CPX: Design of a Standard Cubesat Software Bus

Jacob Farkas
California Polytechnic State University, San Luis Obispo

June 15, 2005

Abstract

The Cubesat standard brings the possibility of building a satellite within reach of undergraduate students. Cubesats can be built with cheap, commercial off the shelf products, and allow for a rapid development time. With the P-Pod launcher and the services of the Cubesat group at Cal Poly, Cubesat launches are within the budget of most University projects. A standard communication and command and data handling system allows for faster development of more complicated satellites, as all of the effort can be focused on creating complex payloads. Rather than focusing their energy on ensuring that the basic satellite functionality is operational, developers can build payloads to test and demonstrate experiments that could not otherwise be performed on Earth. This paper describes a standard software system for the CPX bus system that provides communication with an Earth station, sensor data collection, command and control of the satellite, and an adaptable payload interface.

Contents

1 Introduction **3**
- 1.1 Project History . . . 3
  - 1.1.1 Cubesat . . . 3
  - 1.1.2 CP1 . . . 4
  - 1.1.3 CP2 . . . 5

2 The CP2 Bus **6**
- 2.1 Specifications . . . 6
- 2.2 CP2 Features . . . 7
  - 2.2.1 I^2C . . . 7
  - 2.2.2 Attitude Determination and Control . . . 7

3 Design **9**
- 3.0.3 Module Upgrades . . . 9
- 3.0.4 I^2C Communication Interface . . . 9
- 3.0.5 Documented interprocessor communication and command interface . 9

4 Implementation **10**
- 4.1 Real Time Clock . . . 10
- 4.2 Battery Monitor . . . 11
  - 4.2.1 I^2C Library . . . 14
  - 4.2.2 How to add a command . . . 16

5 Future Work **19**
- 5.0.3 Peak Power Tracking . . . 19
- 5.0.4 Software Checksum . . . 20
- 5.0.5 Bootloader . . . 20
- 5.0.6 Contingency Mode . . . 20

6 Acknowledgments **21**

7 Source Code **22**
7.1 Battery Monitor 22
7.2 Battery Monitor 26
7.3 Real Time Clock 28
7.4 Real Time Clock 43
7.5 Real Time Clock Structure 45
7.6 I^2C Library 48
7.7 I^2C Library 52
7.8 I^2C Slave Code 65

Chapter 1

Introduction

A requirement for practically any satellite is the ability to collect basic operational data and communicate that data with an earth based communication station. A standard Cubesat bus will collect system sensor data, provide command and communication uplink and downlink to the earth station, and provide an interface to one or more payloads. The bus software must be easily adaptable to meet the requirements of different payloads. The CPX bus will speed the development of student-built Cubesats by providing a base system upon which to build more complex satellites.

1.1 Project History

1.1.1 Cubesat

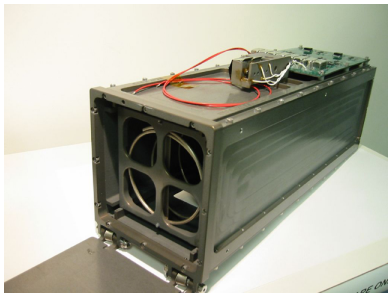


Figure 1.1: P-POD Mk. I

In 1998, Dr. Jordi Puig-Suari started a satellite design class at Cal Poly. His plan was to create a student-built small satellite. Stanford already had a small satellite group in existence at their Space Systems Development Laboratory working on a picosatellite

named OPAL. Dr. Bob Twiggs of Stanford decided to develop a picosatellite standard based on the size of a beanie baby box. Cal Poly worked with Stanford to develop the P-Pod deployer to carry multiple cubesats into space.

In 2001 the cubesat standard was released[1] and Cal Poly built the first revision of the P-Pod. The Cal Poly Cubesat team decided to build a cubesat of their own to learn about the issues other satellite developers were encountering in building to the cubesat standard.

In 2003, the first test of the P-Pod and the cubesat standard came when 3 cubesats were launched in June 2003 by Eurockot Launch Services. 14 cubesats in 5 P-Pods are scheduled to launch in late 2005[5].

1.1.2 CP1

Cal Poly's first satellite, CP1, was built so that the Cal Poly cubesat group could understand the issues that cubesat developers were facing in building cubesats for the P-Pod. The CP1 core team was comprised of seven Cal Poly students from aerospace and electrical engineering. The software was written without the input of any experienced programmers. CP1 is scheduled to launch on the 2005 DNEPR cubesat launch.

Figure 1.2: CP1: Cal Poly's First Satellite

Specifications

CP1's main bus used a single Netmedia BasicX-24 module for control[2]. The processor runs at 8mhz and contains 400 bytes of RAM and 32kB of EEPROM storage. An Alinco DJ-C5T radio was disassembled and used as the communication board for the satellite. CP1 contains two payloads: an Optical Electronics Technology sun sensor, and a mangetorquer built by Cal Poly students.

1.1.3 CP2

A new team was formed at Cal Poly dedicated solely to building Cubesats. The Polysat project was composed of many of the members of the original CP1 team. CP2 is Polysat's effort to create a satellite with a standardized bus that can be used on future cubesats. The power board and communication systems were designed from scratch by Cal Poly students, which required considerable effort. The structure was also designed and manufactured entirely by Cal Poly students. CP2 also marks the first Cal Poly satellite with a dedicated software team, and the software is written in embedded C rather than BASIC. The payload on CP2 was purchased by an external customer. Payload specifications were provided to the Polysat CP2 team and the payload was built by Cal Poly students to meet the customer's specifications. CP2 is scheduled to launch the 2005 DNEPR cubesat launch along with CP1.

Figure 1.3: CP2: Cal Poly's Second Satellite

The major difference between the design and construction of CP1 and CP2 is the multidisciplinary approach of CP2. Students from a variety of engineering majors worked together to produce parts of the satellite. CP1 was built almost entirely by aerospace engineering majors.

