]The PIC LF series of processors are low-power parts, capable of running with a supply voltage as low as 2V. Additionally, the LF parts are capable of entering a sleep mode in which they draw only about 1 μA of current.

[2]See Table 4.3

File	Description
cw.c	Implements morse code beacon functionality.
cc1000.c	Implements the CC1000 transceiver device interface
tnc.c	Implements the software TNC
comm.c	Implements the communications controller main processing loop
i2c.c	Implements the I^2C main bus interface

Table 3.2: Software Module Descriptions

The relationship between modules of the software design is given in the block diagram below (Fig. 3.3).

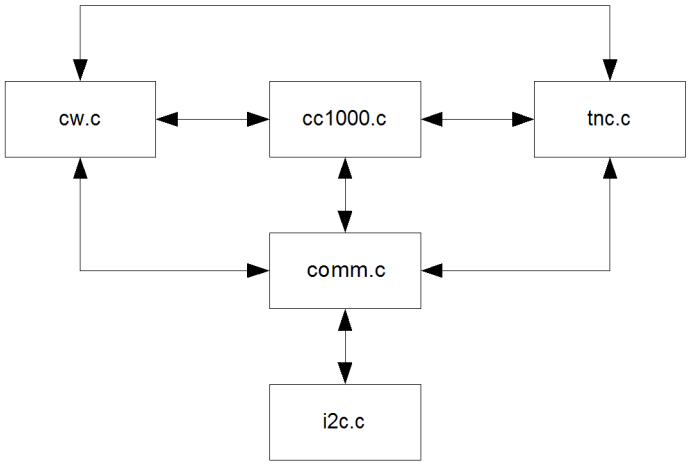


Figure 3.3: Software Module Block Diagram

Detailed Functionality

This section provides the details of each file in the software design outlined above. The full source code to each of these modules is available in Appendix F.

Main Communications Control (comm.c)

The behavior of the communications subsystem is governed by simple state machine. There are three main modes of operation – Pre-ops, Normal Ops, and Contingency – which correspond directly with the three main state machine states. Three auxiliary states are also defined which validate and execute commands from the ground command station as well as transmit telemetry beacons and payload data.

Fig. 3.4, below, is a state diagram detailing the behavior of the main communications control state machine.

In Pre-ops, functionality is minimal: Morse code and AX.25 beacons are transmitted at 2 minute intervals while waiting for an uplink command. Once an uplink command is received, the satellite proceeds to Normal Ops. In Normal Ops, the main task is to ensure that uplink commands are received and decoded and that AX.25 beacons and payload data are transmitted in response. If the main bus has failed, a contingency mode is activated in which the communications controller is isolated from the bus and attempts to maintain a "minimum operations level" necessary to complete the mission.

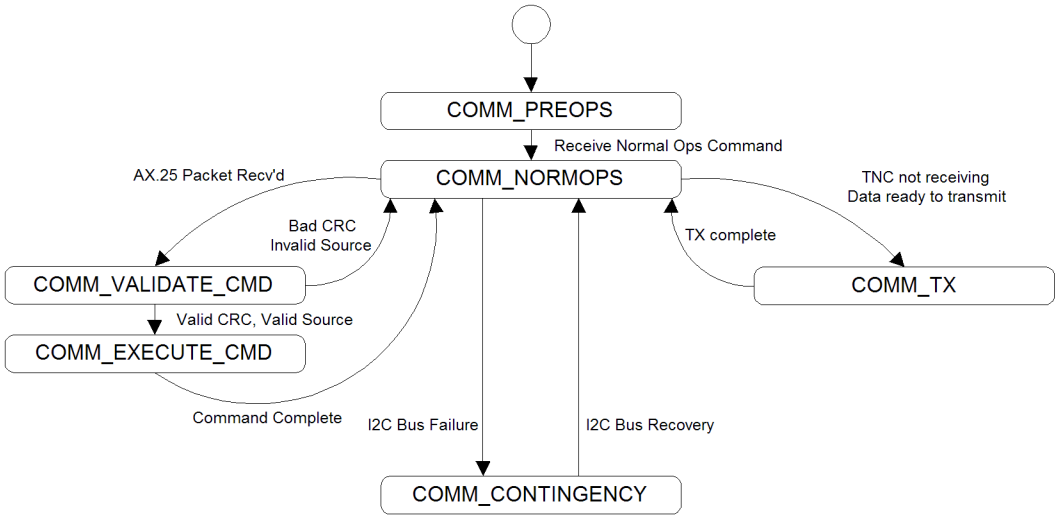


Figure 3.4: Communications Controller Software State Machine

Contingency Mode Details

Contingency mode operation is a fail-safe mode in which the communications controller assumes full control of the satellite payload in order to attempt to complete the mission. This mode detects main bus failure by setting a timeout that resets with each I^2C transaction received from the main controller. If the communications controller does not receive an I^2C transaction from the command and data handling (CDH) processor before the timeout expires, the communications controller assumes that the CDH processor or main bus is dead, and isolates itself and the payload circuitry from the rest of the satellite bus. The communications controller then commands the payload as necessary and transmits the results back to earth.

If the main bus/processor failure was transient and is restored, the communications controller relinquishes control of the payload to the CDH controller, and returns to normal operations mode.

Figures 3.5 and 3.6, below, show the operation of the bus in both normal ops and contingency modes.

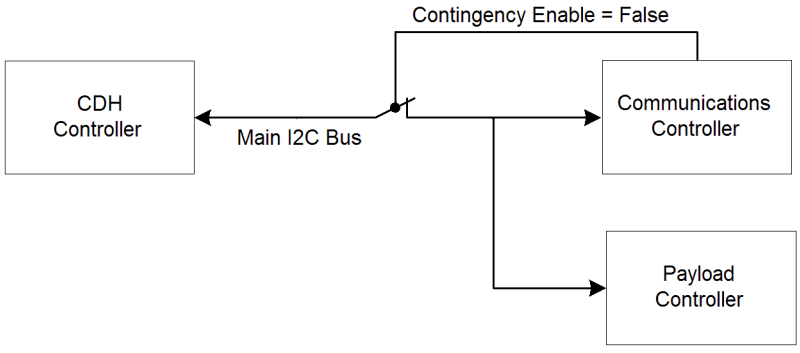


Figure 3.5: Normal Ops Mode

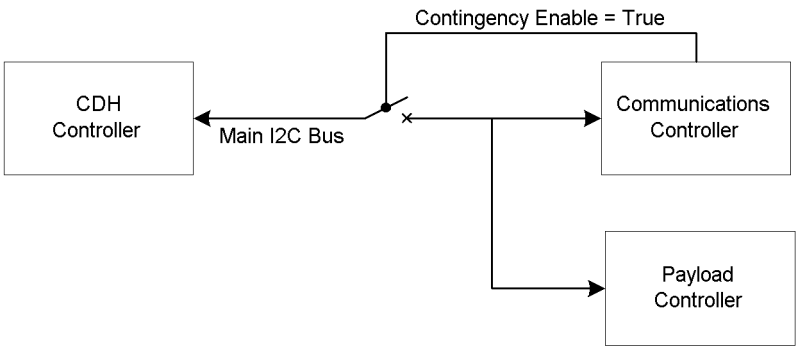


Figure 3.6: Contingency Mode

Main Bus Interface (i2c.c)

The command and data handling (CDH) and payload processors are connected to the communications processors using the Philips I^2C two-wire serial bus protocol [24]. The CDH processor is configured as an I^2C master device, while the communications and payload processors are slave devices. In this configuration, the CDH processor must initiate all data transfers since it is the only master device on the bus.

The PIC18LF6720 has on-board support for the I^2C protocol, including the ability to interrupt on an I^2C slave address match. This capability is used to implement an interrupt-based bus communication system that ensures that no processor has to wait in an idle loop longer than necessary in order for a bus transaction to complete.

A block diagram of the I^2C interrupt service routine is given in Figure 3.7, below.

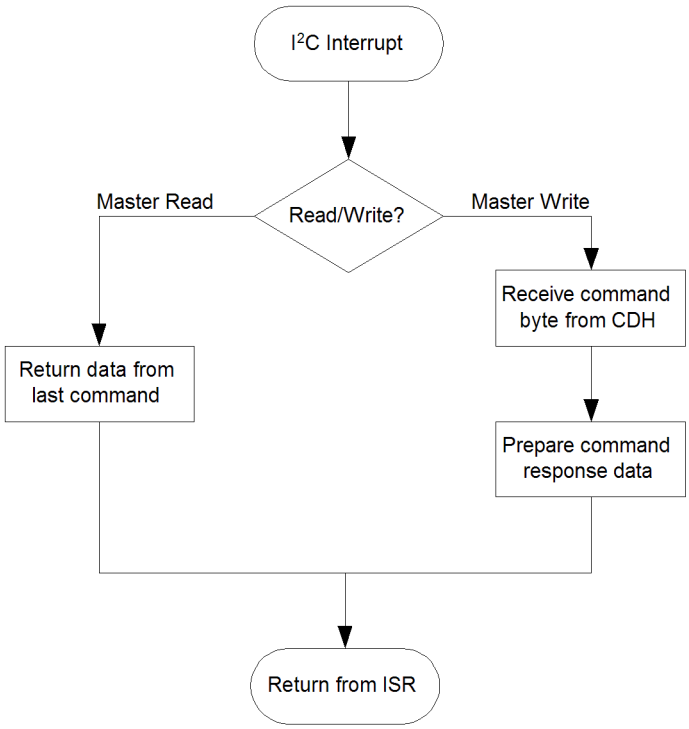


Figure 3.7: I^2C Interfacing

Morse Code Beacon (cw.c)

The morse code beacon transmits a sequence of dits and dahs at 1200 Hz audio tone and using to an ASCII-to-morse lookup table. When the appropriate morse equivalent is found, it is transmitted by turning the CC1000 internal power amplifier on and off at the proper intervals. This technique is known as on-off-keying (OOK) and is described in the CC1000 Application Note AN0016 [25].

The beacon itself provides basic health information on the satellite, in a simple alphabetic format described in Appendix D.1.

Transceiver Interface (cc1000.c)

The CC1000 transceiver is a highly programmable device. Using its programming bus, the carrier frequency and power consumption levels can be set, the PLL can be recalibrated, and switching between transmit and receive modes is quite painless. This programmability is achieved through the inclusion of 28 8-bit registers on the CC1000 itself. These registers control every aspect of the operation of the CC1000 and are fully programmable though the programming bus.

The low-level details of calibration, mode change (transmit to receive and vice versa) and full chip reset are abstracted away to single line function calls, but the ability to write and read registers individually is also available.

Software TNC (tnc.c)

In order to transmit telemetry information, it must first be encoded in the AX.25 data packet format. Additionally, in order to receive commands from the ground command station, the communications subsystem must be able to decode AX.25 packets. In other Cubesats, this functionality is typically accomplished with a dedicated device, either a modified handheld or single IC device (see Table 3.1). In this design, the PIC18LF6720 is programmed with a software TNC that is responsible for both encoding and decoding AX.25 data.

Only one other Cubesat project to date, the University of Tokyo, uses a software TNC implemented on a PIC microcontroller. In their implementation, a custom FM transceiver is interfaced with an MX614 tone modem chip, which converts AX.25 AFSK encoded data into digital data bits that are fed to the PIC. Our implementation differs in that the CC1000 transceiver device provides the digital bits directly to the PIC, obviating the need for a separate modem chip. Thus the choice of transceiver and decision to implement the TNC in software led directly to a reduced part count and simplified hardware.

The software TNC is implemented in the PIC as a state machine that updates after each bit of information is transmitted or received. The states of the software TNC correspond to the fields of an AX.25 frame, providing information on where the software TNC is currently processing. The TNC is interrupt driven, with the interrupt signal provided by the clock of the CC1000 (DCLK). The state diagrams of the software TNC, in both receive and transmit modes, are shown on the next page (Fig. 3.8 and 3.9).

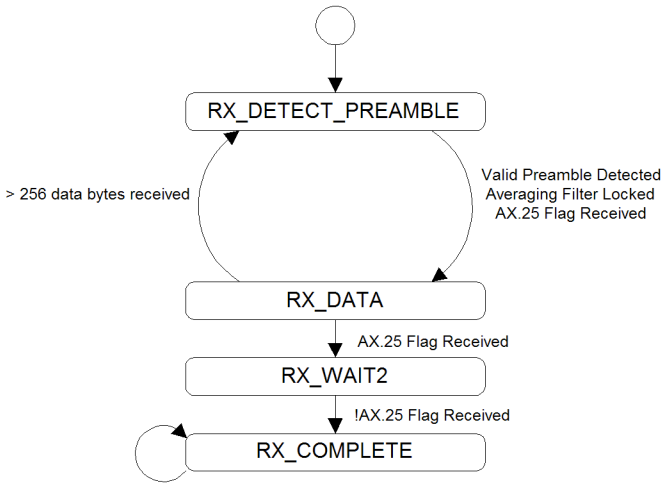


Figure 3.8: Software TNC RX State Machine Diagram

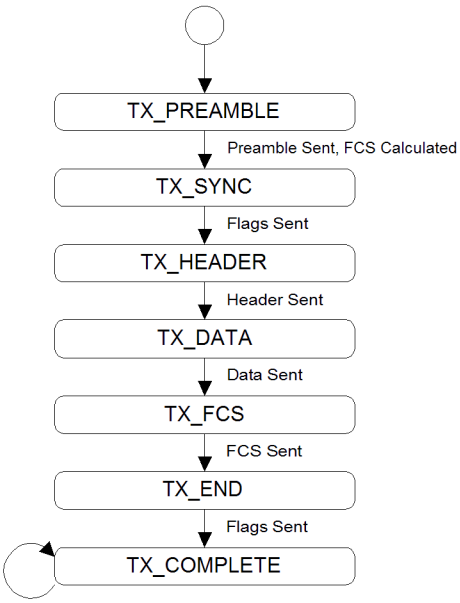


Figure 3.9: Software TNC TX State Machine Diagram

Chapter 4

Evaluation

4.1 Prototype

A prototype of the specified design is shown below (Fig. 4.1). This prototype includes redundant communications processors and transceivers, but does not include the RF amplifiers. Without the amplifiers the prototype is functionally equivalent to the full design except that it transmits at a much lower power (10 mW instead of 1 W).

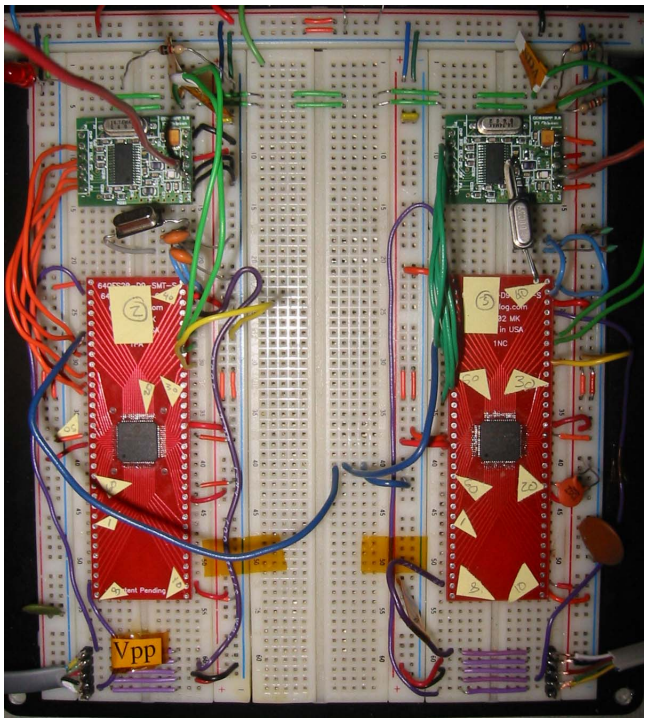


Figure 4.1: Communications Subsystem Design Prototype

On the left side, top to bottom, are the Chipcon CC1000 transceiver and the Microchip PIC18LF6720 microcontroller that are considered Communications A. To the right, the identically configured Communications B is the redundant spare. Along the top of the prototype board you will find the horizontal two-wire bus which simulates the I^2C bus lines SDA (top) and SCL (bottom). The CC1000 programming and data busses are also clearly visible between communications controllers and their respective transceivers. Also, the RF_SEL pins of each processor are connected to the center of the proto-board and can

be used to simulate the enabling and disabling of either Communications A or B.

Fig. 4.2, below, shows the completed command and data handling (CDH) board, which is where the communications subsystem is physically located. In the lower left corner are the footprints for the redundant RF 2117 amplifiers and in the middle left are the footprints for the redundant CC1000 devices. The lower right corner shows the footprint of a single PIC18LF6720 communications processor. The second (redundant) processor is located on the opposite side of the board.

The entire command and data handling and communications subsystems are implemented on a single board measuring roughly 3" by 3".

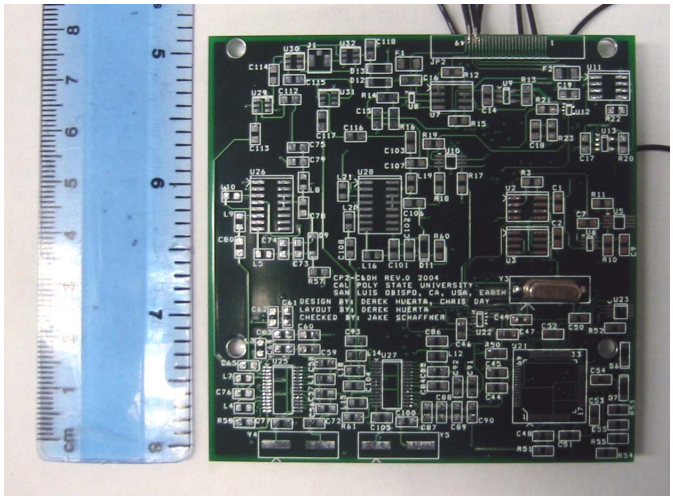


Figure 4.2: Communications Subsystem Board

4.2 Discussion

With the design completed, the communications subsystem should now be capable of functioning under the requirements and constraints given in Chapter 2 (Specification).

The design evaluation matrix on the next page (repeated from Section 2.5), assists in evaluating the design by displaying the completion status of our stated requirements evaluation tests. Each test addresses a specific behavioral component of a requirement, and by completing each test successfully, the design can be proven to meet the stated requirements.

#	Test	Status	Req't	Date
1	Transmit simple morse code beacon	Complete	2.3.1	4/3/04
2	Transmit status encoded morse code beacon	Partial	2.3.1	4/3/04
3	Encode/transmit AX.25 data packet	Complete	2.3.2	5/15/04
4	Receive/decode AX.25 data packet	Complete	2.3.2	5/28/04
5	Parse basic commands from ground station	Complete	2.3.2	5/28/04
6	Respond to CDH request for subsystem status	Complete	2.3.2	4/15/04
7	Respond to CDH request for data transmission	Complete	2.3.2	4/15/04
8	Detect main bus (or processor) failure	Complete	2.3.3	6/1/04
9	Implement contingency mode	Complete	2.3.3	6/1/04
10	Evaluate component costs	Complete	2.3.4	3/26/04
11	Measure power consumption in TX mode	Partial	2.3.5	6/10/04
12	Measure power consumption in RX mode	Partial	2.3.5	6/10/04

Table 4.1: System Test Requirements

This evaluation will focus on how the derived requirements given in Section 2.3 were met. The itemized list below restates each derived requirement and explains how it was met by the design.

- Implement a Simple, Easily Recognizable Beacon (2.3.1).

  This derived requirement was met through the implementation of a morse code beacon. A morse code beacon is instantly recognizable and can be decoded in memory by amateur radio operators with enough experience. The implementation of the beacon itself involved switching the output power of the transceiver between full on and full off, a method called on-off-keying (see [25] for more details).

  Tests 1-2 address the behavior of the morse code beacon. Test 1 is marked completed as a simple morse code beacon can be transmitted. Test 2 is partially completed because the average temperatures and voltages cannot be encoded until the Command and Data Handling board containing the voltage and temperature sensors is completed.

- Use Short, Reliable Communications Bursts (2.3.2).

  This derived requirement was met through the decision to use 1200 baud, AFSK encoded AX.25 data packets. AX.25 provides a frame check sequence (FCS) which provides a simple way to determine whether the data inside of a packet has been corrupted. In addition, a simple command/acknowledge protocol will be implemented within the data field of the AX.25 packet in order to have completed knowledge that a command was received properly by the satellite.

  Tests 3-5 address the behavior of the Software TNC responsible for encoding and decoding AX.25 packets. All tests for this derived requirement have completed successfully.

- Increase Reliability (2.3.3).

  This derived requirement was met through the use of fully redundant hardware, smart-fuses and a software contingency mode.

  The hardware of the communications subsystem is fully redundant in order to minimize single point failures. This solution was not cost prohibitive because of the use of COTS parts. Also, the power budget is unaffected because the redundant set is disabled until it is selected.

  The threat of SEL's is mitigated through the use of smart-fuses in the power supply system of the Command and Data Handling board. When a SEL occurs, the latchup causes a short in the power system which causes the smart-fuse to activiate. When active, it intelligently cycles the power, clearing the latchup (theoretically). SEU's

are handled similarly; if they occur in the right place, they will cause the software to hang, which will trigger the watchdog timer and cause the system to power cycle. In the worst case, an SEU will corrupt a single bit of telemetry or payload data and be completely unnoticeable. This final case is not detected in the current design.

The software contingency mode allows the satellite to continue its payload mission even in the event of main processor and main bus failure.

Tests 8-10 address the software contingency mode implementation. All tests for this derived requirement have completed successfully.

Reliability Testing

In order to attempt to evaluate the reliability of our system, the following scenarios must be tested in order to meet a minimum reliability standard in the face of "real-world" situations (e.g. invalid or corrupted data).

Due to time constraints, these tests will be conducted at a later date.

Test	**Pass**	**Fail**	**Date**
Verify AX.25 decode with zero length packet			—
Verify AX.25 decode with max length packet			—
Verify AX.25 decode with oversized packet			—
Verify AX.25 decode with incomplete packet			—
Verify command decode with zero length command			—
Verify command decode with max length command			—
Verify command decode with oversized command			—
Verify CDH write to comm			—
Verify CDH write to comm with oversized transmission			—
Verify CDH write to comm with simulated bus collision			—
Verify CDH write to comm with simulated bus failure			—
Verify contingency mode operation			—
Verify return to normal ops from contingency mode			—

Table 4.2: Reliability Test Requirements

- Minimize Hardware Cost (2.3.4).

  This derived requirement was met through the use of COTS parts and the choice to use a software TNC. COTS parts, while not rated for use in space, are relatively inexpensive. The use of these parts for short-duration satellite missions has been tried and tested [26], and appears to be a viable solution to low-cost satellite development.

  The software TNC component allows the communications processor to take on the task that would normally be handled by a separate physical device. Since the communications processor is already running and is nowhere near maximum utilization, the added computation does not adversely affect its behavior.

  Test 10 states that the component costs should be evaluated. The total cost of hardware and development kits for this design is given on the next page in Tables 4.3 and 4.4.

Quantity	Part	Description	Cost (ea.)
2	PIC18LF6720	PIC microcontroller	$10.73
2	RF2117	RF Power Amplifier	$12.60
2	CC1000	Transceiver Modules	$45.00
		Total	$136.66

Table 4.3: Hardware Costs

Quantity	Description	Cost (ea.)
2	Microchip ICD2 PIC programmer	$159.99
1	Microchip MCC18 C Compiler suite	$495.00
1	CC1000-DK433 development kit	$440.00
1	CC1000SK Sample Kit (5 transceivers)	$45.00
	Total	$1299.98

Table 4.4: Development Kit Costs

A total of $1,436.64 meets our definition of low-cost for a reliable, low-power, satellite communication subsystem.

- Minimize Power Consumption (2.3.5).

  This derived requirement was met through the use of low power devices and a software TNC. All devices in the design operate with 3V positive supply and both the CC1000 and the PIC18LF6720 are designed to draw low currents.

  Tests 11-12 provide the quantitative measure of power consumption for this design. Table 4.5, below, shows the results of these tests. Note that these tests are only partially completed because the measurements below reflect the prototype design and do not include the RF2117 amplifier.

Mode	Voltage	Current	Power
Receive	3 V	15 mA	45 mW
Transmit	3 V	34 mA	102 mW

Table 4.5: Prototype Power Consumption (without amplifier)

From the RF2117 data sheet, we expect the amplifier to consume roughly 3.3 W of power (1,100 mA at 3V). Thus the transmit mode measurement in the table above should increase by approximately 3.3 W. In receive mode, the power consumption should remain unchanged.

Chapter 5

Conclusion

Designing and implementing a communications subsystem for a Cubesat is a project full of engineering tradeoffs. The size and limited power budget of the satellite restricts type and number of electronic components, and the limited budget further constrains the choice of parts. The mission requirements further state that the communication subsystem should be as reliable as possible, which directly contradicts the low power and low cost constraints. Thus a design satisfying all of these conditions must be able to provide the most reliability possible with the least amount of impact on financial and power consumption budgets.

The current communications subsystem design of CP2 meets these conditions by exploiting low power, commercial-off-the-shelf components, which are both relatively cheap and power efficient, and using low power, programmable microcontrollers to provide, in software, functionality that typically requires additional hardware. The reliability of the subsystem, which is a top priority, is ensured through a two level approach: First, through hardware redundancy, and second, through software detection and correction.

The road to success was paved with many hours of debugging and frustration. The original design envisioned a 1200 bps uplink, while technical difficulties (detailed in Appendix B) forced us to reduce to a 600 bps uplink. Thankfully, this decision did not adversely impact the mission due to the small size of uplink commands (256 bytes or less). The choice of the CC1000 transceiver was made very early on in the CP2 design, without knowledge of the issues involved, and led to many of the problems we encountered. A future design will hopefully take notice of the issues discovered in this design, and correct them in the next satellite.

The lack of decent development software was an unexpected hinderance; we assumed that the C compilers and IDE's for the PIC were stable and mature, while we discovered exactly the opposite. We found it extremely difficult to write and debug code for a communications subsystem when the IDE crashes consistently and the C compiler generates bad code.

In the end, however, the design met all the stated requirements and represents a solid approach to the implementation of a Cubesat communications system.

Further enhancements and remaining issues in the design are discussed in the next chapter.

Chapter 6

Future Work

Given an unlimited budget, no deadlines and a bottomless pot of coffee, all of the issues in the current design could be addressed and corrected. But the constraints of the real world force us to mark the design as final, even though there are many issues that remain to be fixed. These issues represent the lessons learned through the course of designing and implementing the communications subsystem.

The bulk of the issues below are due to the transceiver part selection. While the Chipcon CC1000 met our low power and low cost constraints, it is a device that is not designed for transmitting 1200 bps AX.25 data packets over amateur radio frequencies (see Appendix B.3 for more details). In hindsight, a more traditional transceiver design with multiple components, rather than the highly integrated CC1000 transceiver, could have allowed us to tailor the carrier frequency and binary FSK frequency separation to meet our AX.25 derived requirement. Alternatively, as noted below, if we had discovered the issues with 1200 bps data packets earlier on, we could have made more progress into implementing 9600 bps transmission.

Currently, the prototype design for the next communications subsystem does not use the CC1000; instead, a more traditional separate component approach will be taken, and a more conventional high speed modulation scheme (QPSK) will be used to obtain higher data rates while still remaining within bandwidth constraints. Accordingly, the power consumption of the power system will increase, but we hope that the magnitude of the increase is manageable and within our power budget.

- Replace the CC1000 with a more suitable device.

  The CC1000 transceiver is a wideband transceiver, designed for frequency separations on the order of 64kHz. In this project, the decision to use AX.25 over 1200 bps AFSK was made before the frequency separation limitations of the CC1000 were known. The limitations of the CC1000 were not discovered until the communications board schematics and layout were already completed, leaving no choice but to continue with the CC1000. The CC400 is a similar device offered by Chipcon, which is designed for narrowband operation. A future design may require a new board layout, and at that point the CC400 could replace the CC1000.

- Update the software to support 9600 bps operation.

  Modifying the current software TNC code and CC1000 programming to encode AX.25 over 9600 bps FSK (based on a well-tested hardware design by G3RUH [27]), would increase frequency separation, increase data rate by a factor of 8, and decrease bandwidth usage by using a more efficient encoding scheme. The changes should be localized to software only, as the necessary scrambling and descrambling hardware specified in the G3RUH design can easily be performed in software. This could allow the next design to continue to use the CC1000 and avoid an unnecessary redesign.

- Increase the sensitivity of the contingency mode detection.

  Contingency mode detection is currently implemented using a software I^2C watchdog timer which monitors the I^2C bus and, if triggered, signals to the communications processor that the main bus and/or main processor have become inoperable. There are other main processor failure modes which might require contingency mode operation which would still leave the I^22 bus in an operable state, and thus not trigger the contingency detection of the communications processor. Other failure modes could be checked for by using other pins connected to the main processor or through an I^2C command response from the communications processor that could trigger a self-diagnostic test to be performed on the main processor.

- Implement a polled receiver mode to reduce power requirements.

  Due to time constraints, a polled receive mode was not implemented in the current design. A future version of the software could easily implement a polled receiver loop in order to further minimize power consumption in receive mode.

Chapter 7

Acknowledgments

- Chris Day (KF6HIL) and Spencer Studley (KG6PGA)

  For their leadership in the Polysat project.

- Jake Schaffner (W6RWS)

  For his incredible depth of knowledge, and willingness to share said knowledge with yours truly.

- Kyle Leveque (KG6TXT) and Jacob Farkas (KL1PU)

  For all the late weekend nights in the lab spent pondering the existence of do/while loops.

- Dr. Jordi Puig-Suari and Dr. Clark S. Turner (WA3JPG)

  For allowing me to give them a hard time.

- Michael Gray (KD7LMO)

  For giving us permission to use his MicroBeaconII source code as the basis of our software TNC.

- Denis Nechitailov (UU9JDR)

  For his willingness to modify MixW2 to work with our transceiver.

- The House of the Flaming Sword (1503 Slack Street)

  For keeping all this satellite business in perspective.

And of course, all members of the Polysat team, without whom there would be no picosatellite, and no communications system to design.

Appendix A

Glossary of Terms

AOS. Acquisition of Signal. The point of time during a satellite pass when the satellite's communication signal is first heard.

AX.25. Amateur X.25. A protocol used for transmission of data packets over radio frequencies.

Cal Poly. California Polytechnic State University, San Luis Obispo, California.

CDH. Command and Data Handling. Refers to the satellite subsystem responsible for obtaining sensor data and executing commands received from the ground command station.

COTS. Commercial Off The Shelf. Refers to commodity parts that are mass-produced for use in commercial applications. COTS parts are typically much cheaper than equivalent space-rated parts, although technically they are not qualified to operate in space

CP1. The first Cubesat designed, manufactured and tested by undergraduates at Cal Poly.

CP2. Cal Poly's second undergraduate picosatellite.

CW. Continuous Wave. Shorthand for morse code.

FCC. Federal Communications Commission. The government entity responsible for allocation and coordination of radio frequency bands within the United States.

I^2C Bus. Inter-IC Communication Bus. A standardized two-wire bus protocol developed by Philips Semiconductor.

LEO. Low Earth Orbit. A satellite orbit characterized by low orbital altitude and high orbital velocity.

LOS. Loss of Signal. The point of time during a satellite pass when the satellite's communication signal is lost.

P-POD. Poly-Picosatellite Orbital Depolyer. The Cal Poly designed deployer used to deploy Cubesats.

PIC. A family of microcontollers produced by Microchip Devices, Inc.

RAM. Random Access Memory. A non-volatile digital storage device.

SEU. Single Event Upset. A condition in which the state of a bit in memory is inverted due to the impact of a charged alpha particle. SEU's are a common occurrence in space due to the lack of protective atmosphere.

SEL. Single Event Latchup. A condition in which a solid-state transistor shorts due to the impact of a charged alpha particle. SEL's are a common occurrence in space due to the lack of protective atmosphere.

TNC. Terminal Node Controller. A device that is used to decode digital radio data packets.

Appendix B

Problems Encountered

B.1 Microchip Development Tools

- Microchip MPLAB IDE

  The MPLAB IDE is an integrated development environment, made freely available by Microchip, Inc., in order to assist in programming and debugging the PIC processor. Sadly, the IDE was more of a hinderance than a help, as it crashed without reason nearly every day. Microchip seemed responsive to their users as new versions of the IDE were released almost bi-weekly, but with each new version came fixes for old bugs which brought on newer and often more serious bugs.

  Over time, the actions that would cause the IDE to crash most often were used less and less, until working around them became second nature.

B.2 Chipcon Development Kit Software

- CC1000 SmartRF Studio Software Bugs

  The CC1000 development kit includes a useful software tool called SmartRF that allows a user to program the development board using a parallel port connection from a PC. This allows the user to experiment with different frequency and bit rate settings without having to struggle with interfacing directly with the serial programming bus.