Chapter 2

The CP2 Bus

The majority of the CP2 bus code was written by Chris Noe. Preliminary code was also written by Matt Kaiser, but most of it was not used in the flight satellite. Chris deserves a great deal of credit for the work he put into the CP2 C&DH and Comm system. For a very in-depth analysis of the CP2 comm system, see Chris' senior project[3]

The CP2 bus will serve as the model for the CPX bus. A major benefit of the CP2 bus is its onboard communications components, designed from scratch and laid out on board rather than using a gutted ham radio. The communications subsystem software developed by Chris Noe[3] is a valuable asset to the CP2 bus. The power board, developed by Chris Day[4] provides power to the CP2 bus, side panels, and payload. The power board couples with the command and data handling board and provides connections between all of the satellite's boards.

Some problems were encountered in the design and development of the CP2 bus that must be resolved before the bus can become a standard base system for future Cubesats. The largest problems are electrical layout mistakes, which can be expected in any complex board design, and will be fixed in the next revision of the boards. Some software features have been left out due to pressing deadlines, and the interprocessor communication interface is specialized for CP2.

2.1 Specifications

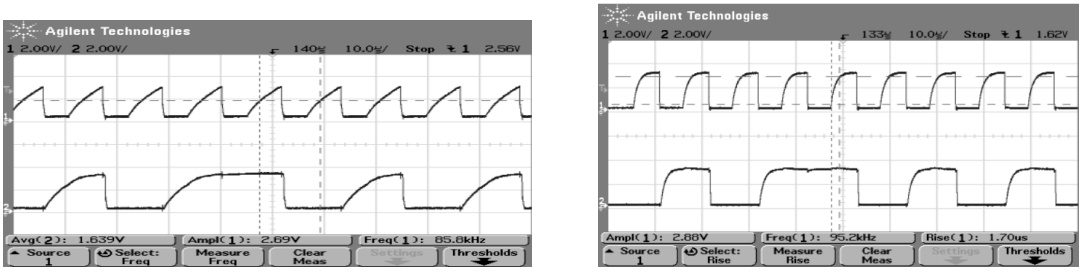
The CP2 bus was designed with three PIC18F6720 microprocessors; one command and data handling (C&DH) processor, and a redundant pair of communication processors. Communication to the payload, side panels, and between the C&DH and comm processors is done through the two wire I^2C bus. The communication system was developed using Chipcon CC1000 transceivers[3]. CP2 can communicate to an earth station with a 1200bps downlink and a 600bps uplink speed. For attitude determination and conrol, magnetorquers and HMC1052 2-axis magnetic sensors are built into the side panels of CP2.

The power board has two 4.2V lithium-ion batteries that provide a regulated 3V supply to the CP2 bus and side panels and an unregulated supply to the payload. The power board charges from solar panels on the side boards and has circuitry in place for peak power tracking.

2.2 CP2 Features

2.2.1 I^2C

A board design issue was encountered with the I^2C bus. I^2C is a serial protocol developed by Phillips that is used extensively on CP2 for communication between components. When the custom built boards came back from the manufacturer, the I^2C lines had a large amount of capacitance on them, resulting in sawtooth signals rather than square waves. This was unexpected, as we saw perfect square waves when we connected all of the parts on proto boards. Adding a Phillips P82B715 I^2C bus extender improved the signal a bit, but the signal was still far from a square wave. The signal was good enough that all of the I^2C devices could send and receive on the bus.



(a) (b)

Figure 2.1: (a) I^2C bus capacitance, (b) improved I^2C bus signal

Members of the EE team discovered that MOSFETs put in place to prevent a device from locking up and pulling the I^2C line low were causing the capacitance on the line. These MOSFETs were unnecessary because there is already enough protection built into the bus to avoid such a failure. After removing the unnecessary MOSFETs the I^2C bus signal became much cleaner.

2.2.2 Attitude Determination and Control

Aerospace majors involved in the Polysat project have worked at characterizing the magnetometers on the CP2 side panels in order to create an attitude determination and control (ADC) algorithm to detumble the satellite. The algorithm they chose to implement is the

B-Dot algorithm. The goal of the algorithm is to pulse the magnetorquers at the appropriate times to interact with the Earth's magnetic field and stabilize the satellite. Precise

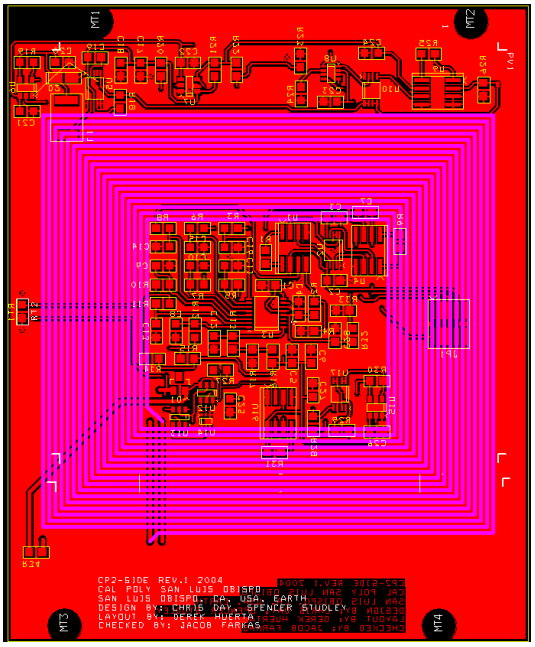


Figure 2.2: Magnetorquer traces in the CP2 side panels

positioning is not available with this algorithm, but ADC is a difficult problem to solve and not much work is done with ADC at the undergraduate level. The magnetorquers are an experimental idea for CP2; the torquers are constructed from coils of traces laid out in the four inside layers of the six layer boards used to make the side panels. By powering these coils with a pulse width modulated signal, the torquers should react with the Earth's magnetic field and move the satellite.

B-Dot has been implemented on CP2 by Kyle Leveque. Once CP2 is in orbit, data will be collected on its effectiveness.

Chapter 3

Design

3.0.3 Module Upgrades

The real time clock changed from CP2 to CPX. The chips are both made by the same manufacturer and use the same core, so upgrading the module for the new chip should not require much work. The existing battery monitor code is poorly written and is not adaptable. Many times during the CP2 project we wanted to improve the code but a rewrite would take too long and the existing code worked. Most of the old battery monitor code was written in inline assembly and is nonintuitive. The battery monitor code needs to be improved to be adaptable to pin changes, as the pins have changed between the CP2 and CPX bus and it is possible they will change in the future. A rewrite using C code or at least a cleaner interface to inline assembly is desired.

3.0.4 I^2C Communication Interface

The PIC18F6720 chips used on the satellite contain a hardware I^2C implementation. The Microchip C18 compiler libraries contain functions to interface with the hardware I^2C, but the library software is not suitable for mission critical flight software. The C18 I^2C libraries contain the potential for infinite loops and do not consistently report error messages. A rewrite of the libraries should take into account that the I^2C could be stuck low and allow for the satellite to continue functioning without I^2C. Error handling should be built into the new functions so that errors can be logged and reported to earth.

3.0.5 Documented interprocessor communication and command interface

Chris Noe developed a robust interprocessor communication and command system for CP2, but there is not much documentation provided aside from some comments in the code and the examples given by existing code. In this paper I will document the IPC and command interface and detail how to add a new command to the satellite.

Chapter 4

Implementation

4.1 Real Time Clock

A temperature compensated real time clock was added to the revision 3 board design of the CPX bus. The DS3231 chip provides a temperature compensated crystal, battery, and real time clock in one package. The chip interfaces with the C&DH processor through I^2C and provides timing accurate to approximately ± 3.5 parts per million. Our previous

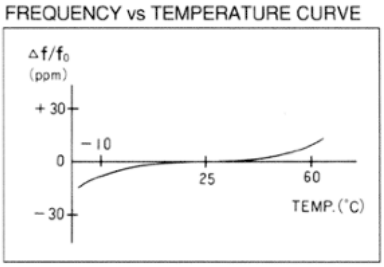


Figure 4.1: Frequency drift of CP2's RTC

real time clock, a DS1339, was located on the payload board and used an external crystal that provided a base of ± 20ppm accuracy that drifted with temperature. This real time clock can be expected to drift one second in half a day, and with extreme temperature variations in orbit it is likely that it drifts much more than that. A real time clock with a temperature compensated oscillator will drift one second in approximately five days. We need the temperature compensated oscillator to provide accurate timing for test runs or orbit determination. The DS3231 uses an internal temperature sensor and a capacitance array to compensate for the crystal's change in oscillation.

A concern in cubesat design is size, and although the DS3231 is a relatively small IC, it is very large in comparison with the rest of the chips on the CPX bus. When selecting

a new RTC for the satellite I did not take the size of the chip into consideration and was a bit surprised when I received samples of the chip. This is one of those experiences that is unique to a cubesat project compared to a full-sized satellite.

The DS3231 core is based on the DS1339, which made the transition very easy. I only needed to add a few extra registers to my code for the DS1339 and write a new function to access those registers. Since the DS3231 must use a temperature sensor internally to compensate for temperature effects on the crystal, it makes the temperature sensor available externally, so we get an extra temperature sensor on the satellite by upgrading the real time clock.

4.2 Battery Monitor

The CPX bus uses the Dallas DS2761 battery monitor chips to monitor and protect the onboard Li-Ion batteries. The battery monitor chips are commanded through the Dallas 1-Wire bus, which posed an interesting challenge. Communicating over a single wire means that the protocol is very time critical. The CPX bus uses PIC18 chips running at 4 MHz and the processor requires four cycles per instruction, so the satellite processors perform one million instructions per second, or one instruction every 1μs. The minimum timing on the one wire bus is 15μs, so the processor should be able to communicate on the one wire bus.

The CP2 code contained an existing implementation of a DS2761 battery monitor module, but the code is very disorganized and difficult to update if pins change. My goal was to create a layer of abstraction for the battery monitor chips so the functions would not have any code limited by the pins a battery monitor was on or the number of battery monitors that were connected. To communicate with the battery monitors I needed to change tristate pins to input or output and read and write to those pins. Reading a bit, for example, requires changing the pin to an input, reading the pin, and shifting it in to a variable. There is a window for each bit read or written that is at maximum 120μs. For special cases, such as reading a one, the maximum timing is 15μs. Abstracting the battery monitors from their actual pins involves extra processing overhead which takes us over or very close to the maximum timing requirements. Ultimately I had to make a compromise between abstraction and speed.

My first approach to rewriting the battery monitor code was to create a battery monitor handler struct which contained pointers to functions that would manipulate the pin for an individual battery monitor. This would encapsulate an external battery monitor as a struct that could be passed between functions to read and write from the battery monitor. I coded this solution and ran timing tests on it. I discovered that dereferencing the various pointers involved drove the time up to around 30μs for a single function call.

Listing 4.1: Function Call Design

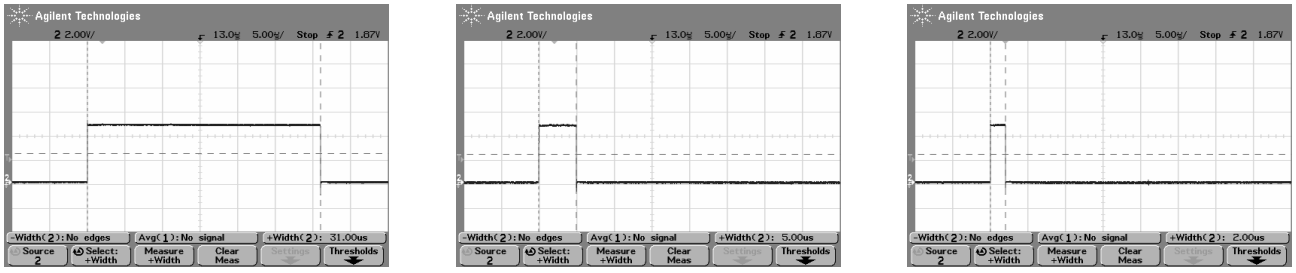
```
struct batteryHandler {
    void  (*io)(uchar);
    uchar (*in)(void);
    void  (*out)(uchar);
};
typedef struct batteryHandler batHandler;

// pin access functions, pointed to by function pointers in batHandler struct
void    batAIO(uchar io) { TRISBbits.TRISB0=io; }
uchar   batAin(void) { return PORTBbits.RB0; }
void    batAout(uchar out) { LATBbits.LATB0=out; }

void    batBIO(uchar io) { TRISBbits.TRISB1=io; }
uchar   batBin(void) { return PORTBbits.RB1; }
void    batBout(uchar out) { LATBbits.LATB1=out; }

// battery handler structs. reference the access functions above
batHandler batA = {batAIO, batAin, batAout},
           batB = {batBIO, batBin, batBout};
```

My next approach was to use macros that manipulated the pins for a battery monitor depending on a parameter that is passed in. The macros were a bit more efficient since they didn't require dereferencing pointers, but the way in which they compiled down to assembly instructions was indeterminate and there was the potential for timing issues to arise. The macro design had an average timing of 4μs for each call (switching the state of a tristate port, writing to a port, or reading from a port), meaning that the code would just barely meet the 15μs timing requirement for reading on the one wire bus. This was not good enough though because it was not a guaranteed time, only an approximate given testing of the individual macros.