  A default frequency of 436.8466 MHz was chosen arbitrarily for testing. After a frequency coordination request with the IARU [28], the frequency assigned to the mission was 437.485 MHz. Using SmartRF studio, the values of the 6 CC1000 frequency registers can be easily calculated from the desired frequency in MHz. When the 437.485 MHz frequency is given to the software, however, it calculates configuration values for a frequency of 437.4137 MHz. Through trial and error, the next highest frequency supported by the software is 437.6028 MHz, even though the data sheet explicitly mentions that the frequency is programmable in 250 Hz increments. This led to using the data sheet, which contained the description of the frequency registers, to calculate the necessary values by hand.[1]

B.3 Chipcon CC1000 Hardware

- No way to easily transmit AFSK data necessary for AX.25

[1] This appears to be a limitation of the CC1000 chip itself. Although the data sheet specifies frequency programmability of 250 Hz, that is achieved only with frequency separations much higher than 1 kHz.

The CC1000 is designed for use in FSK (frequency-shift-keying) applications. FSK transmits digital data by modulating the frequency of the carrier in order to signify a binary 0 or 1. The spacing (in the frequency domain) between the frequency of the zero bit and the one bit is called the frequency separation, and, according to the CC1000 data sheet, is configurable between 0 and 64 kHz.

AX.25 transmits data using AFSK (audio-FSK) which involves a common carrier frequency on top of which audio tones of 1200 Hz and 2200 Hz are modulated. A 1200 Hz tone signifies a zero bit, while 2200 Hz signifies a one bit. In this modulation scheme, a common RF carrier frequency is employed, but the carrier is modulated with an audio tone which carries the data bits.

The CC1000 is not designed to transmit AFSK. The inputs to the chip are simply DIO (data bits) and DCLK (data clock). The transceiver itself is responsible for modulating the bits on DIO into FSK. Therefore, in order to use the CC1000 as a part of a software TNC, some method of transforming between FSK and AFSK must be found.

The Yaesu FT-847 radio, used at our ground station, can operate in lower-side-band (LSB) mode, a mode in which only the lower half the FSK signal from the CC1000 is demodulated, which maps our original FSK transmit frequencies to audio tones which correspond closely with the 1200Hz/2200Hz tones necessary for AX.25 reception. In this mode, it is possible for our ground station to treat the FSK transmission from the CC1000 as an AFSK transmission and decode it accordingly.

- Inability to receive 1200 bps AX.25 data streams due to frequency separation issues

  In order to map between FSK and AFSK, a frequency separation of 1 kHz must be programmed into the CC1000. At this frequency separation, 1200 bps AX.25 packets can be transmitted successfully. For successful reception, however, the frequency separation of the received signal must be greater than or equal to twice the bitrate[2]. Thus, with a separation of 1 kHz, it is impossible to transmit a signal faster than 500 bits per second.

  In order to work around this limitation, there are two solutions: increasing the frequency separation or decreasing the bitrate. Increasing the frequency separation was untenable, due to limitations on the audio filters of the radio: in order to obtain a 1200 bps bitrate, an frequency separation of at least 3 kHz was necessary, and at this bandwidth the edges of the radio filters begin to affect the signal. Decreasing the bitrate was a reasonable solution, since the uplink commands are not bandwidth or speed limited in any way (uplink commands are very short packets). Thus a value of 600 bits per second was chosen, since it is the lowest bitrate that the CC1000 supports. In order to support this lower bitrate, the frequency separation on receive was increased to 2 kHz.

B.4 MixW2 Software TNC

- Inability to receive 600 bps AX.25 data streams without balanced preamble

  Using a 600 bps uplink is nonstandard, and the only way to do it is with a software TNC on the groundstation called MixW2. Using the soundcard of a PC, MixW2 will encode data into AX.25 format and transfer it to the radio so that it can be transmitted. When transmitting, however, the MixW2 program does not begin transmission with a DC balanced preamble[3]. Every hardware TNC that was tested transmits a DC balanced

[2]This fact was not published in any data sheet available from Chipcon, but was obtained from email dialogs with a technician from Chipcon.

[3]A DC balanced preamble is a requirement for any modern digital phase-locked-loop PLL based communications device. A DC balanced preamble is simply a string of alternating bits (1010101 ...) that are

preamble, however this software TNC does not. Without a DC balanced preamble, the averaging filter of the CC1000 cannot properly decode the data that it receives, and thus cannot receive any valid data.

As a possible solution to this issue, the MixW2 authors were asked to implement a DC balanced preamble in their next revision of the software, and Denis Nechitailov (UU9JDR) was extremely helpful in modifying his code to do just that.

With the updated version of MixW2 in place, a successful uplink was finally obtained!

transmitted before any actual data, in order to help the receiver PLL lock on to the proper timing of each bit.

Appendix C

AX.25 Specification

AX.25 is an amateur radio specification which describes how to encode digital data in order to transmit it over radio frequencies. The AX.25 specification mandates a bitrate of 1200 baud and uses AFSK (audio frequency shift keying) encoding to represent binary values 0 and 1 with audio tones of 1200 Hz and 2200 Hz, respectively.

AX.25 is a full featured, stateful communications protocol that has been used for many years by ham radio enthusiasts worldwide. The latest revision of the official AX.25 specification is available at `http://www.tapr.org/tapr/html/Fax25.html`.

Appendix D

Communication Protocol

When in orbit, CP2 is programmed to broadcast a CW (morse code) beacon to assist in tracking the satellite and obtaining basic status information. Immediately following this CW beacon, an AX.25 data packet containing telemetry and sensor readings is also sent. This data packet gives more detailed information on the health and well-being of the satellite. This appendix describes the formats used to exchange data with the satellite while it is in orbit.

D.1 CW Beacon Data Format

CP2 is programmed to broadcast a CW beacon at a set interval of either 2 or 5 minutes, depending on its current mode of operation (Pre-Ops or Normal Ops). This CW beacon contains rough status information about the health of the satellite. The CW is generated at approximately 15 wpm, using a 1200 Hz tone[1]. The CW beacon contains 5 characters, which encode temperature, voltage and other satellite status as detailed in Table D.1, below.

Field Name	**Value(s)**	**Description**
Header	Fixed: CP	Say Hello
Temperature	Variable: A-Z	Encode average satellite temperature, with 5 deg. resolution per letter, A = -30 deg C and Z = 100 deg C
Voltage	Variable: A-Z	Encode average satellite voltage, with 75 mV resolution, A = 3V and Z = 5V

Table D.1: CW Beacon Format

D.2 Telemetry Beacon Downlink Data Format

To be determined when CP2 design is finalized.

D.3 Uplink Protocol and Data Formats

To be determined when CP2 design is finalized.

[1]This awkward tone is a side effect of the using the CC1000 to transmit AX.25 data as well as perform CW generation

Appendix E

Hardware Schematics

The schematics for the most current revision of the Command and Data Handling electronics board of the CP2 Cubesat are included in this section. Schematic pages one through five detail the general electronics of the Cubesat and pages six through ten detail the electronics of the communication subsystem.

Table E.1: Schematic Pages

For the complete set of schematics, including front panels, side panels and the power distribution boards, see [8].