(a) (b) (c)

Figure 4.2: Battery monitor timings for (a) function call, (b) macro, (c) inline assembly

Listing 4.2: Macro Design

```
#define BAT_A                       0x01
#define BAT_B                       0x02

#define BAT_TRIS                    TRISB
#define BAT_LAT                     LATB
#define BAT_PORT                    PORTB

#define BAT_SET_OUTPUT(bat)         (BAT_TRIS = BAT_TRIS & (~bat))
#define BAT_SET_INPUT(bat)          (BAT_TRIS = BAT_TRIS | bat)
#define BAT_PIN_READ(bat)           (BAT_PORT & bat)
#define BAT_PIN_HIGH(bat)           (BAT_LAT = BAT_LAT | bat)
#define BAT_PIN_LOW(bat)            (BAT_LAT = BAT_LAT & (~bat))
```

The macros were changed to use inline assembly so that the timing of each line in the timing critical sections could be determined exactly. This approach to writing the battery monitor code resolved the timing issues, but resulted in code that is a bit less clean that desired. The benefit is that an instruction for setting a port as an output now takes only 1μs instead of 30μs. The drawback is that the code assumes there will be only two battery monitors and there are different inline assembly macros for each battery monitor. This assumption seemed to be safe to make because the power board is only designed to handle two batteries. Adding more batteries would require a considerable redesign of the bus system.

Listing 4.3: Inline Assembly Macro Design

```
#define BAT_A                       0x01
#define BAT_B                       0x02

#define BAT_TRIS                    TRISB
#define BAT_LAT                     LATB
#define BAT_PORT                    PORTB

#define BAT_A_SET_OUTPUT            _asm BCF BAT_TRIS, 0, 0  _endasm
#define BAT_A_SET_INPUT             _asm BSF BAT_TRIS, 0, 0  _endasm
#define BAT_A_OUTPUT_HIGH           _asm BSF BAT_LAT,  0, 0  _endasm
#define BAT_A_OUTPUT_LOW            _asm BCF BAT_LAT,  0, 0  _endasm

#define BAT_B_SET_OUTPUT            _asm BCF BAT_TRIS, 1, 0  _endasm
#define BAT_B_SET_INPUT             _asm BSF BAT_TRIS, 1, 0  _endasm
#define BAT_B_OUTPUT_HIGH           _asm BSF BAT_LAT,  1, 0  _endasm
#define BAT_B_OUTPUT_LOW            _asm BCF BAT_LAT,  1, 0  _endasm

#define BAT_PIN_READ(bat)           (BAT_PORT & bat)  // ~4us
```

Although the rewrite turned out not to be possible in pure C, the code has been cleaned up a great deal. The macros around inline assembly code makes the code easier to read and understand. The nature of the assembly code was also changed a bit. Using different assembly commands I was able to perform the necessary pin manipulation functions in half the time of the old code.

4.2.1 I^{2}C Library

The I^{2}C library provided with the MC18 compiler had a flaw that could cause a potentially serious problem on a satellite. The I^{2}C libraries use busy waiting on the I^{2}C register to avoid bus collisions. The problem with this is that if one device locks up and holds the I^{2}C line low, the processors will end up stuck forever polling the I^{2}C line. Fortunately the watchdog timers will reset the processors after 2 minutes, but the processors will lock up not long after they reset when they try and access the I^{2}C line again. If the processors can not communicate on the I^{2}C bus the system is not of much use, since data can not flow from the C&DH processor to the comm processor and out to the earth station. It is still critical for the processor to remain running in the event of an I^{2}C failure. Many of the devices do not rely on I^{2}C and a good amount of data can still be collected and logged. In future satellites it may be necessary to have attitude control and determination always running, and if the processor was hung up on an I^{2}C failure it would not be able to achieve this.

I rewrote the MC18 I^{2}C libraries to be more robust. The functions use a for loop instead of a while loop and time out after a specified number of retries. I also added a small delay between retries so the processor did not retry too many times within a short period of time and then give up. The MC18 library required a call to `IdleI2C()` before almost every other I^{2}C library call to ensure that the bus was free before attempting to use it. This was an inconvenience to the programmer and also created repetitive code, so I rolled the `IdleI2C()` function into my version of the libraries where needed.

Listing 4.4: Example of writing bytes to a device using MC18 libraries.

```
OpenI2C(MASTER, SLEW_ON);
IdleI2C();

StartI2C();
IdleI2C();
if(PIR2bits.BCLIF)
    return(ERR_BUSCOL);

/* send the device address */
if(WriteI2C(address))
    return(ERR_WRITE);
IdleI2C();
if(SSPCON2bits.ACKSTAT)
    return(ERR_NOACK);

/* send the data memory address to the device */
if(WriteI2C(msb_addr))
    return(ERR_WRITE);
IdleI2C();
if(SSPCON2bits.ACKSTAT)
    return(ERR_NOACK);

if(WriteI2C(lsb_addr))
    return(ERR_WRITE);
IdleI2C();
```

```
if(SSPCON2bits.ACKSTAT)
    return(ERR_NOACK);

/* write the bytes in the array to the device */
for(i=0;i<len && i<MAX_WRITE;i++) {
    if(WriteI2C(*(data+i)))
        return(ERR_WRITE);
    IdleI2C();
    if(SSPCON2bits.ACKSTAT)
        return(ERR_NOACK);
}

StopI2C();
```

There were common sequences in the I^2C communication with slave devices. Communicating to a device using the MC18 libraries required the programmer to remember a sequence of library calls to set up the connection, talk to the device, and close the connection. These steps were the same no matter which chip you were talking to, again creating repetitive code. An example of this code can be seen in listing 4.4. I combined the reading and writing sequences into single read and write functions, as shown in figure 4.5, greatly simplifying the process of reading and writing to slave devices over I^2C.

Listing 4.5: New library functions.

```
char writeToSlave(unsigned char address, unsigned char data[], int dataLength)
{
    int index;

    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }
    if ((err = CP2_WriteI2C(address | I2C_WRITE)) < 0) {
        CP2_StopI2C();
        return err;
    }
    if ((err = CP2_CheckAckI2C()) < 0) {
        CP2_StopI2C();
        return err;
    }

    for (index = 0; index < dataLength; index++) {
        if ((err = CP2_WriteI2C(data[index])) < 0) {
            CP2_StopI2C();
            return err;
        }
    }

    CP2_StopI2C();

    return err;
}
```

4.2.2 How to add a command

One of the most common modifications to the CPX bus code will be the addition of new commands, both from the earth station and between processors. Since the command must flow from the comm processor to the C&DH processor and possibly to the payload processor, multiple files must be edited to handle a new command.

In this example I will outline the changes necessary to add a command for rotating the satellite around a given axis. The first step we need to take is to define the amount of data we expect to transmit to the satellite and between the processors. Currently the largest command allowed is 30 bytes. This limit was added in to conserve memory and to provide a simple sanity check on incoming commands. If you wish to have a longer command you must change the macro `COMMAND_BUFFER_SIZE` in `shared/cp2-commands.h`. In our case we will use a 16 bit number for the angle to rotate to and a 2 bit variable for the axis of rotation. Padding the bits out we end up with a 3 byte command.

We now need to add the command to the commands file so that it is recognized as a valid command. Edit `shared/cp2-commands.h` and add the command under the appropriate section. This command will run on the payload processor, so we add it under the last payload command listed in the file. Add the line at the bottom of listing 4.6.

Listing 4.6: Adding the new command to `cp2-commands.h`.

```
52 #define CMD_PAYLOAD_ADCS_SNAP       0x85
53 #define CMD_PAYLOAD_ROTATE          0x86
```

Now the comm code must be changed to recognize the new command as a valid command. The validation is done in a switch statement in `comm/comm-main.c`. Add line 405 as shown in listing 4.7.

Listing 4.7: Adding the new command to `comm-main.c`.

```
403 case CMD_PAYLOAD_GET_ADCS_SNAP:
404 case CMD_PAYLOAD_DUMP_ADCS_DATA:
405 case CMD_PAYLOAD_ROTATE:
406     commStatus |= COMM_STAT_CMD_RECVD;
407     COMM_INT = 1;       // let CDH know we have a command
408     break;
```

Upon receiving command `0x8A`, the comm processor will recognize it as a valid command and store it until C&DH polls it. When C&DH polls comm it will get the command and store the command data in the `commandData` buffer. The C&DH processor must then pass the data on to the payload processor. The interprocessor communication for the rotate command must be set up. First, the length of the data to be transferred to and from the payload processor must be defined in `cp2-i2c.h`. The TX size is the amount of data to be transferred from the C&DH processor to the slave processor. The RX length is the amount of data to be subsequently read from the slave processor.

Listing 4.8: Adding the new command to `cp2-i2c.h`.

```
403 #define IPC_PAYLOAD_ADCS_SNAP           0x85
404 #define IPC_PAYLOAD_ADCS_SNAP_TX        0
405 #define IPC_PAYLOAD_ADCS_SNAP_RX        5
406
407 #define IPC_PAYLOAD_ROTATE              0x86
408 #define IPC_PAYLOAD_ROTATE_TX           3
409 #define IPC_PAYLOAD_ROTATE_RX           0
```

If there is data transmitted in either the transmit or receive of the interprocessor communication, the TX and RX lengths defined above must be entered in the command table. The entries must be made in the position in the command table corresponding to their command number relative to the first command for that processor. For this example the first command for the payload processor is 0x80 and this command is 0x86, so it should occupy the 6th entry in the command table. Be sure to increment the command count for the command table you are adding the command to.

Listing 4.9: Adding the new command to `cp2-i2c.c`.

```
25 // this table maps commands to their expected TX/RX lengths
26 #define IPC_COMM_COMMAND_COUNT      13
27 #define IPC_PAYLOAD_COMMAND_COUNT   7
28 #define IPC_COMMAND_TABLE_WIDTH     2
29
30 unsigned char
31   payloadCommandTable[IPC_PAYLOAD_COMMAND_COUNT][IPC_COMMAND_TABLE_WIDTH] = {
32     {IPC_PAYLOAD_STATUS_SIZE_TX,         IPC_PAYLOAD_STATUS_SIZE_RX},
33     {IPC_PAYLOAD_SENSOR_SNAP_SIZE_TX,    IPC_PAYLOAD_SENSOR_SNAP_SIZE_RX},
34     {IPC_PAYLOAD_TEST_SNAP_SIZE_TX,      IPC_PAYLOAD_TEST_SNAP_SIZE_RX},
35     {IPC_PAYLOAD_NEW_TEST_SIZE_TX,       IPC_PAYLOAD_NEW_TEST_SIZE_RX},
36     {IPC_PAYLOAD_GET_TEST_SIZE_TX,       IPC_PAYLOAD_GET_TEST_SIZE_RX},
37     {IPC_PAYLOAD_ADCS_SNAP_TX,           IPC_PAYLOAD_ADCS_SNAP_RX}
38     {IPC_PAYLOAD_ROTATE_TX,              IPC_PAYLOAD_ROTATE_RX}
39 };
```

A new state must be added to the C&DH code in order for C&DH to process the command. This state will handle passing the command data to the payload controller and doing any necessary work on C&DH's side. You must add a new case for the `CMD_PAYLOAD_ROTATE` command in the switch statement in `executeCommand()`. In listing 4.10, the case has been added at the end of the switch statement, right before the default case.