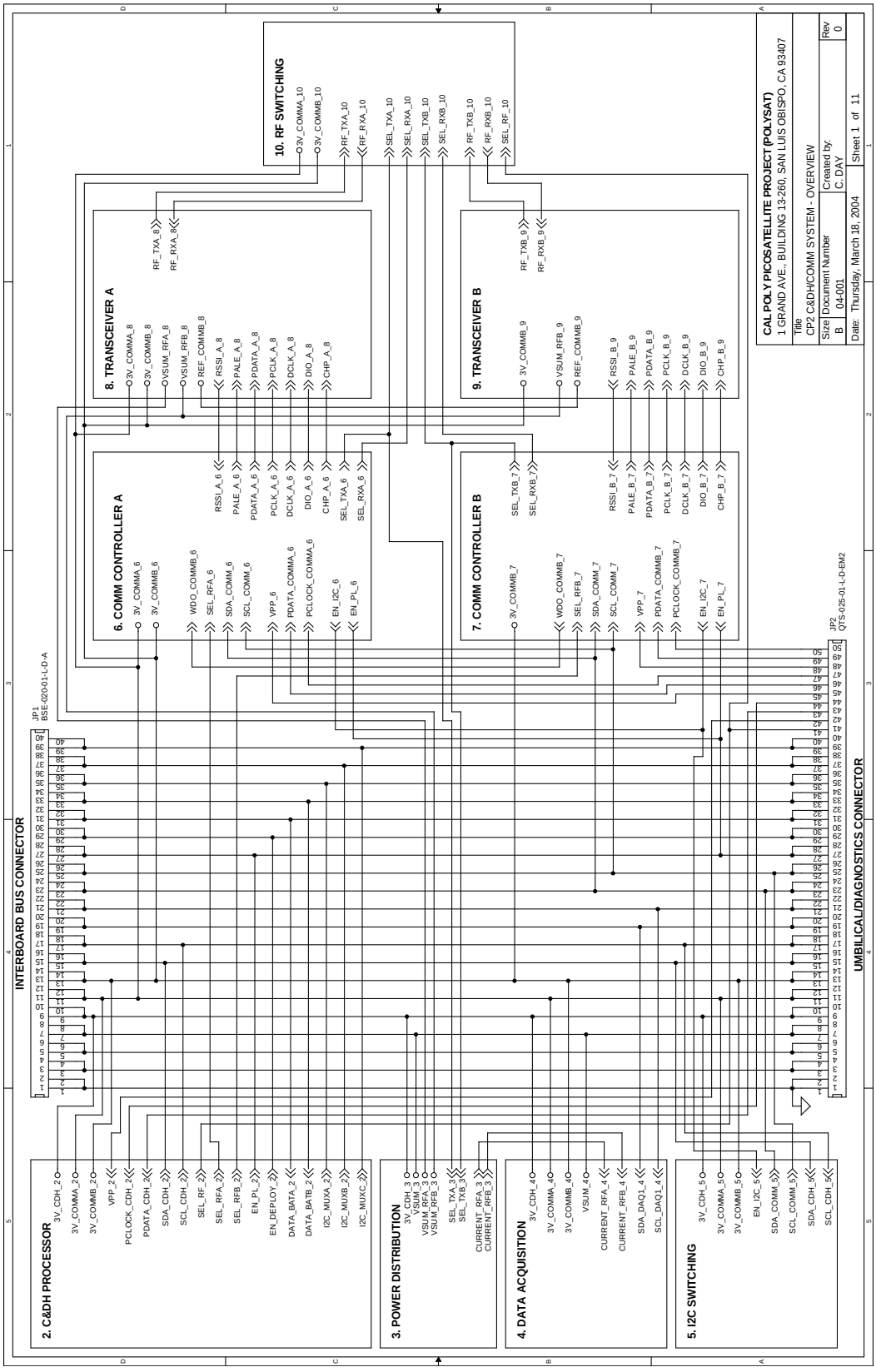


Figure E.1: Hardware Schematic: Command and Data Handling (CDH) Block Diagram

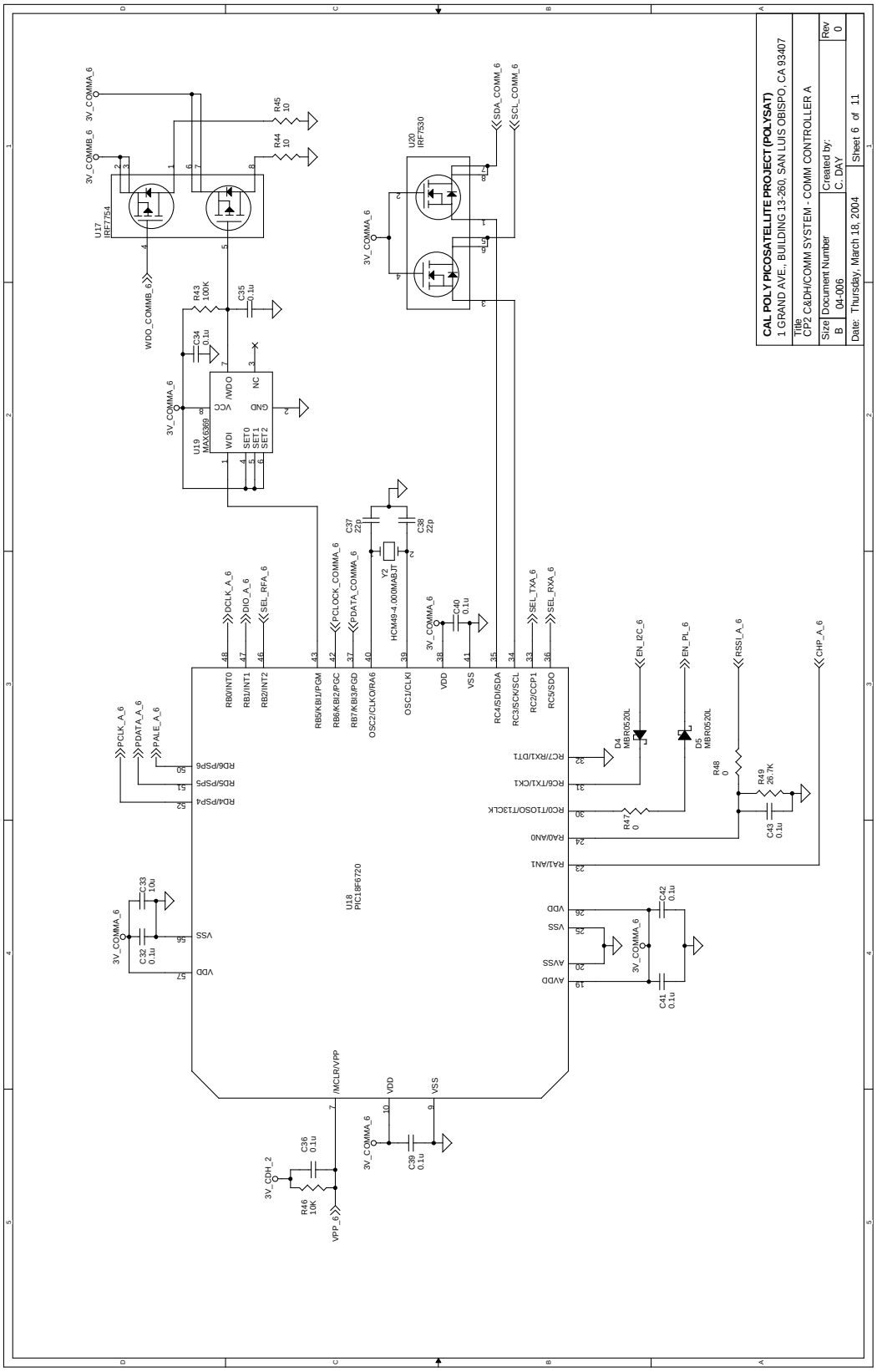


Figure E.2: Hardware Schematic: Communications Controller A

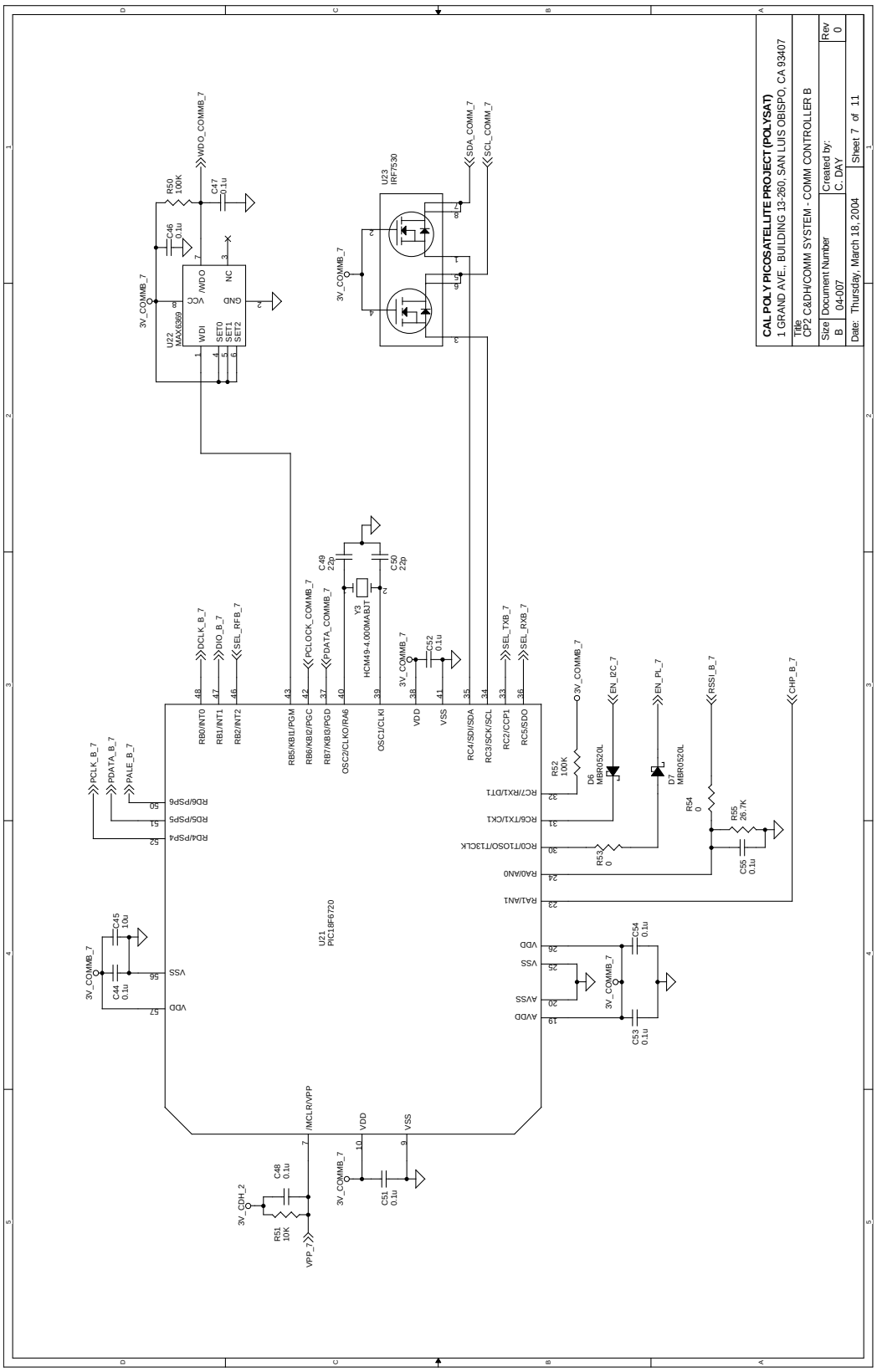


Figure E.3: Hardware Schematic: Communications Controller B

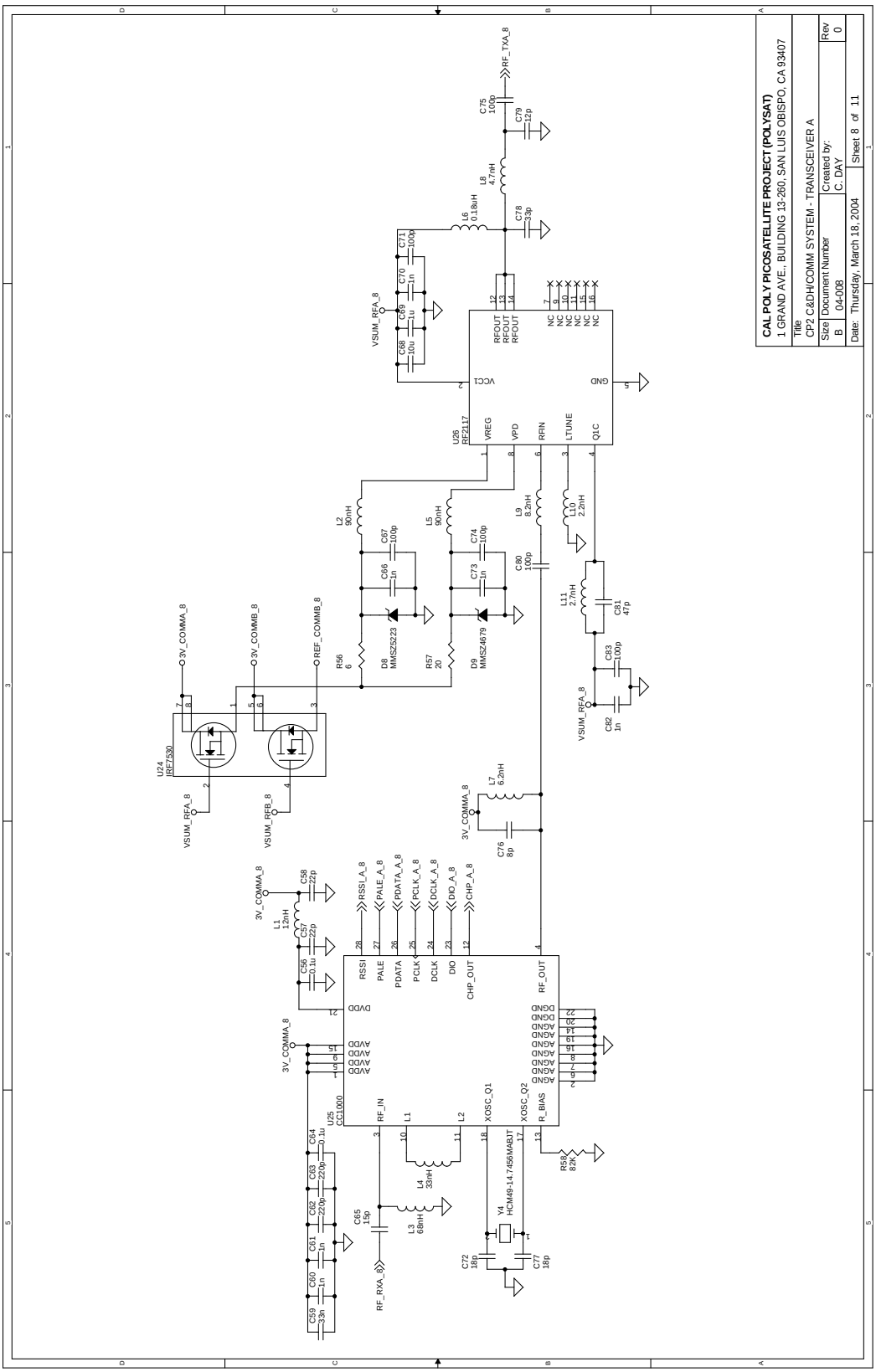


Figure E.4: Hardware Schematic: Transceiver A

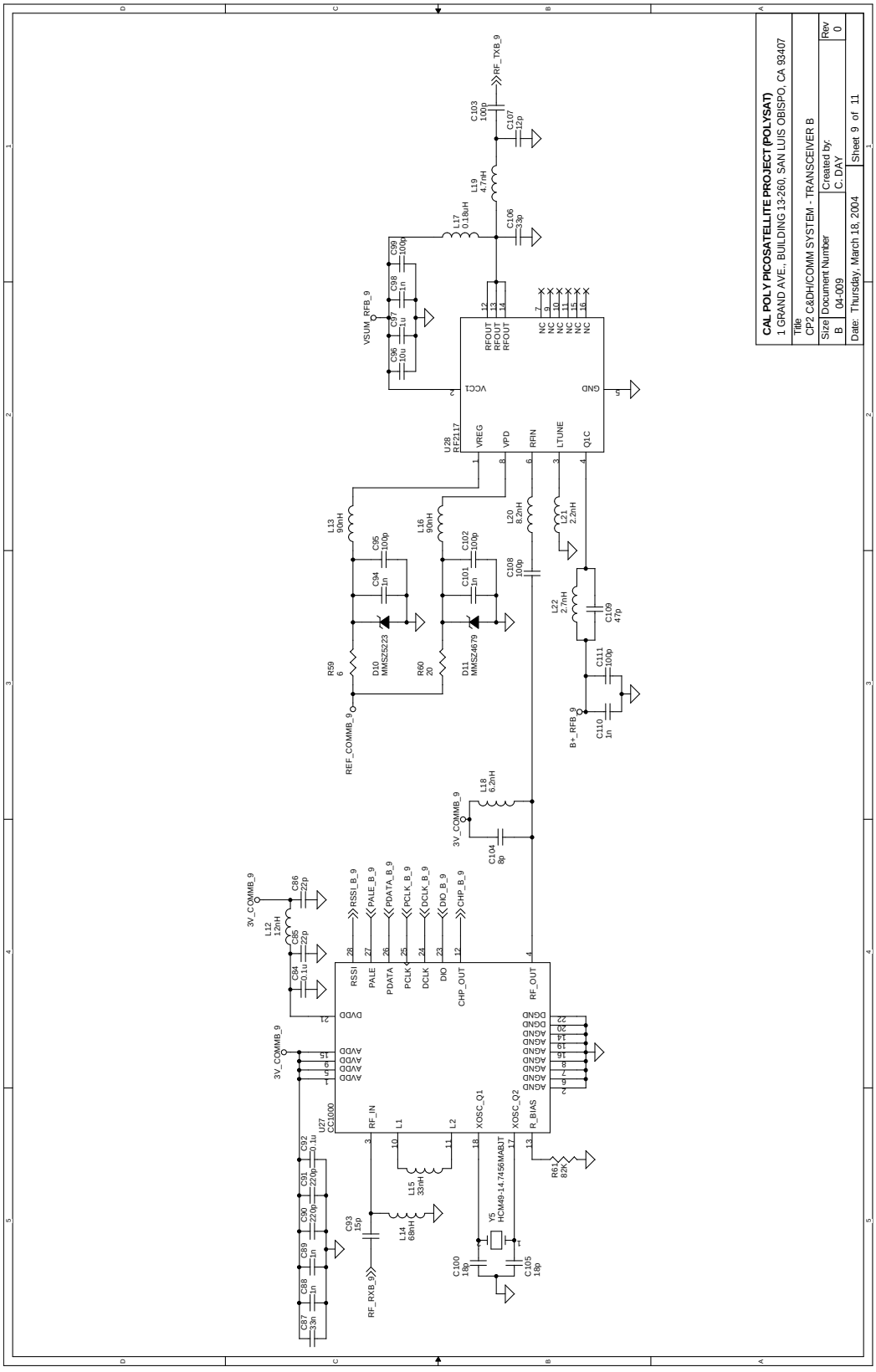


Figure E.5: Hardware Schematic: Transceiver B

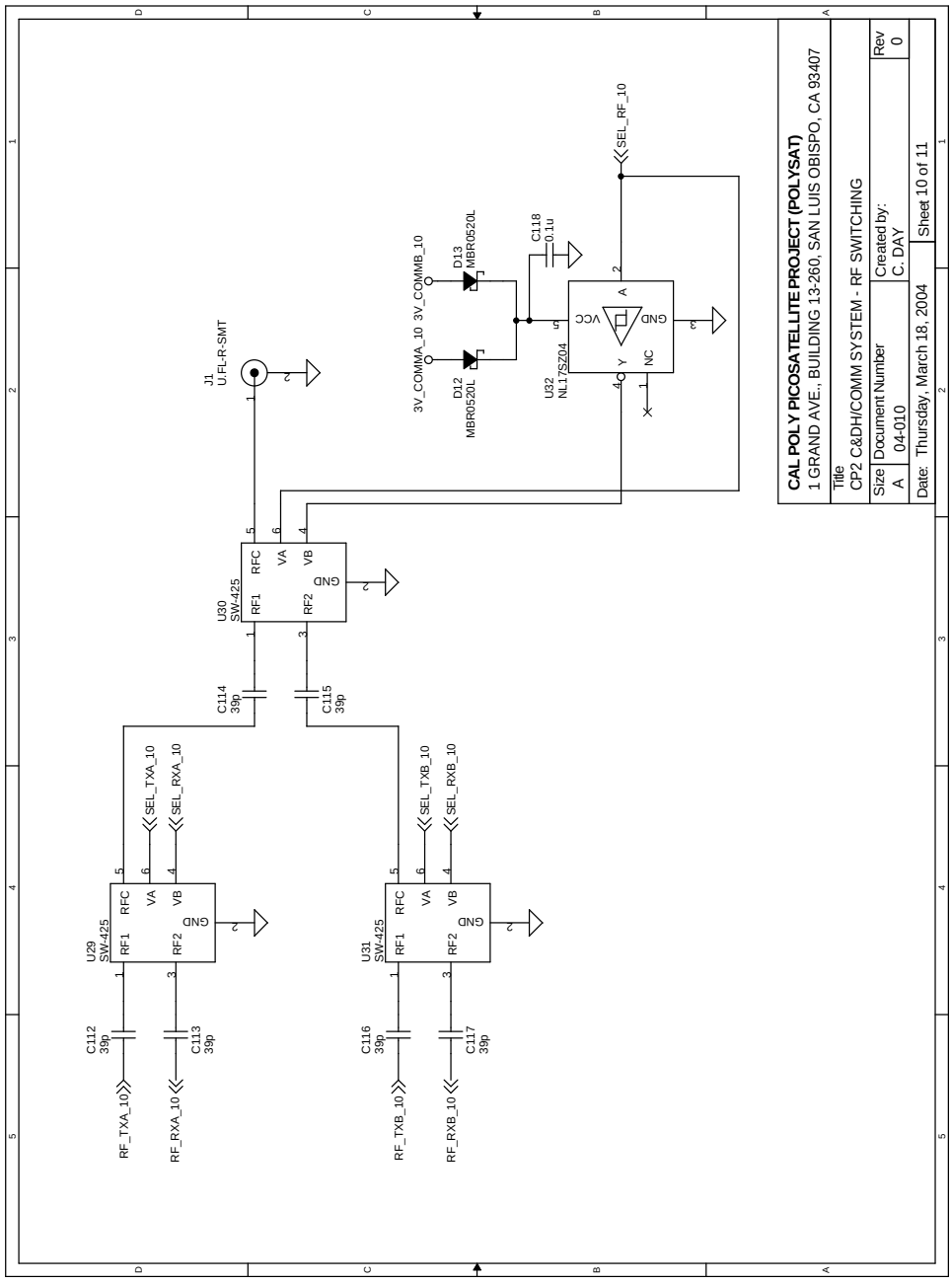


Figure E.6: Hardware Schematic: RF Switching

Appendix F

Software API

Main Communications Control (comm.c)

Function	Description
commSetBeaconInterval()	Set time between beacons
portInit()	Initialize input/output ports
timer0ISR()	ISR to handle beacon timer expiration

Table F.1: Software API, comm.c

Main Bus Interface (i2c-comm.c)

Function	Description
i2cInit()	Initialize I^2C ports and registers
i2cISR()	ISR that implements an I^2C response state machine

Table F.2: Software API, i2c-comm.c

CC1000 Interface (cc1000.c)

Function	Description
cc1000Calibrate()	Calibrate both RX and TX modes
cc1000Init()	Program CC1000 registers and switch to RX mode
cc1000PowerDown()	Put CC1000 into low power mode
cc1000ReadRegister()	Read data from CC1000 config register
cc1000Reset()	Reset CC1000
cc1000SetupRX()	Switch to RX mode
cc1000SetupTX()	Switch to TX mode
cc1000WriteRegister()	Write data to CC1000 config register

Table F.3: Software API, cc1000.c

Morse Code Beacon (cw.c)

Function	Description
cwSendChar()	Transmit ASCII character in morse code
cwSendDash()	Transmit morse code dash
cwSendDot()	Transmit morse code dot
cwSendString()	Transmit ASCII string in morse code
cwTransmitBeacon()	Transmit morse code status beacon

Table F.4: Software API, cw..c

Software TNC (tnc.c)

Function	Description
tncFCS()	Calculate Frame Check Sequence (FCS)
tncISR()	ISR that implements the TNC state machine
tncReset()	Reset TNC state machine
tncTxByte()	Place a byte in the transmit buffer
tncTxPacket()	Transmit packet from transmit buffer
tncVerifyPacket()	Check address and verify FCS of received packet

Table F.5: Software API, tnc.c

Appendix G

Source Code

G.1 Copyrights

All source code for the CP2 communications subsystem is Copyright (C) 2004, Chris Noe, unless otherwise stated within the source code itself. The source code is available under the terms of the GNU Public Licence, Version 2 (GPLv2).

```
Copyright (C) 2004 Chris Noe <cnoe@calpoly.edu>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

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

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

The file `beacon.c`, from Michael Gray's MicroBeaconII, served as the base of our current `tnc.c` file and was the motivation behind our state machine design. The original header from the `beacon.c` file has been removed from `tnc.c` for brevity, but is reproduced in full, below.

```
//  Revision History:
//
//   M. Gray   25 Sep 2001  V1.00  Initial release.  Flew ANSR-3 and ANSR-4.
//
//   M. Gray    5 Dec 2001  V1.01  Changed startup message and applied #SEPARATE
//                                 pragma to several methods for memory usage.
//
//   M. Gray   26 Oct 2002  V2.00  Micro Beacon II hardware changes including
//                                 PIC18F252 processor, serial EEPROM, GPS power
//                                 control, additional ADC input, and LM60
//                                 temperature sensor.
//
//  COPYRIGHT (c) 2001-2002 Michael Gray, KD7LMO
```

G.2 Main Communications Control

Main Communications Control (comm.h)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* $Id: comm.h,v 1.6 2004/06/01 18:56:10 cnoe Exp $ */

/* comm.h */

#ifndef __COMM_H
#define __COMM_H

#include <delays.h>
#include <p18cxxx.h>
#include <portb.h>

#include "boolean.h"
#include "cc1000.h"
#include "cw.h"
#include "i2c-comm.h"
#include "tnc.h"

/* Comm State Machine modes */
#define COMM_PREOPS         0x00
#define COMM_NORMOPS        0x01
#define COMM_TX             0x02
#define COMM_VALIDATE_CMD   0x03
#define COMM_EXECUTE_CMD    0x04
#define COMM_CONTINGENCY    0x05

/* Comm Beacon Intervals w/ 4MHz XTAL, Prescaler = 32 */
// With PS = 32, 1 interrupt = 2,097,120 usec */
#define COMM_BEACON_PREOPS    57     /* 119,535,840 usec */
#define COMM_BEACON_NORMOPS   143    /*  299,888,160 usec */

// Intervals used for testing
//#define COMM_BEACON_2MIN    1       /* 2,097,120 usec */
//#define COMM_BEACON_PREOPS      10    /* preops 10 sec */
//#define COMM_BEACON_NORMOPS     15   /* normops 15 sec */

/* status byte masks: see I2C protocol doc */
#define COMM_STAT_CMD_RECVD 0x01    /* valid command received? */
#define COMM_STAT_XCVR_CAL  0x02    /* transceiver calibrated? */
#define COMM_STAT_XCVR_TEMP 0x04    /* transceiver temp good? */
#define COMM_STAT_XCVR_MODE 0x08    /* transceiver mode: RX=0,TX=1 */
#define COMM_STAT_SEL_TX    0x10    /* SEL_TX pin */
#define COMM_STAT_SEL_RX    0x20    /* SEL_RX pin */
#define COMM_STAT_EN_PL     0x40    /* EN_PL pin */
#define COMM_STAT_SEL_RF    0x80    /* SEL_RF pin */

/* port definitions */
#define COMM_ID PORTCbits.RC7       /* processor ID */

#define EN_PL   PORTCbits.RC0       /* payload enable */
#define EN_I2C  PORTCbits.RC6       /* contingency enable */

#define SEL_TX  PORTCbits.RC2       /* antenna switching */
#define SEL_RX  PORTCbits.RC5
#define SEL_RF  PORTBbits.RB2

#define RSSI    PORTAbits.RA0       /* received signal strength */
#define CHP_OUT PORTAbits.RA1       /* CC1000 status out */

void commSetStatusBit(unsigned char b);
void commClearStatusBit(unsigned char b);
void timer0ISR(void);
void commTransmitBeacon(void);
#endif
```

Main Communications Control (comm.c)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* $Id: comm.c,v 1.13 2004/06/01 07:54:55 cnoe Exp $ */

#include "comm.h"
```

```
/* Comm status byte */
unsigned char commStatus = 0;

/* Comm state machine */
unsigned char commCurrentState = COMM_PREOPS;
unsigned char commNextState = COMM_PREOPS;

/* Comm beacon control */
static volatile unsigned char commBeaconWaiting = false;
static volatile unsigned char commBeaconTimer = 0;
static volatile unsigned char commBeaconDelayAmount = COMM_BEACON_PREOPS;

/* Contingency Mode Detection */
extern unsigned char i2cActivityDetect;

/* Current TNC mode (tnc.c) */
extern volatile unsigned char tncCurrentState;
extern volatile unsigned char tncNextState;

/* Clear a status bit */
void commClearStatusBit(unsigned char b)
{
    commStatus &= ~b;
}

/* Set a status bit */
void commSetStatusBit(unsigned char b)
{
    commStatus |= b;
}

/* Set interval for transmitting cw beacon */
void commSetBeaconInterval(unsigned char interval)
{
    // Set new timer comparison value
    commBeaconDelayAmount = interval;

    // ... and reset timer
    commBeaconTimer = 0;
}

/* Initialize port input/output */
void portInit(void)
{
    // RA7..2 (input) = unused
    // RA1 (input) = CHP_OUT, CC1000
    // RA0 (input, analog) = received signal strength, CC1000
    TRISA = 0b11111111;

    // RB7..6 (input) = unused (PGC/PGD programming pins)
    // RB5 (output) = watchdog pulse
    // RB4..3 (input) = unused
    // RB2 (input) = SEL_RFA (comm A or B active)
    // RB1 (bi-di) = DIO (CC1000 data I/O)
    // RB0 (input, interrupt) = DCLK (CC1000 data clock)
    TRISB = 0b11011111;

    // RC7 (input) = comm identifier (0 = A, 1 = B)
    // RC6 (output) = EN_I2C
    // RC5 (output) = SEL_RX
    // RC4 (input) = I2C SCL
    // RC3 (input) = I2C SDA
    // RC2 (output) = SEL_TX
    // RC1 (output) = XXX: CRC timing
    // RC0 (output) = XXX: TNC ISR timing
    TRISC = 0b10011000;

    // RD7 (input) = unused
    // RD6 (output) = PALE (CC1000 programming enable)
    // RD5 (bi-di) = PDATA (CC1000 programming data)
    // RD4 (output) = PCLK (CC1000 programming clock)
    // RD3..0 (input) = unused
    TRISD = 0b10001111;
}

void main()
{
    portInit();                     // init ports
    i2cInit();                      // init I2C
    tncReset(TNC_RX_PREAMBLE);      // init TNC
    cc1000Init();                   // init CC1000 into RX
```

```
    // init beacon timer
    OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_32);

    // Enable interrupts
    RCONbits.IPEN = 0;              // disable interrupt priority
    INTCONbits.PEIE = 1;            // enable peripheral interrupts
    PIR1bits.SSPIF = 0;             // clear I2C interrupt flag
    PIE1bits.SSPIE = 1;             // enable I2C interrupt
    INTCONbits.INT0F = 0;           // clear INT0 interrupt flag
    INTCONbits.TMR0IE = 1;          // enable timer0 (beacon) interrupt
    INTCONbits.GIE = 1;             // enable global interrupt

    // main loop
    while(1) {

        // First see if we are the active processor...
        // To find out, XOR processor ID pin with select pin.
        // Comm A ID = 0, Comm B ID = 1

        // Therefore when SEL_RF = 0, comm B will be the active processor
        // and when SEL_RF = 1, comm A is the active processor

        if (COMM_ID ^ SEL_RF) {
            switch (commCurrentState) {
                /* ------------------------------ */
                /* COMM_PREOPS: Pre-Operations    */
                /* ------------------------------ */
                case COMM_PREOPS:
                    /* is it time to beacon? */
                    if (commBeaconWaiting == true) {
                        cwTransmitBeacon();
                        tncTxPacket(PACKET_PREOPS);
                        commBeaconWaiting = false;
                    }

                    /* Stay in pre-ops until uplink command recv'd */
                    if (tncCurrentState == TNC_RX_COMPLETE) {
                        if (tncVerifyPacket()) {
                            /* transmit acknowledgement */
                            Delay10KTCYx(20);   // 2 second
                            tncTxPacket(PACKET_CMD_ACK);

                            /* On to normal ops ... */
                            commSetBeaconInterval(COMM_BEACON_NORMOPS);
                            commNextState = COMM_NORMOPS;
                        } else {
                            /* bad uplink command, reset */
                            tncReset(TNC_RX_PREAMBLE);
                        }
                    }
                    break;

                /* ------------------------------ */
                /* COMM_NORMOPS: Normal Operations */
                /* ------------------------------ */
                case COMM_NORMOPS:
                    /* is it time to beacon? */
                    if (commBeaconWaiting == true) {
                        cwTransmitBeacon();
                        tncTxPacket(PACKET_NORMOPS);
                        commBeaconWaiting = false;
                    }

                    /* have we received a packet? */
                    if (tncCurrentState == TNC_RX_COMPLETE) {
                        commNextState = COMM_VALIDATE_CMD;
                    }

                    break;

                /* ------------------------------ */
                /* COMM_TX: Transmit bulk data    */
                /* ------------------------------ */
                case COMM_TX:
                    /* bulk data is transmitted from I2C buffer */
                    break;

                /* ---------------------------------- */
                /* COMM_VALIDATE_CMD: Validate Command */
                /* ---------------------------------- */
```

```
                case COMM_VALIDATE_CMD:
                    /* is it a valid packet + command? */
                    if (tncVerifyPacket()) {
                        /* yes, execute */
                        commNextState = COMM_EXECUTE_CMD;
                    } else {
                        /* no, drop it */
                        tncReset(TNC_RX_PREAMBLE);
                        commNextState = COMM_NORMOPS;
                    }
                    break;

                /* ---------------------------------- */
                /* COMM_EXECUTE_CMD: Execute Command */
                /* ---------------------------------- */
                case COMM_EXECUTE_CMD:
                    /* for now, just transmit acknowledgement */
                    Delay10KTCYx(10);   // 1 second
                    tncTxPacket(PACKET_CMD_ACK);

                    /* Back to normal ops ... */
                    commNextState = COMM_NORMOPS;
                    break;

                /* ---------------------------------- */
                /* COMM_CONTINGENCY: Contingency Mode */
                /* ---------------------------------- */
                case COMM_CONTINGENCY:
                    Delay10KTCYx(0);    // 2.5 sec
                    tncTxPacket(PACKET_CONTINGENCY);

                    // has bus/processor come back to life?
                    if (i2cActivityDetect == true) {
                        commNextState = COMM_NORMOPS;
                    }
                    break;
            }

            if (commNextState != commCurrentState)
                commCurrentState = commNextState;
        } else {
            /* Processor is currently disabled */
            Delay10KTCYx(0);    // 2.5 sec
            tncTxPacket(PACKET_DISABLED);
        }
    }
}

/* Handle interrupts */
#pragma interrupt interruptHandling
void interruptHandling(void)
{
    if (PIR1bits.SSPIF)         // i2c address interrupt?
        i2cISR();

    if (INTCONbits.TMR0IF)      // beacon interrupt?
        timer0ISR();

    if (INTCONbits.INT0F)       // software tnc interrupt?
        tncISR();
}

/* Set interrupt vector */
#pragma code low_vector=0x18
void isr(void) { _asm GOTO interruptHandling _endasm }
#pragma code

/* Timer0 interrupt occurs every 2.097 sec */
/* Used for beacon and contingency mode detection */
#pragma interrupt timer0ISR
void timer0ISR(void)
{
    /* Beacon timer expired? */
    if (++commBeaconTimer == commBeaconDelayAmount) {
        commBeaconWaiting = true;
        commBeaconTimer = 0;
    }

    /* Contingency mode check */
    if (i2cActivityDetect == false) {
        // No, enter contingency mode
```

```
        commNextState = COMM_CONTINGENCY;
    } else {
        // Yes, reset activity detector
        i2cActivityDetect = false;
    }

    // Clear interrupt flag
    INTCONbits.TMR0IF = 0;
}

#pragma code
```

G.3 Main Bus Interface

Main Bus Interface (i2c-comm.h)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* $Id: i2c-comm.h,v 1.2 2004/06/01 18:56:10 cnoe Exp $ */

/* i2c-comm.h */

#ifndef __I2C_COMM_H
#define __I2C_COMM_H

#include <i2c.h>
#include <p18cxxx.h>

#include "cc1000.h"
#include "comm.h"

/* I2C slave address */
#define I2C_ADDR_COMM           0x00

/* define comm I2C commands */
#define I2C_COMM_STATUS         0xDD

#define I2C_COMM_SENSORS        0x01
#define I2C_COMM_CC1K_DUMP      0x02
#define I2C_COMM_CC1K_ON        0x03
#define I2C_COMM_CC1K_OFF       0x04
#define I2C_COMM_TX_AX25        0x05
#define I2C_COMM_TX_BEACON      0x06
#define I2C_COMM_GET_DATA       0x07

#define I2C_COMM_STATUS_SIZE    1
#define I2C_COMM_SENSORS_SIZE   10
#define I2C_COMM_CC1K_DUMP_SIZE 29
#define I2C_COMM_CC1K_ON_SIZE   0
#define I2C_COMM_CC1K_OFF_SIZE  0
#define I2C_COMM_TX_AX25_SIZE   255
#define I2C_COMM_TX_BEACON_SIZE 90

/* I2C buffer */
#define I2C_BUF_MAX         256

/* I2C ISR */
void i2cInit(void);
void i2cISR(void);

#endif
```

Main Bus Interface (i2c-comm.c)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* $Id: i2c-comm.c,v 1.3 2004/06/01 07:54:55 cnoe Exp $ */

#include "i2c-comm.h"

#pragma udata I2CBUF
static unsigned char i2cBuffer[I2C_BUF_MAX];
#pragma udata

extern unsigned char commStatus;    /* comm status byte (comm.c) */
```

```
unsigned char i2cActivityDetect;    /* any i2c transactions yet? */

/* Initialize I2C module */
void i2cInit(void)
{
    TRISC |= 0b00011000;        // Set SDA/SCL as inputs
    SSPADD = I2C_ADDR_COMM;     // Set slave address

    OpenI2C (SLAVE_7, SLEW_OFF);
    IdleI2C();

    SSPCON1bits.CKP = 1;        // Ensure clock is released
    SSPSTATbits.CKE = 0;        // Disable SMBus specifics
    SSPCON2bits.SEN = 1;        // Enable clock stretching
}

/* I2C address match interrupt */
#pragma interrupt i2cISR
void i2cISR(void)
{
    static unsigned char i2cBufferIndex;    // Index into current I2C buffer
    static unsigned char i2cCommand;        // Last I2C command recv'd
    static unsigned char commandReceived;   // Have we received a command?

    unsigned char i;
    unsigned char data;

    // Record that we've received an i2c request
    i2cActivityDetect = true;

    // Examine S, RW, DA and BF to determine I2C state
    switch (SSPSTAT & 0x2D) {
        // -----------------------------------------------
        // State 1: Master Write, previous byte was address
        // -----------------------------------------------
        // S = 1, RW = 0, DA = 0, BF = 1
        case 0x09:
            i2cBufferIndex = 0;         // Reset buffer index
            i2cCommand = 0;             // Reset last command
            commandReceived = 0;        // Reset cmd recvd
            i = 0;                      // Reset loop iterator
            data = SSPBUF;              // Dummy read SSPBUF to clear BF
            break;

        // -----------------------------------------------
        // State 2: Master Write, previous byte was data
        // S = 1, RW = 0, DA = 1, BF = 1
        // -----------------------------------------------
        case 0x29:
            /* Store command byte separately from data */
            if (commandReceived) {
                i2cBuffer[i2cBufferIndex++] = SSPBUF;
            } else {
                commandReceived = true;     // Command received
                i2cCommand = SSPBUF;        // Store command
            }
            break;

        // -----------------------------------------------
        // State 3: Master Read, previous byte was address
        // S = 1, RW = 1, DA = 0, BF = 0
        // -----------------------------------------------
        // Description: Return length of command data
        // -----------------------------------------------
        case 0x0C:
            switch (i2cCommand) {
                case I2C_COMM_STATUS:
                data = I2C_COMM_STATUS_SIZE;
                break;

                case I2C_COMM_SENSORS:
                data = I2C_COMM_SENSORS_SIZE;
                break;

                case I2C_COMM_CC1K_DUMP:
                data = I2C_COMM_CC1K_DUMP_SIZE;
                break;

                case I2C_COMM_CC1K_ON:
                data = I2C_COMM_CC1K_ON_SIZE;
                break;
```

```
                case I2C_COMM_CC1K_OFF:
                data = I2C_COMM_CC1K_OFF_SIZE;
                break;

                case I2C_COMM_TX_AX25:
                data = I2C_COMM_TX_AX25_SIZE;
                break;

                case I2C_COMM_TX_BEACON:
                data = I2C_COMM_TX_BEACON_SIZE;
                break;

                case I2C_COMM_GET_DATA:
                data = 0xFF;            /* Return size of received packet */
                break;

                default:
                break;
            }

            while (SSPSTATbits.BF)      /* Wait for xmit buffer to empty */
            { /* XXX: need to add timeout */ }

            SSPBUF = data;              /* Buffer next byte */
            break;

        // ----------------------------------------------
        // State 4: Master Read, previous byte was data
        // S = 1, RW = 1, DA = 1, BF = 0
        // ----------------------------------------------
        case 0x2C:
            /* Return command data, if any */
            switch (i2cCommand) {
                case I2C_COMM_STATUS:
                /* Update current pin values */
                commStatus |= (SEL_TX << 4);
                commStatus |= (SEL_RX << 5);
                commStatus |= (SEL_RF << 6);
                commStatus |= (EN_PL  << 7);
                data = commStatus;
                break;

                case I2C_COMM_SENSORS:
                break;

                case I2C_COMM_CC1K_DUMP:
                if (i < I2C_COMM_CC1K_DUMP_SIZE) {
                    data = cc1000ReadRegister(i++);
                } else {
                    i = 0;
                }
                break;

                case I2C_COMM_CC1K_ON:
                break;

                case I2C_COMM_CC1K_OFF:
                break;

                case I2C_COMM_TX_AX25:
                break;

                case I2C_COMM_TX_BEACON:
                break;

                default:
                break;
            }

            while (SSPSTATbits.BF)      /* Wait for xmit buffer to empty */
            { /* XXX: need to add timeout */ }