Listing 4.10: Adding the new command to `cdh-main.c`.

```
403     case CMD_PAYLOAD_ROTATE:
404         if(cdhCurrentState == NORMAL_OPS) {
405             commandAck();
406
407             //put the command data into the i2c transmit buffer
408             memcpy((void*)i2cTxBuffer, (void*)&commandData, CMD_DATA_LENGTH);
409             if((err = transferI2C(PIC_PAYLOAD, IPC_PAYLOAD_ROTATE)) < 0) {
410                 logError(ERR_I2C_PAYLOAD);
411             }
412         } else {
```

```
413             commandNack();
414         }
415         break;
416
417     /* bad command */
418     default:
419         commandNack();
420         break;
421 }
```

To transmit data over the IPC interface, fill the `i2cTxBuffer` array with the data you wish to send and call `transferI2C` to perform the transfer. `transferI2C` uses the command table to perform the sending and receiving of data between the processors, sending data of length `IPC_PAYLOAD_ROTATE_TX` from `i2cTxBuffer` and reading `IPC_PAYLOAD_ROTATE_RX` bytes of data into the `i2cRxBuffer` array.

The payload code uses a state machine as the interrupt handler for I^2C. A sample of the code can be found in the appendix, listing 7.8.

Chapter 5

Future Work

5.0.3 Peak Power Tracking

Solar panel voltage and current levels vary as their temperature changes. In orbit the temperatures of the side panels will fluctuate as the sides face the sun or are hidden from the sun in the Earth's eclipse. In our design, a digital potentiometer is connected across each solar panel, and by varying the resistance of the digipot the voltage and current levels of the side panel can be altered. In theory, the power output of a solar panel should follow a parabolic curve for different digipot levels. By sweeping the digipot through its settings, a peak can be found, and by varying the digipot slightly every update and recording the power output it should be possible to find and track the peak power.

In practice it is not so simple. A peak power tracking scenario was never satisfactorily set up by the electrical engineering student that designed the peak power tracking circuitry, and testing peak power tracking software was difficult to test. Varying the digipot levels often seemed to have little to no effect on either the voltage or the current, and attempts by the software team to connect physical measuring devices resulted in fried components. The solar panel voltage and current readings were far too noisy to read any useful data from, and when they were filtered out their latency was too great for the software to appropriately respond to. We could not get the electrical engineering student to help us test the circuitry, and due to a proper knowledge of the system, peak power tracking was shelved for a future satellite.

Peak power tracking is difficult to characterize. The angle of incidence of sunlight, the intensity of the sunlight, the temperature of the solar panel, the battery voltage, and the digipot setting all affect the solar panel voltage and current. It is difficult to narrow the variables to only one or two of these factors across multiple tests to verify that peak power tracking is performing better than a static solar panel. There is also the problem of the digipot getting stuck on a high or low level while in eclipse to the point where sunlight on the panels is not sensed.

Peak power tracking would be a nice addition to the CPX bus, but there are doubts

as to the amount of benefit it would provide compared to the amount of work required to implement.

5.0.4 Software Checksum

When performing the final programmings of the satellite it was hard to keep track of what changes were made to the code and what satellite had which version of the code programmed on it. A checksum command would help determine what version of code was programmed on a satellite without having to connect the satellite to a debugger.

5.0.5 Bootloader

A bootloader will allow reprogramming of the processors through a serial interface using a small program on the PC, rather than through the programming lines and using the Microchip MPLAB IDE. This could allow for more processors to be accessed through the CP2 bus, since only two lines would be needed for each processor.

The bootloader could also use the checksum routine during processor startup. If the code in memory does not match the checksum, a copy of the code could be loaded from the onboard flash EEPROM memory. This would fix any errors possibly introduced in the code by radiation in space.

An implementation of a serial bootloader is not currently possible, as the boards for CP2 do not have serial lines exposed to the umbilical box. Revision 3 of the C&DH board has RS232 lines on the umbilical. As of the time of writing, revision 3 boards are still unavailable.

5.0.6 Contingency Mode

The electrical design for CP2 included a contingency mode, in which one of the comm processors would take over for the C&DH processor and communicate directly with the payload processor. Contingency mode was not implemented in the CPX software, although the hardware is in place for allow it. Dealing with multiple masters on the I^2C bus creates many concurrency problems, and there is no easy way to resolve bus contention with the PIC I^2C libraries we are using.

Chapter 6

Acknowledgments

- Chris Noe for his work on developing the original CP2 bus system
- Kyle Leveque for the long nights of coding and arguing about do/while loops
- All of the Polysat and Cubesat team members for making the lab a great place to work in
- Dr. Jordi Puig-Suari for letting us hang out on his roof
- Dr. Clark Savage Turner for his assistance and advice