            SSPBUF = data;              /* Buffer next byte */
            break;

        // ----------------------------------------------
        // State 5: Master NACK
        // S = 1, RW = 0, DA = 1, BF = 0
        // ----------------------------------------------
        case 0x28:
```

```
            i2cCommand = 0;                 // Reset command
            commandReceived = 0;            // Reset cmd recvd
            i2cBufferIndex = 0;             // Reset index
            i = 0;                          // Reset loop iterator
            break;
    }

    /* Release SCL to free the bus */
    SSPCON1bits.CKP = 1;

    // Clear interrupt flag
    PIR1bits.SSPIF = 0;
}
#pragma code
```

G.4 Morse Code Beacon

Morse Code Beacon (cw.h)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* $Id: cw.h,v 1.6 2004/06/01 18:56:10 cnoe Exp $ */

/* cw.h */

#ifndef __CW_H
#define __CW_H

/* CW parameters */

/* pitch = 600 Hz wanted, 1700Hz is what we're currently at*/
/* 15 wpm */
/* dot = 90 msec */
/* dash = 3*dot */
/* interchar delay = 3*dot */

#include <delays.h>
#include <string.h>
#include <timers.h>

#include "boolean.h"
#include "cc1000.h"
#include "comm.h"

#define CW_DOT_TIME 9
#define CW_OUT_PIN  PORTBbits.RB1

unsigned char asciiToCw(unsigned char ascii);
void cwSendDot(void);
void cwSendDash(void);
void cwSendChar(unsigned char a);
void cwSendString(unsigned char *string);
void cwEndWord(void);
void cwTransmitBeacon(void);

#endif
```

Morse Code Beacon (cw.c)

```
/* Authors: Kevin McCray, Steven Lyuee 10/15/2003 */
/* Modified by Chris Noe <cnoe@calpoly.edu> for integration into CP2 */
/* $Id: cw.c,v 1.8 2004/06/01 18:56:10 cnoe Exp $ */

#include "cw.h"

/* 8 bits: XXXX XNNN */
/* NNN = number of morse dits or dahs */
/* XXXXX = the actual morse pattern, 0 = dit, 1 = dah */
/* starts with MSB */
const unsigned char asciiToCwTable[] = {0x42, 0x84, 0xA4, 0x83, 0x01, 0x24,
                                        0xC3, 0x04, 0x02, 0x74, 0xA3, 0x44,
                                        0xC2, 0x82, 0xE3, 0x64, 0xD4, 0x43,
                                        0x03, 0x81, 0x23, 0x14, 0x63, 0x94,
```

```
                                        0xB4, 0xC4};

unsigned char asciiToCw(unsigned char ascii)
{
    return asciiToCwTable[ascii - 65];
}

/* Send dot */
void cwSendDot(void)
{
    // Set to low tone
    CW_OUT_PIN = 1;

    // Turn on power amp
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|RESET_N);

    // Wait 1 Dot time
    Delay10KTCYx(CW_DOT_TIME);

    // Turn off power amp
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|TX_PD|RESET_N);
}

/* Send dash */
void cwSendDash(void)
{
    // Set to low tone
    CW_OUT_PIN = 1;

    // Turn on power amp
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|RESET_N);

    // Wait 1 dash time = 3 * dot time
    Delay10KTCYx(3 * CW_DOT_TIME);

    // Turn off power amp
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|TX_PD|RESET_N);
}

/* Send character in morse code */
void cwSendChar(unsigned char ch)
{
    unsigned char a = asciiToCw(ch);
    unsigned char n = a & 0x07, j;

    for(j = 0; j < n; j++) {
        if((0x80 & a) != 0)
            cwSendDash();
        else
            cwSendDot();

        a = a << 1;

        Delay10KTCYx(CW_DOT_TIME);
    }

    // Inter character space
    Delay10KTCYx(3 * CW_DOT_TIME);
}

/* Pause between cw words */
void cwEndWord(void)
{
    // Wait 7 dot times (3 from last character + 4)
    Delay10KTCYx(4 * CW_DOT_TIME);

    // Turn off power amp
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|TX_PD|RESET_N);
}

/* Transmit a full CW beacon */
void cwTransmitBeacon()
{
    cc1000SetupTX();

    // Turn off power amp
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|TX_PD|RESET_N);

    cwSendChar('C');
    cwSendChar('P');
    cwSendChar('A');
```

```
    cwSendChar('B');
    cwSendChar('C');
    cwEndWord();

    // Turn off power amp
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|TX_PD|RESET_N);
}
```

G.5 Transceiver Interface

Transceiver Interface (cc1000.h)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* $Id: cc1000.h,v 1.11 2004/06/01 18:56:10 cnoe Exp $ */

/* cc1000.h */

#ifndef __CC1000_H
#define __CC1000_H

#include <adc.h>
#include <p18cxxx.h>
#include <delays.h>

#include "boolean.h"
#include "comm.h"

/* Constants defined for CC1000 */
/* output pins */
#define PCLK         PORTDbits.RD4
#define PDATA        PORTDbits.RD5
#define PDATA_DIR    TRISDbits.TRISD5
#define PDATA_PORT   PORTD
#define PALE         PORTDbits.RD6

/* CC1000 configuration registers */
#define CC1000_MAIN             0x00
#define CC1000_FREQ_2A          0x01
#define CC1000_FREQ_1A          0x02
#define CC1000_FREQ_0A          0x03
#define CC1000_FREQ_2B          0x04
#define CC1000_FREQ_1B          0x05
#define CC1000_FREQ_0B          0x06
#define CC1000_FSEP1            0x07
#define CC1000_FSEP0            0x08
#define CC1000_CURRENT          0x09
#define CC1000_FRONT_END        0x0A
#define CC1000_PA_POW           0x0B
#define CC1000_PLL              0x0C
#define CC1000_LOCK             0x0D
#define CC1000_CAL              0x0E
#define CC1000_MODEM2           0x0F
#define CC1000_MODEM1           0x10
#define CC1000_MODEM0           0x11
#define CC1000_MATCH            0x12
#define CC1000_FSCTRL           0x13
#define CC1000_FSHAPE7          0x14
#define CC1000_FSHAPE6          0x15
#define CC1000_FSHAPE5          0x16
#define CC1000_FSHAPE4          0x17
#define CC1000_FSHAPE3          0x18
#define CC1000_FSHAPE2          0x19
#define CC1000_FSHAPE1          0x1A
#define CC1000_FSDELAY          0x1B
#define CC1000_PRESCALER        0x1C
#define CC1000_TEST6            0x40
#define CC1000_TEST5            0x41
#define CC1000_TEST4            0x42
#define CC1000_TEST3            0x43
#define CC1000_TEST2            0x44
#define CC1000_TEST1            0x45
#define CC1000_TEST0            0x46

/* register values that differ in RX/TX other than FREQ & FSEP */
#define CC1000_MAIN_TX      0xE1
#define CC1000_MAIN_RX      0x11
```

```
#define CC1000_CURRENT_TX   0x81
#define CC1000_CURRENT_RX   0x44

#define CC1000_PLL_TX       0x48        // REFDIV = 9
#define CC1000_PLL_RX       0x68        // REFDIV = 13

/* 2 is the MSB register */
/* working @ 437.414.588 LSB */
#define CC1000_RX_FREQ_2    0x60
#define CC1000_RX_FREQ_1    0x40
#define CC1000_RX_FREQ_0    0x00

/* need values @ 437.484 LSB */
//#define CC1000_RX_FREQ_2    0x
//#define CC1000_RX_FREQ_1    0x
//#define CC1000_RX_FREQ_0    0x

/* working @ 437.414.588 LSB */
#define CC1000_TX_FREQ_2    0x42
#define CC1000_TX_FREQ_1    0x9E
#define CC1000_TX_FREQ_0    0x75

/* working @ 437.484.833 LSB */
//#define CC1000_TX_FREQ_2    0x42
//#define CC1000_TX_FREQ_1    0xA1
//#define CC1000_TX_FREQ_0    0x2A
//#define CC1000_TX_FREQ_0    0x42

// Shortest filter lock time is 14 bits (=2 bytes) of balanced preamble
// This is the least accurate filter locking interval
#define CC1000_UNLOCK_FILTER    0x09    // unlock avg filter
#define CC1000_LOCK_FILTER      0x19    // lock avg filter

// Longest filter lock time is 89 bits (=12 bytes) of balanced preamble
// This is the most accurate filter locking interval
//#define CC1000_UNLOCK_FILTER  0x0F    // unlock avg filter
//#define CC1000_LOCK_FILTER    0x1F    // lock avg filter

// Output power, see data sheet
#define CC1000_TX_POWER     0xFF

/* CC1000 register fields */
/* MAIN */
#define RXTX        0x80
#define F_REG       0x40
#define RX_PD       0x20
#define TX_PD       0x10
#define FS_PD       0x08
#define CORE_PD     0x04
#define BIAS_PD     0x02
#define RESET_N     0x01

/* CAL */
#define CAL_START       0x80
#define CAL_DUAL        0x40
#define CAL_WAIT        0x20
#define CAL_CURRENT     0x10
#define CAL_COMPLETE    0x08
#define CAL_ITERATE     0x06

/* Functions for accessing the CC1000 */
void cc1000Calibrate(void);
void cc1000Init(void);
void cc1000PowerDown(void);
void cc1000Reset(void);
void cc1000SetupRX(void);
void cc1000SetupTX(void);
void cc1000WriteRegister(unsigned char addr, unsigned char data);
void cc1000WriteRegisterWord(unsigned int addranddata);
unsigned char cc1000ReadRegister(unsigned char addr);

#endif
```

Transceiver Interface (cc1000.c)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* Based on AN009 from Chipcon website */
```

```
/* $Id: cc1000.c,v 1.11 2004/06/01 18:56:10 cnoe Exp $ */

#include "cc1000.h"

/* Write to a single cc1000 register */
void cc1000WriteRegister(unsigned char addr, unsigned char data)
{
    unsigned int val;

    val = (unsigned int) (addr & 0x7F) << 9 | (unsigned int) data & 0x00FF;
    cc1000WriteRegisterWord(val);
}

/* Write to a cc1000 register with register/data in a single 8 bit word */
void cc1000WriteRegisterWord(unsigned int addr_data)
{
    unsigned char addr, data, mask;

    addr = (unsigned char) ((addr_data & 0xFE00) >> 9); // high 7 bits
    data = (unsigned char) (addr_data & 0x00FF);        // low 8 bits

    PALE = 1;
    PALE = 0;

    PDATA_DIR = 0;          // enable PDATA as an output

    // Send address bits (7)
    for (mask = 0b01000000; mask != 0; mask >>= 1) {
        PCLK = 1;

        if (addr & mask)
            PDATA = 1;
        else
            PDATA = 0;

        PCLK = 0;
    }

    // Send read/write bit
    // Ignore bit in data, always use 1
    PCLK = 1;
    PDATA = 1;
    PCLK = 0;

    PCLK = 1;
    PALE = 1;

    // Send data bits
    for (mask = 0b10000000; mask != 0; mask >>= 1) {
        PCLK = 1;

        if (data & mask)
            PDATA = 1;
        else
            PDATA = 0;

        PCLK = 0;
    }

  PCLK = 1;

}

/* Read from a single cc1000 register */
unsigned char cc1000ReadRegister(unsigned char addr)
{
    volatile unsigned char data, mask, bitcount;

    PALE = 1;
    PALE = 0;
    PDATA_DIR = 0;          // enable PDATA as an output

    // Send address bits (7)
    for (mask = 0b01000000; mask != 0; mask >>= 1) {
        PCLK = 1;

        if (addr & mask)
            PDATA = 1;
        else
            PDATA = 0;
```

```
        PCLK = 0;
    }

    // Clear read/write bit
    // Ignore bit in data, always use 0
    PCLK = 1;
    PDATA = 0;
    PCLK = 0;

    PCLK = 1;
    PALE = 1;

    PDATA_DIR = 1;          // enable PDATA as an input

    // Now receive data bits
    for (bitcount = 0; bitcount < 8; bitcount++) {
        PCLK = 1;
        data = (data << 1) | PDATA;
        PCLK = 0;
    }

    PDATA_DIR = 0;          // Set PDATA back to an output
    PCLK = 1;

    return data;
}

/* Reset CC1000, clearing all registers */
void cc1000Reset(void)
{
    unsigned char main;

    main = cc1000ReadRegister(CC1000_MAIN);

    // Reset
    cc1000WriteRegister(CC1000_MAIN, main & 0xFE);
    cc1000WriteRegister(CC1000_MAIN, main | 0x01);

    return;
}

// cc1000PowerDown: put CC1000 into power down mode
void cc1000PowerDown(void)
{
    cc1000WriteRegister(CC1000_MAIN, RX_PD|TX_PD|FS_PD|BIAS_PD);
    cc1000WriteRegister(CC1000_PA_POW, 0x00);

    return;
}

// cc1000Init: initialize CC1000, switch to receive mode
void cc1000Init(void)
{
    INTCON2bits.RBPU = 1;   // disable port b pullups

    INTCONbits.INT0IF = 0;
    INTCONbits.INT0IE = 1;  // enable interrupt on INT0 (CC1K clock)

    // See p26 of data sheet for details on initialization
    cc1000WriteRegister(CC1000_MAIN, RX_PD|TX_PD|FS_PD|BIAS_PD);

    // Reset
    cc1000Reset();

    // Wait 2ms, do 3
    Delay1KTCYx(3);

    // Program all registers except main
    cc1000WriteRegister(CC1000_FREQ_2A, CC1000_RX_FREQ_2);
    cc1000WriteRegister(CC1000_FREQ_1A, CC1000_RX_FREQ_1);
    cc1000WriteRegister(CC1000_FREQ_0A, CC1000_RX_FREQ_0);

    cc1000WriteRegister(CC1000_FREQ_2B, CC1000_TX_FREQ_2);
    cc1000WriteRegister(CC1000_FREQ_1B, CC1000_TX_FREQ_1);
    cc1000WriteRegister(CC1000_FREQ_0B, CC1000_TX_FREQ_0);

    cc1000WriteRegister(CC1000_FSEP1, 0x00);
    cc1000WriteRegister(CC1000_FSEP0, 0x14);

    cc1000WriteRegister(CC1000_CURRENT, CC1000_CURRENT_RX);
    cc1000WriteRegister(CC1000_FRONT_END, 0x12);
```

```
    cc1000WriteRegister(CC1000_PLL, CC1000_PLL_RX);
    cc1000WriteRegister(CC1000_LOCK, 0x10);
    cc1000WriteRegister(CC1000_CAL, 0x26);

    cc1000WriteRegister(CC1000_MODEM2, 0x8B);
    cc1000WriteRegister(CC1000_MODEM1, CC1000_UNLOCK_FILTER);
    cc1000WriteRegister(CC1000_MODEM0, 0x03);

    cc1000WriteRegister(CC1000_MATCH, 0x70);
    cc1000WriteRegister(CC1000_FSCTRL, 0x01);
    cc1000WriteRegister(CC1000_PRESCALER, 0x00);

    // Calibrate
    cc1000Calibrate();

    // Power Down
    cc1000WriteRegister(CC1000_MAIN,
                        RX_PD|TX_PD|FS_PD|CORE_PD|BIAS_PD|RESET_N);

    cc1000WriteRegister(CC1000_PA_POW, 0x00);

    // Now power oscillator back up
    cc1000WriteRegister(CC1000_MAIN, RX_PD|TX_PD|FS_PD|BIAS_PD|RESET_N);

    // Wait 2ms, for oscillator to settle..
    Delay1KTCYx(2);

    // Now power bias generator back up
    cc1000WriteRegister(CC1000_MAIN, RX_PD|TX_PD|FS_PD|RESET_N);

    // Wait 200us to settle
    Delay10TCYx(20);

    // Switch to RX
    cc1000SetupRX();

    // set up A/D on CC1000 RSSI signal
    OpenADC(ADC_FOSC_2 & ADC_RIGHT_JUST & ADC_1ANA,
        ADC_CH0 & ADC_INT_OFF & ADC_VREFPLUS_VDD & ADC_VREFMINUS_VSS);

    return;
}

// Calibrate CC1000 (RX and TX)
void cc1000Calibrate(void)
{
    // Clear calibrated flag
    commClearStatusBit(COMM_STAT_XCVR_CAL);

    // Clear CAL_DUAL
    cc1000WriteRegister(CC1000_CAL, 0x00);

    // See p23 of data sheet for calibration flow chart
    cc1000WriteRegister(CC1000_FREQ_2A, CC1000_RX_FREQ_2);
    cc1000WriteRegister(CC1000_FREQ_1A, CC1000_RX_FREQ_1);
    cc1000WriteRegister(CC1000_FREQ_0A, CC1000_RX_FREQ_0);

    cc1000WriteRegister(CC1000_FREQ_2B, CC1000_TX_FREQ_2);
    cc1000WriteRegister(CC1000_FREQ_1B, CC1000_TX_FREQ_1);
    cc1000WriteRegister(CC1000_FREQ_0B, CC1000_TX_FREQ_0);

    // Calibrate RX
    cc1000WriteRegister(CC1000_MAIN, TX_PD|RESET_N);

    // Set VCO_CURRENT
    cc1000WriteRegister(CC1000_CURRENT, CC1000_CURRENT_RX);

    // Set CAL_START
    cc1000WriteRegister(CC1000_CAL, CAL_START|CAL_WAIT|CAL_ITERATE);

    // Wait 34 ms = 17,000 inst cycles, round up for safety
    Delay1KTCYx(40);

    // Reset CAL_START
    cc1000WriteRegister(CC1000_CAL, CAL_WAIT|CAL_ITERATE);

    // Calibrate TX
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|RESET_N);

    // Set VCO_CURRENT and PLL for TX
    cc1000WriteRegister(CC1000_CURRENT, CC1000_CURRENT_TX);
```

```
    cc1000WriteRegister(CC1000_PLL, CC1000_PLL_TX);

    // Turn off PA to avoid spurious transmission
    cc1000WriteRegister(CC1000_PA_POW, 0x00);

    // Set CAL_START
    cc1000WriteRegister(CC1000_CAL, CAL_START|CAL_WAIT|CAL_ITERATE);

    // Wait max 28 ms = 14,000 inst cycles, round up for safety
    Delay1KTCYx(40);

    // Reset CAL_START
    cc1000WriteRegister(CC1000_CAL, CAL_WAIT|CAL_ITERATE);

    // set calibrated flag
    commSetStatusBit(COMM_STAT_XCVR_CAL);

    return;
}

/* Switch to RX mode */
void cc1000SetupRX(void)
{
    // Set DIO as input
    TRISBbits.TRISB1 = 1;

    // Registers for RX 600 baud, 2kHz fsep that differ from TX
    cc1000WriteRegister(CC1000_FSEP0, 0x14);
    cc1000WriteRegister(CC1000_MODEM2, 0x8B);
    cc1000WriteRegister(CC1000_MODEM0, 0x03);
    cc1000WriteRegister(CC1000_PLL, CC1000_PLL_RX);

    // Power down RX & TX
    cc1000WriteRegister(CC1000_MAIN, RX_PD|TX_PD|RESET_N);

    // Set RX CURRENT register
    cc1000WriteRegister(CC1000_CURRENT, CC1000_CURRENT_RX);

    // Wait 250us...
    Delay10TCYx(26);

    // Power up RX
    cc1000WriteRegister(CC1000_MAIN, CC1000_MAIN_RX);

    // Set RB0 to interrupt on rising edge of CC1K clock
    INTCONbits.INT0IF = 0;
    INTCON2bits.INTEDG0 = 1;

    // Update comm status byte
    commClearStatusBit(COMM_STAT_XCVR_MODE);

    return;
}

/* Switch CC1000 into TX mode */
void cc1000SetupTX(void)
{
    // Set DIO as output
    TRISBbits.TRISB1 = 0;

    // Turn off power amp
    cc1000WriteRegister(CC1000_PA_POW, 0x00);

    // Registers for TX @1200 baud, 1kHz sep, that differ from RX
    cc1000WriteRegister(CC1000_FSEP0, 0x0A);
    cc1000WriteRegister(CC1000_MODEM2, 0x8A);
    cc1000WriteRegister(CC1000_MODEM0, 0x13);
    cc1000WriteRegister(CC1000_PLL, CC1000_PLL_TX);

    // Power up TX
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|RESET_N);
    cc1000WriteRegister(CC1000_CURRENT, CC1000_CURRENT_TX);

    // Wait 250us...
    Delay10TCYx(26);

    // Set output power
    cc1000WriteRegister(CC1000_PA_POW, CC1000_TX_POWER);

    // Wait 20us...
    Delay10TCYx(2);
```

```
    // Set RB0 to interrupt on falling edge of CC1K clock
    INTCONbits.INT0IF = 0;
    INTCON2bits.INTEDG0 = 0;

    // Update comm status byte
    commSetStatusBit(COMM_STAT_XCVR_MODE);

    return;
}
```

G.6 Software TNC

Software TNC (tnc.h)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* $Id: tnc.h,v 1.13 2004/06/01 18:56:10 cnoe Exp $ */

/* tnc.h */

#ifndef __TNC_H
#define __TNC_H

#include <delays.h>
#include <p18cxxx.h>
#include <stdlib.h>
#include <string.h>

#include "boolean.h"
#include "cc1000.h"

// AX.25 Flag Byte
#define AX25_FLAG       0x7E

// Map I/O names to the hardware pins.
#define IO_PACKET_TX    PORTBbits.RB1
#define IO_PACKET_RX    PORTBbits.RB1

// Modes for the packet state machine.
#define PACKET_PREOPS       0x10
#define PACKET_NORMOPS      0x11
#define PACKET_BEACON       0x12
#define PACKET_CONTINGENCY  0x13
#define PACKET_REPEATER     0x14
#define PACKET_CMD_ACK      0x15
#define PACKET_DISABLED     0x16

// The number of start flag bytes to send before the packet message.
#define TNC_TX_DELAY    10

// Modes for the TNC Rx state machine.
#define TNC_RX_PREAMBLE 0x01        // detect preamble + first flag
#define TNC_RX_DATA     0x02        // information field
#define TNC_RX_WAIT2    0x03        // handle repeated closing flags
#define TNC_RX_COMPLETE 0x04        // packet is complete

// Modes for the TNC Tx state machine.
#define TNC_TX_PREAMBLE 0x80
#define TNC_TX_SYNC     0x81
#define TNC_TX_HEADER   0x82
#define TNC_TX_DATA     0x83
#define TNC_TX_FCS      0x84
#define TNC_TX_END      0x85
#define TNC_TX_COMPLETE 0x86

// Mask to determine if we are transmitting.
#define TNC_TX_MODE     0x80

// The size of the TNC Rx packet buffer.
#define TNC_RX_BUFFER_SIZE 256

// The size of the TNC output buffer.
#define TNC_TX_BUFFER_SIZE 256

// When we receive 5 1's followed by a 0, we need to un-bitstuff the zero.
// This mask tells us when it occurs.
```

```
#define TNC_UNSTUFF_MASK    0x3E

// Minimum number of preamble flags to receive before moving to next state.
#define TNC_RX_MIN_FLAGS    4

// Minimum number of preamble bits we must detect before we believe it.
#define TNC_MIN_PREAMBLE_BITS   8
#define TNC_MAX_PREAMBLE_ERRORS 10

// Length of TX preamble, in 10 msec units.
#define TNC_TX_PREAMBLE_LENGTH  30

// this inline asm is a workaround for a C compiler bug... this line of code:
// tncRxBitStuff = ((tncRxBitStuff << 1) | tncRxCurDataBit) & 0x3F;
// compiles to code that doesn't work correctly..
#define UPDATE_TNCRXBITSTUFF()              \
            _asm                            \
            MOVLB   0                       \
            RLNCF   tncRxBitStuff, 0, 1     \
            ANDLW   0xFE                    \
            IORWF   tncRxCurDataBit, 0, 1   \
            ANDLW   0x3F                    \
            MOVWF   tncRxBitStuff, 1        \
            _endasm

// Precomputed CRC (FCS) value for the static header.
#define TNC_AX25_HEADER_FCS     0x5B20

// Size of the static header.
#define TNC_AX25_HEADER_SIZE    (sizeof(TNC_AX25_HEADER) - 1)

void tncReset(unsigned char nextState);
unsigned int tncFCS(volatile char *buffer, unsigned int length, unsigned int fcs);
void tncTxByte (unsigned char value);
void tncTxPacket(unsigned char tncTxPacketType);
boolean tncVerifyPacket(void);
void tncISR(void);

#endif
```

Software TNC (tnc.c)

```
/* Chris Noe <cnoe@calpoly.edu> */
/* based on beacon.c, used with permission, courtesy of Michael Gray, KD7LMO */
/* $Id: tnc.c,v 1.16 2004/06/01 11:14:01 cnoe Exp $ */

#include "tnc.h"

// Comm status byte
extern unsigned char commStatus;

// TNC shared state variables
volatile unsigned char tncCurrentState = TNC_RX_PREAMBLE;
volatile unsigned char tncNextState = TNC_RX_PREAMBLE;

// TNC Tx state machine variables
volatile unsigned char tncTxLastBit, tncTxShift;
volatile unsigned char tncTxBitCount, tncTxBitStuff;
volatile unsigned int tncTxFCS;

// XXX: ints used where chars should do... this is easily fixed, do it
volatile unsigned int tncTxLength, tncTxIndex;

// This needs to be a char pointer for memcmp() functions
volatile char *tncTxBufferPnt;

// TNC Rx state machine variables
volatile unsigned char tncRxPreambleDetected, tncRxPreambleCount;
volatile unsigned char tncRxPreambleErrors, tncRxLength;
volatile unsigned char tncRxCurDataBit, tncRxLastBit, tncRxByte;
volatile unsigned char tncRxBitCount, tncRxBitStuff, tncRxFlagCount;
volatile unsigned char tncRxLastPacketLength;

// needs to be char pointer for memcmp() functions
volatile char *tncRxBufferPnt;

// TNC Rx state machine counts
volatile unsigned char tncRxBadPacketCount;
```

```
#pragma udata AX25TX
volatile char tncTxBuffer[TNC_TX_BUFFER_SIZE];
#pragma code

#pragma udata AX25RX
volatile char tncRxBuffer[TNC_RX_BUFFER_SIZE];
#pragma code

const rom char TNC_AX25_HEADER[] = {
    'N' << 1, '6' << 1, 'C' << 1, 'P' << 1, ' ' << 1, ' ' << 1, 0xE0, \     // dest
    'N' << 1, '6' << 1, 'C' << 1, 'P' << 1, ' ' << 1, ' ' << 1, 0x61, \     // source
    0x03, 0xf0 };

#pragma interrupt tncISR
void tncISR(void)
{
    if (tncCurrentState & TNC_TX_MODE) {
        // Transmit current bit
        if (tncTxLastBit == 0)
            IO_PACKET_TX = 0;
        else
            IO_PACKET_TX = 1;
    } else {
        /* cnoe: workaround INT0 interrupt edge bug */
        /* description: when configured to interrupt only on the rising edge of INT0, */
        /* eg INTCON2.INTEDG0=0, INTCON.INT0IE=1, INTCON.INT0IF=0, the PIC still gets */
        /* interrupted on the falling edge (in the simulator)! (PIC18LF6720) */

        /* software workaround: on falling edge, RB0 will read as zero, so detect and */
        /* exit the ISR in this case */
        if (PORTBbits.RB0 == 0)
            goto tncInterruptComplete;

        // Receive bit
        // NRZI decode the incoming bit.. note: we NRZI decode *everything*
        // including the preamble! so in order to detect preamble, check for
        // a string of all (0x00) instead of (0x55 || 0xAA)
        if (tncRxLastBit ^ IO_PACKET_RX)
            tncRxCurDataBit = 0;
        else
            tncRxCurDataBit = 1;

        tncRxLastBit = IO_PACKET_RX;
    }

    // ------------------
    // TNC State Machines
    // ------------------
    switch (tncCurrentState) {
        // ----------------------------
        // TNC Receive State Machine
        // ----------------------------
        case TNC_RX_PREAMBLE:
            /* 0x55 = 0101 0101 and 0xAA = 1010 1010 */
            /* both are signs of possible valid preamble */
            /* but since we are NRZI encoded, both cases become 0x00! */
            tncRxByte = ((tncRxByte << 1) | tncRxCurDataBit);

            /* have we detected a minimal amount of preamble? */
            if (tncRxPreambleDetected == true) {
                /* check for AX25 flag */
                if (tncRxByte == AX25_FLAG) {
                    /* received AX.25 flag, now we're in the data field */
                    tncRxByte = 0;
                    tncRxBitCount = 0;
                    tncRxBitStuff = 0;
                    tncRxLength = 0;
                    tncRxBufferPnt = tncRxBuffer;
                    tncNextState = TNC_RX_DATA;
                } else if (tncRxByte == 0x00) {
                    /* still receiving preamble bits */
                    tncRxPreambleCount++;
                } else if (tncRxPreambleErrors == 0) {
                    tncRxPreambleErrors++;
                }

                if (tncRxPreambleErrors > 0) {
                    tncRxPreambleErrors++;
                }
```

```
                if (tncRxPreambleErrors > TNC_MAX_PREAMBLE_ERRORS) {
                    /* too many errors... reset detection and avg filter */
                    tncRxPreambleDetected = false;
                    cc1000WriteRegister(CC1000_MODEM1, CC1000_UNLOCK_FILTER);
                }
            } else {
                /* still no valid preamble detected... keep looking */
                if (tncRxByte == 0x00) {
                    tncRxPreambleCount++;
                } else {
                    tncRxPreambleCount = 0;
                }

                // detected enough preamble that we're confident
                // this is an actual transmission?
                if (tncRxPreambleCount >= TNC_MIN_PREAMBLE_BITS) {
                    // yes, lock the averaging filter now
                    cc1000WriteRegister(CC1000_MODEM1, CC1000_LOCK_FILTER);
                    tncRxPreambleErrors = 0;
                    tncRxPreambleDetected = true;
                }
            }
            break;

        case TNC_RX_DATA:
            /* receiving packet data */
            /* determine if we need to unstuff a zero */
            UPDATE_TNCRXBITSTUFF();

            /* if tncRxBitStuff = 0x3E, we need to drop the current zero bit */
            if (tncRxBitStuff == TNC_UNSTUFF_MASK)
            goto tncInterruptComplete;

            /* build receive byte */
            tncRxByte = tncRxByte | (tncRxCurDataBit << tncRxBitCount);

            /* full byte yet? */
            if (++tncRxBitCount == 8) {
                tncRxBitCount = 0;

                /* is it the ending flag? */
                if (tncRxByte == AX25_FLAG) {
                    /* yes, check for repeated closing flags */
                    tncNextState = TNC_RX_WAIT2;
                } else {
                    /* store received byte */
                    *tncRxBufferPnt++ = tncRxByte;
                    tncRxLength++;
                    tncRxByte = 0;

                    /* XXX: sanity check the buffer pointer */
                    if (tncRxBufferPnt - tncRxBuffer == TNC_RX_BUFFER_SIZE - 1) {
                        // buffer is full with bad info, drop and restart
                        tncRxBadPacketCount++;
                        tncReset(TNC_RX_PREAMBLE);
                    }
                }
            }
            break;

        case TNC_RX_WAIT2:
            /* waiting for repeated closing flags */
            tncRxByte = tncRxByte | (tncRxCurDataBit << tncRxBitCount);

            if (++tncRxBitCount == 8) {
                tncRxBitCount = 0;
                /* is it another flag? */
                if (tncRxByte == AX25_FLAG) {
                    /* drop it */
                    tncRxByte = 0;
                } else {
                    /* end of reception... unlock averaging filter */
                    cc1000WriteRegister(CC1000_MODEM1, CC1000_UNLOCK_FILTER);

                    tncRxLastPacketLength = tncRxBufferPnt - tncRxBuffer;

                    /* no more closing flags, packet is complete */
                    tncNextState = TNC_RX_COMPLETE;
                }
            }
            break;
```

```
200         case TNC_RX_COMPLETE:
                /* packet complete, do nothing */
                break;

            // ----------------------------
            // TNC Transmit State Machine
            // ----------------------------
            case TNC_TX_PREAMBLE:
                // Generate a test signal for setting levels.  This
            // alternates the 1200 and 2220 Hz tone every bit time.
210             if (tncTxLastBit == 0)
                    tncTxLastBit = 1;
                else
                    tncTxLastBit = 0;
                break;

            case TNC_TX_SYNC:
                // The variable tncTxShift contains the lastest data byte.
                // NRZI enocde the data stream.
                if ((tncTxShift & 0x01) == 0x00)
220                 if (tncTxLastBit == 0)
                        tncTxLastBit = 1;
                    else
                        tncTxLastBit = 0;

                // When the flag is done, determine if we need to send more or data.
                if (++tncTxBitCount == 8) {
                    tncTxBitCount = 0;
                    tncTxShift = 0x7e;

230                 // Once we transmit x mS of flags, send the data.
                    // txDelay bytes * 8 bits/byte * 833uS/bit = x mS
                    if (++tncTxIndex == TNC_TX_DELAY) {
                        tncTxIndex = 0;
                        tncTxShift = TNC_AX25_HEADER[0];
                        tncTxBitStuff = 0;
                        tncNextState = TNC_TX_HEADER;
                    }
                } else
                    tncTxShift = tncTxShift >> 1;
240             break;

            case TNC_TX_HEADER:
                // Determine if we have sent 5 ones in a row, if we have send a zero.
                if (tncTxBitStuff == 0x1f) {
                    if (tncTxLastBit == 0)
                        tncTxLastBit = 1;
                    else
                        tncTxLastBit = 0;