Chapter 7

Source Code

7.1 Battery Monitor

```
#include "cdh-batmon.h"

#undef BATMON_UNITTEST

#ifdef BATMON_UNITTEST

void main(void) {
    struct BatteryMonitorData data;
    char status;

    ResetBatteryMonitor(BAT_A);
    ResetBatteryMonitor(BAT_B);

    while(1) {
        status=BatteryMonitorRead(&data, BAT_A);
        status=BatteryMonitorRead(&data, BAT_B);
    }
}
#endif

// resets the protection register flags and clears the current accumulation register
// returns:
//  -1 if the batmon is not responding
//  0 otherwise
char ResetBatteryMonitor(char bat) {
    /* write PROT reg */
    if(BatmonWriteReg(REG_PROT, bat)) {
        return -1;
    }
    // clear all protection register flags except charge and discharge enable
    WriteByte1Wire(0x03, bat);

    /* zero out accumulated current reg */
    if(BatmonWriteReg(REG_ACC_CUR_MSB, bat)) {
        return -1;
    }
```

```
    WriteByte1Wire(0, bat);
    WriteByte1Wire(0, bat);

    return 0;
}

// fills the BatteryMonitorData struct with battery monitor data
// returns
//  -1 if the batmon is not responding
//  0 otherwise
char BatteryMonitorRead(struct BatteryMonitorData *batmon, uchar bat) {
    char curMSB=0, curLSB=0;
    memset(batmon, 0x00, sizeof(struct BatteryMonitorData));

    if(BatmonReadReg(REG_PROT, bat)) {
        return -1;
    }
    batmon->protection=ReadByte1Wire(bat);
    batmon->status=ReadByte1Wire(bat);

    if(BatmonReadReg(REG_VOLT_MSB, bat)) {
        return -1;
    }
    batmon->voltage=ReadByte1Wire(bat);
    // throw away LSB of voltage
    ReadByte1Wire(bat);

    curMSB = ReadByte1Wire(bat);
    curLSB = ReadByte1Wire(bat);
    batmon->current=curMSB;
    batmon->current <<= 8;
    batmon->current|=curLSB;
    batmon->current >>= 3;
    // sign extension doesn't work for that previous shift. wtf?
    if(curMSB & 0x80) {
        batmon->current |= 0xE000;
    }

    curMSB = ReadByte1Wire(bat);
    curLSB = ReadByte1Wire(bat);
    batmon->accumCurrent=((int)curMSB)<<8;
    batmon->accumCurrent|=curLSB;

    if(BatmonReadReg(REG_TEMP_MSB, bat)) {
        return -1;
    }
    batmon->temperature=ReadByte1Wire(bat);

    return 0;
}

// sets up bus for reading from a register. call this function then perform reads and needed.
// returns:
//  -1 if the batmon is not responding
//  0 otherwise
char BatmonReadReg(uchar reg, uchar bat) {
    if(Reset1WireBus(bat))
        return -1;
```

```
    WriteByte1Wire(SKIP_NET_ADD, bat);
    WriteByte1Wire(READ_FUNCTION, bat);
    WriteByte1Wire(reg, bat);
    return 0;
}

// sets up bus for writing to a register. call this function then perform writes as needed.
// returns:
//  -1 if the batmon is not responding
//  0 otherwise
char BatmonWriteReg(uchar reg, uchar bat) {
    if(Reset1WireBus(bat))
        return -1;
    WriteByte1Wire(SKIP_NET_ADD, bat);
    WriteByte1Wire(WRITE_FUNCTION, bat);
    WriteByte1Wire(reg, bat);
    return 0;
}

void ReadBytes1Wire(uchar *outbytes, uchar len, char bat) {
    uchar *data;

    for(data=outbytes; data<outbytes+len; data++) {
        *data=ReadByte1Wire(bat);
    }
}

void WriteBytes1Wire(const uchar *inbytes, uchar len, char bat) {
    const uchar *data;

    for(data=inbytes; data<inbytes+len; data++) {
        WriteByte1Wire(*data, bat);
    }
}

uchar ReadByte1Wire(char bat) {
    uchar mask, data=0;

    // reset both pin states
    BAT_A_OUTPUT_HIGH
    BAT_B_OUTPUT_HIGH
    BAT_A_SET_INPUT
    BAT_B_SET_INPUT

    for(mask=0x01; mask; mask<<=1) {
        // disable interrupts
        INTCONbits.GIE = 0;

        // drop line low to initiate the read
        if(bat == BAT_A) {
            BAT_A_OUTPUT_LOW
            BAT_A_SET_OUTPUT
            BAT_A_SET_INPUT
        } else {
            BAT_B_OUTPUT_LOW
            BAT_B_SET_OUTPUT
            BAT_B_SET_INPUT
        }
```

```
        // the read must happen within 15us of dropping the pin low.
        // setting the pin as an input takes 1us (2 more for branch)
        // and reading the pin takes 3us. timing should never be more than 6us.
        if(BAT_PIN_READ(bat)) {
            data |= mask;
        }

        DELAY_SLOT;

        // enable interrupts
        INTCONbits.GIE = 1;
    }

    BAT_A_SET_INPUT
    BAT_B_SET_INPUT

    return data;
}

void WriteByte1Wire(uchar data, char bat) {
    uchar mask;

    BAT_A_OUTPUT_HIGH
    BAT_B_OUTPUT_HIGH
    BAT_A_SET_INPUT
    BAT_B_SET_INPUT

    if(bat == BAT_A) {
        BAT_A_SET_OUTPUT
    } else {
        BAT_B_SET_OUTPUT
    }

    for(mask=0x01; mask; mask<<=1) {
        // disable interrupts
        INTCONbits.GIE = 0;

        if(data&mask) {
            BAT_A_OUTPUT_LOW
            NOP
            BAT_A_OUTPUT_HIGH

            BAT_B_OUTPUT_LOW
            NOP
            BAT_B_OUTPUT_HIGH
        } else {
            BAT_A_OUTPUT_LOW
            BAT_B_OUTPUT_LOW
        }

        DELAY_SLOT;

        BAT_A_OUTPUT_HIGH
        BAT_B_OUTPUT_HIGH

        // enable interrupts between bits so we don't bog
        // the rest of the system down
```

```
        INTCONbits.GIE = 1;
    }

    BAT_A_SET_INPUT
    BAT_B_SET_INPUT

    return;
}

// Returns:
//  -1 if batmon does not respond
//   0 if everything is ok
char Reset1WireBus(char bat) {
    char retval = 0;

    // disable interrupts
    INTCONbits.GIE = 0;

    BAT_A_SET_INPUT
    BAT_B_SET_INPUT
    BAT_A_OUTPUT_HIGH
    BAT_B_OUTPUT_HIGH

    if(bat == BAT_A) {
        BAT_A_SET_OUTPUT
    } else {
        BAT_B_SET_OUTPUT
    }

    // send reset pulse
    BAT_A_OUTPUT_LOW
    BAT_B_OUTPUT_LOW

    // keep low for reset time
    DELAY_T_RSTL;

    // verify response
    BAT_A_SET_INPUT
    BAT_B_SET_INPUT
    DELAY_T_PDH;
    if(BAT_PIN_READ(bat)) {
        retval = -1;
    }

    // enable interrupts
    INTCONbits.GIE = 1;

    DELAY_T_RSTH;

    return retval;
}
```