250                 tncTxBitStuff = 0x00;
                    goto tncInterruptComplete;
                }

                // The variable tncTxShift contains the lastest data byte.
                // NRZI enocde the data stream.
                if ((tncTxShift & 0x01) == 0x00)
                    if (tncTxLastBit == 0)
                        tncTxLastBit = 1;
                    else
260                     tncTxLastBit = 0;

                // Save the data stream so we can determine if bit stuffing is
                // required on the next bit time.
                tncTxBitStuff = ((tncTxBitStuff << 1) | (tncTxShift & 0x01)) & 0x1f;

                // If all the bits were shifted, get the next byte.
                if (++tncTxBitCount == 8) {
                    tncTxBitCount = 0;

270                 // After the header is sent, then send the data.
                    if (++tncTxIndex == sizeof(TNC_AX25_HEADER)) {
                        tncTxIndex = 0;
                        tncTxShift = tncTxBuffer[0];
                        tncNextState = TNC_TX_DATA;
                    } else
                        tncTxShift = TNC_AX25_HEADER[tncTxIndex];
                } else
                    tncTxShift = tncTxShift >> 1;
                break;
```

```
280
        case TNC_TX_DATA:
            // Determine if we have sent 5 ones in a row; if we have, send a zero.
            if (tncTxBitStuff == 0x1f) {
                if (tncTxLastBit == 0)
                    tncTxLastBit = 1;
                else
                    tncTxLastBit = 0;

                tncTxBitStuff = 0x00;
290             goto tncInterruptComplete;
            }

            // The variable tncTxShift contains the lastest data byte.
            // NRZI enocde the data stream.
            if ((tncTxShift & 0x01) == 0x00)
                if (tncTxLastBit == 0)
                    tncTxLastBit = 1;
                else
                    tncTxLastBit = 0;
300
            // Save the data stream so we can determine if bit stuffing is
            // required on the next bit time.
            tncTxBitStuff = ((tncTxBitStuff << 1) | (tncTxShift & 0x01)) & 0x1f;

            // If all the bits were shifted, get the next byte.
            if (++tncTxBitCount == 8) {
                tncTxBitCount = 0;

                // have we reached the end of the packet?
310             if (++tncTxIndex == tncTxLength) {
                    tncTxIndex = 0;
                    tncTxShift = tncTxFCS & 0xff;   // send low byte of FCS
                    tncNextState = TNC_TX_FCS;
                } else
                    tncTxShift = tncTxBuffer[tncTxIndex];
            } else
                tncTxShift = tncTxShift >> 1;

            break;
320
        case TNC_TX_FCS:
            // Determine if we have sent 5 ones in a row; if we have, send a zero.
            if (tncTxBitStuff == 0x1f) {
                if (tncTxLastBit == 0)
                    tncTxLastBit = 1;
                else
                    tncTxLastBit = 0;

                tncTxBitStuff = 0x00;
330             goto tncInterruptComplete;
            }

            // The variable tncTxShift contains the lastest data byte.
            // NRZI enocde the data stream.
            if ((tncTxShift & 0x01) == 0x00)
                if (tncTxLastBit == 0)
                    tncTxLastBit = 1;
                else
                    tncTxLastBit = 0;
340
            // Save the data stream so we can determine if bit stuffing is
            // required on the next bit time.
            tncTxBitStuff = ((tncTxBitStuff << 1) | (tncTxShift & 0x01)) & 0x1f;

            // If all the bits were shifted, get the next byte.
            if (++tncTxBitCount == 8) {
                tncTxBitCount = 0;

                // Finished sending FCS yet?
350             if (++tncTxIndex == 2) {
                    // yes, start sending end flags
                    tncTxIndex = 0;
                    tncTxShift = 0x7e;
                    tncNextState = TNC_TX_END;
                } else
                    tncTxShift = (tncTxFCS >> 8) & 0xff;    // send high byte of FCS
            } else
                tncTxShift = tncTxShift >> 1;
            break;
360
```

```
        case TNC_TX_END:
            // The variable tncTxShift contains the lastest data byte.
            // NRZI enocde the data stream.
            if ((tncTxShift & 0x01) == 0x00)
                if (tncTxLastBit == 0)
                    tncTxLastBit = 1;
                else
                    tncTxLastBit = 0;

            // If all the bits were shifted, get the next one.
            if (++tncTxBitCount == 8) {
                tncTxBitCount = 0;
                tncTxShift = 0x7e;

                // Transmit two closing flags.
                if (++tncTxIndex == 2) {
                    // Switch back to receive mode to stop transmitting
                    TRISBbits.TRISB1 = 1;       // set DIO as input
                    cc1000WriteRegister(CC1000_MAIN, CC1000_MAIN_RX);
                    tncNextState = TNC_TX_COMPLETE;
                    goto tncInterruptComplete;
                }
            } else
                tncTxShift = tncTxShift >> 1;
            break;

        case TNC_TX_COMPLETE:
            /* packet complete, do nothing */
            break;
    }

    /* update state machine (if changed within ISR) */
    if (tncNextState != tncCurrentState)
        tncCurrentState = tncNextState;

tncInterruptComplete:
    // Clear interrupt flag
    INTCONbits.INT0IF = 0;
}
#pragma code

/* Reset everything, set TNC state = nextState */
void tncReset(unsigned char nextState)
{
    // Set DIO low
    IO_PACKET_TX = 0;

    tncRxPreambleDetected = false;
    tncRxPreambleCount = 0;
    tncRxCurDataBit = 0;
    tncRxLastBit = 0;
    tncRxByte = 0;
    tncRxLength = 0;
    tncRxBitCount = 0;
    tncRxBitStuff = 0;
    tncRxBufferPnt = tncRxBuffer;
    tncRxFlagCount = 0;

    tncTxLastBit = 0;
    tncTxBitCount = 0;

    // Make sure averaging filter is unlocked
    cc1000WriteRegister(CC1000_MODEM1, CC1000_UNLOCK_FILTER);

    tncCurrentState = tncNextState = nextState;
}

/* Calculate the FCS of a given chunk of memory */
unsigned int tncFCS(volatile char *buffer, unsigned int length, unsigned int fcs)
{
    unsigned int i;
    unsigned char bit, value;

    for (i = 0; i < length; i++) {
        value = buffer[i];

        for (bit = 0; bit < 8; ++bit) {
            fcs ^= (value & 0x01);
            fcs = ( fcs & 0x01 ) ? ( fcs >> 1 ) ^ 0x8408 : ( fcs >> 1 );
            value = value >> 1;
        }
```

```
    }

    return fcs ^ 0xffff;
}

/* Write a byte to the transmit buffer */
void tncTxByte(unsigned char value)
{
    *tncTxBufferPnt++ = value;

    // Bounds check: don't go above 256
    if (tncTxLength < TNC_TX_BUFFER_SIZE)
        tncTxLength++;
}

/* Transmit string from program memory */
void tncTxStringROM(const rom char *s)
{
    while (*s != NULL) {
        tncTxByte((unsigned char)*s++);
    }
}


// Transmit a packet of data
void tncTxPacket(unsigned char tncTxPacketType)
{
    unsigned int i;

    // Only transmit if we are in RX_PREAMBLE or RX_COMPLETE
    if (tncCurrentState != TNC_RX_PREAMBLE)
        if (tncCurrentState != TNC_RX_COMPLETE)
            return;

    // Make sure we do not begin to receive a new packet...
    tncReset(TNC_TX_PREAMBLE);

    // Set a pointer to our TNC output buffer.
    tncTxBufferPnt = tncTxBuffer;

    // Set the message length counter.
    tncTxLength = 0;

    // Determine packet type here
    switch (tncTxPacketType) {
        /* ---------------------------------- */
        /* PACKET_PREOPS: Transmit preops data */
        /* ---------------------------------- */
       case PACKET_PREOPS:
            tncTxStringROM("Preops");
            break;

        /* ---------------------------------------- */
        /* PACKET_NORMOPS: Transmit normal ops data */
        /* ---------------------------------------- */
       case PACKET_NORMOPS:
            tncTxStringROM("Normal Ops");
            break;

        /* ------------------------------------------------ */
        /* PACKET_CONTINGENCY: Transmit contingency mode data */
        /* ------------------------------------------------ */
        case PACKET_CONTINGENCY:
            tncTxStringROM("Contingency Mode");
            break;

        /* ------------------------------------------------- */
        /* PACKET_REPEATER: transmit back what we just received */
        /* ------------------------------------------------- */
        case PACKET_REPEATER:
            tncTxStringROM("Repeater: ");
            for (i = 16; i < tncRxLastPacketLength - 2; i++)
                tncTxByte(tncRxBuffer[i]);
            break;

        /* ------------------------------------------------ */
        /* PACKET_CMD_ACK: acknowledge received command      */
        /* ------------------------------------------------ */
        case PACKET_CMD_ACK:
            tncTxStringROM("Command Received: ");
```

```
            for (i = 16; i < tncRxLastPacketLength - 2; i++)
                tncTxByte(tncRxBuffer[i]);
            break;

        case PACKET_DISABLED:
            tncTxStringROM("Transceiver Disabled (not really)");
            break;

    }

    // Calculate the fcs for the message.. header fcs is precomputed
    tncTxFCS = tncFCS(tncTxBuffer, tncTxLength, TNC_AX25_HEADER_FCS ^ 0xffff);

    // Prepare for the ISR.
    tncTxBitCount = 0;
    tncTxShift = 0x7e;
    tncTxLastBit = 0;
    tncTxIndex = 0;

    // Begin transmission
    tncReset(TNC_TX_PREAMBLE);
    cc1000SetupTX();

    // Turn on power amp to begin transmitting preamble
    cc1000WriteRegister(CC1000_MAIN, RXTX|F_REG|RX_PD|RESET_N);

    // Transmit preamble for this long (in 10 msec increments)
    Delay10KTCYx(TNC_TX_PREAMBLE_LENGTH);

    // stop transmitting preamble, start flags and data
    tncNextState = TNC_TX_SYNC;

    // wait for transmission to complete
    while (tncCurrentState != TNC_TX_COMPLETE)
            { /* XXX: put a timeout on this */ }

    // Delay a small amount (6ms) to ensure that the CC1000
    // has time to finish transmission
    Delay1KTCYx(6);

    // Now switch back to RX
    tncReset(TNC_RX_PREAMBLE);
    cc1000SetupRX();
}

boolean tncVerifyPacket(void)
{
    unsigned int fcs;
    unsigned char fcs_high, fcs_low;

    boolean validPacket = false;

    // Does the header match that from our ground station?
    if (!memcmppgm2ram((char *)tncRxBuffer, (rom void *)TNC_AX25_HEADER, 16)) {
        // Calculate fcs (- 2 so that we do not include xmit'd FCS)
        fcs = tncFCS(tncRxBuffer, tncRxLength - 2, 0xffff);

        // Compare with xmit'd FCS (upper 8 bits first, then lower 8)
        fcs_high = fcs & 0xff;
        fcs_low = (fcs >> 8) & 0xff;

        if ((tncRxBuffer[tncRxLength - 2] != fcs_high) ||
                    (tncRxBuffer[tncRxLength - 1] != fcs_low))
            validPacket = false;
        else
            validPacket = true;
    }

    return validPacket;
}
```

Bibliography

[1] Cubesat Specification. Available at `http://cubesat.calpoly.edu/documents/cubesat_spec.pdf`.

[2] L. Stras, D.D. Kekez, G.J. Wells, T. Jeans, R.E. Zee, F.M. Pranajaya, and D.G. Foisy. The Design and Operation of The Canadian Advanced Nanospace eXperiment (CanX-1). *Proc. AMSAT-NA 21st Space Symposium, Toronto, Canada*, pages 150–160, October 2003.

[3] K. Nakaya, K. Konoue, H. Sawada, K. Ui, H. Okada, N. Miyashita, M. Iai, and S. Matunga. Tokyo Tech CubeSat: CUTE-I — Design and Development of Flight Model and Future Plan. *Proceedings of the 21st AIAA International Communications Satellite Systems Conference, Yokohoma, Japan*, 2003.

[4] Y. Tsuda, N. Sako, T. Eishima, T. Ito, Y. Arikawa, N. Miyamura, K. Kanairo, S. Ukawa, S. Ogasawara, S. Ishikawa, and S. Nakasuka. University of Tokyo's CubeSat XI as a Student-Built Educational Pico-Satellite – Final Design and Operation Plan. *23rd International Symposium of Space Technology and Science, Matsue, Japan*, 2002.

[5] J. H. Hales and M. Pedersen. Two-Axis MOEMS Sun Sensor for Pico Satellites. *16th Annual AIAA/USU Conference on Small Satellites, Utah, USA*, 2002.

[6] AAUSat Project Webpage. Available at `http://www.cubesat.auc.dk/`.

[7] K. Doerksen, J.F. Levesque, and I. Mas. Design, Modeling, and Evaluation of a 2.4GHz FHSS Communications System for NarcisSat. *17th Annual AIAA/USU Conference on Small Satellites, Logan, Utah*, August 2003.

[8] Polysat Website. Available at `http://polysat.calpoly.edu`.

[9] J. Schaffner and J. Puig-Suari. The Electronic System Design, Analysis, Integration, and Construction of the Cal Poly State University CP1 CubeSat. Available at `http://polysat.calpoly.edu/documents_cp1/systems/schaffner.pdf`.

[10] S. Lee. CubeSat Project Update January 2004. Available at `http://cubesat.calpoly.edu/documents/cubesat_update_2004_01.pdf`.

[11] I. Nason, J. Puig-Suari, and R. Twiggs. Development of the Standard CubeSat Deployer and a CubeSat Class Picosatellite. *Proceedings of the IEEE Aerospace Conference, Big Sky Montana*, 1:347–353, August 2001.

[12] AMSAT Organization, History Page. Available at `http://www.amsat.org/amsat/amsat-na/amhist.html`.

[13] W. J. Larson and J. R. Wertz. Space Mission Analysis and Design. pages 220,233,537, 1999.

[14] R. Coelho. CP1 Thermal Analysis.

[15] C. Day. CP2 Design Final Schematics, Rev. 0. Available at http://polysat.calpoly.edu/documents_cp2/systems/CP2_Schematic.zip.

[16] C. Day. CP2 Preliminary Communications Power Budget Estimation.

[17] W. A. Beech, D. E. Nielsen, and J. Taylor. AX.25 Link Access Protocol for Amateur Packet Radio. November 1997. Available at http://www.tapr.org/tapr/pdf/AX25.2.2.pdf.

[18] Yaesu Amateur Radio. The VX-1R has been replaced by the VX-2R. Webpage available at: http://www.yaesu.com.

[19] J. Schaffner. CP1 Communications Protocol. Available at http://polysat.calpoly.edu/documents_cp1/comm/cp1_com_a.pdf.

[20] The University of Hawaii CubeSat: An Undergraduate Student Satellite Project. Available at http://www-ee.eng.hawaii.edu/~cubesat/docs/UHCubeSatforUSSS.pdf.

[21] G. Hunyadi, D. M. Klumpar, S. Jepsen, B. Larsen, and M. Obland. A Commercial Off the Shelf (COTS) Packet Communications Subsystem for the Montana EaRth-Orbiting Pico-Explorer (MEROPE) Cubesat. *Proceedings of the 2002 IEEE Aerospace Conference*, 1:473–478, March 2002. Available at http://www.ssel.montana.edu/merope/abstracts/F203_1.pdf.

[22] PIC18FXX20 Data Sheet. Available at http://www.microchip.com.

[23] S. Hellan. AN018: CC1000 Debugging Hints and Troubleshooting. *Chipcon CC1000 Application Notes*. Available at http://www.chipcon.com/files/AN_018_CC1000_Debugging_Hints_1_1.pdf.

[24] Philips Semiconductor I^2C bus homepage. Available at http://www.semiconductors.philips.com/buses/i2c/.

[25] P. M. Evjan, S. Hellan, and S. Olsen. AN016: CC1000/CC1050 used with on-off keying. *Chipcon CC1000 Application Notes*. Available at http://www.chipcon.com/files/AN_016_CC1000_CC1050_On_Off_Keying_1_0.pdf.

[26] Quakesat Website. Available at http://ssdl.stanford.edu/LM-CubeSat/Team4/.

[27] 9600 Baud Packet Radio Modem Design. Available at http://www.amsat.org/amsat/articles/g3ruh/109.txt.

[28] International Amateur Radio Union, Amateur Satellite Frequency Coordination. Available at http://www.iaru.org/satellite/.