7.2 Battery Monitor

```
#ifndef CDH_BATMON_H
```

```
#define CDH_BATMON_H

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

typedef unsigned char uchar;

struct BatteryMonitorData {
    uchar protection;
    uchar status;
    uchar voltage;
    uchar temperature;
    int current;
    int accumCurrent;
};

/* battery monitor functions */
#define SKIP_NET_ADD    0xCC
#define WRITE_FUNCTION  0x6C
#define READ_FUNCTION   0x69

/* bat mon registers */
#define REG_PROT        0x00
#define REG_STAT        0x01
#define REG_EEPROM      0x07
#define REG_SFR         0x08
#define REG_VOLT_MSB    0x0C
#define REG_VOLT_LSB    0x0D // only upper 3 bits matter
#define REG_CUR_MSB     0x0E
#define REG_CUR_LSB     0x0F
#define REG_ACC_CUR_MSB 0x10
#define REG_ACC_CUR_LSB 0x11 // only upper 5 bits matter
#define REG_TEMP_MSB    0x18
#define REG_TEMP_LSB    0x19 // only upper 3 bits matter

/* PROT bits */
#define PROT_DE  0x01    /* discharge enable */
#define PROT_CE  0x02    /* charge enable */
#define PROT_DC  0x04    /* /DC pin (readonly) */
#define PROT_CC  0x08    /* /CC pin (readonly) */
#define PROT_DOC 0x10    /* discharge overcurrent flag */
#define PROT_COC 0x20    /* charge overcurrent flag */
#define PROT_UV  0x40    /* undervoltage flag */
#define PROT_OV  0x80    /* overvoltage flag */

// abstracting away the battery monitors so they were not dependant on specific pins
// was not possible, since the code is very time critical.
#define BAT_A                       0x01
#define BAT_B                       0x02

#define BAT_A_SET_OUTPUT            _asm BCF TRISB, 0, 0  _endasm
#define BAT_A_SET_INPUT             _asm BSF TRISB, 0, 0  _endasm
#define BAT_A_OUTPUT_HIGH           _asm BSF LATB,  0, 0  _endasm
#define BAT_A_OUTPUT_LOW            _asm BCF LATB,  0, 0  _endasm

#define BAT_B_SET_OUTPUT            _asm BCF TRISB, 1, 0  _endasm
```

```
#define BAT_B_SET_INPUT             _asm BSF TRISB, 1, 0  _endasm
#define BAT_B_OUTPUT_HIGH           _asm BSF LATB,  1, 0  _endasm
#define BAT_B_OUTPUT_LOW            _asm BCF LATB,  1, 0  _endasm

#define NOP                         _asm NOP _endasm

#define BAT_PIN_READ(bat)           (PORTB & bat)  // ~4us

// DS2761 timings (DS2761 data sheet p24)
// timing values between the minimum and maximum timings have been chosen to allow some padding
//  - there is no way to specify a delay between 1TCY (a NOP) and 10TCYx (10 instructions).
//    hence the multiple Delay1TCY


#define DELAY_SLOT      (Delay10TCYx(6))

// reset timing
#define DELAY_T_RSTH            (Delay100TCYx(5))
#define DELAY_T_RSTL            (Delay100TCYx(5))
// presence detect
#define DELAY_T_PDH             (Delay10TCYx(7))

char ResetBatteryMonitor(char bat);

char BatteryMonitorRead(struct BatteryMonitorData *batmon, uchar battery);

char BatmonReadReg(uchar reg, uchar bat);
char BatmonWriteReg(uchar reg, uchar bat);

void ReadBytes1Wire(uchar *bytes, uchar len, char bat);
uchar ReadByte1Wire(char bat);
void WriteBytes1Wire(const uchar *bytes, uchar len, char bat);
void WriteByte1Wire(uchar data, char bat);

char Reset1WireBus(char bat);

#endif
```

7.3 Real Time Clock

```
/*
 * rtc.c
 *
 * DS3231 RTC driver
 *
 * Author: Jacob Farkas
 * $Id: rtc.c,v 1.1 2004/12/17 20:37:32 kleveque Exp $
 */
#include "rtc.h"

/* GLOBALS */
static unsigned char rtc_alm1_enable = 0, rtc_alm2_enable = 0;

/*
 * rtc_readRegs(union rtc_regs *rtc_read)
```

```
 *
 * PARAMETERS:
 *  *rtc_read - this union is filled with a copy of the RTC's registers
 *
 * DESCRIPTION:
 *  Reads in all of the registers from the RTC into a rtc_regs union. By calling
 *  this you can get the current time from the RTC. Run the rtc_regs union
 *  through rtc_regsToTime() to get a timestamp that's easier to work with.
 *
 * RETURNS:
 *  ERR_NONE on success
 *  I2C error code on I2C errors
 */
char rtc_readRegs(union rtc_regs *rtc_read)
{
    unsigned char startReg = REG_SECOND;

    memset(rtc_read, 0x00, sizeof(union rtc_regs));

    /* point us at the seconds register */
    if ((err = writeToSlave(RTC_DS3231, &startReg, 1)) < 0) {
        return err;
    }

    /* read from the registers */
    return readFromSlave(RTC_DS3231, sizeof(union rtc_regs), rtc_read->data);
}

/*
 * rtc_writeTime(union rtc_regs *rtc_write)
 *
 * PARAMETERS:
 * *rtc_write - the registers to write to the RTC.
 *
 * DESCRIPTION:
 *  Writes the time information in rtc_write to the RTC. Only second, minute,
 *  hour, day, date, month, and year are written- the alarms, control, and
 *  status registers are left untouched.
 *
 * RETURNS:
 *  ERR_NONE on success
 *  I2C error code on I2C errors
 */
char rtc_writeTime(struct rtc_time *rtctime)
{
    unsigned char rtc_buf[sizeof(union rtc_regs) + 1];
    static union rtc_regs rtc_write;

    rtc_timeToRegs(rtctime, &rtc_write);

    rtc_buf[0] = REG_SECOND;
    memcpy((void*)(rtc_buf+1), (void*)&rtc_write, sizeof(union rtc_regs));

    return writeToSlave(RTC_DS3231, rtc_buf, sizeof(union rtc_regs) + 1);
}

/*
 * rtc_readtime
```

```
 *
 * DESCRIPTION:
 *  reads the time from the RTC into an rtc_time struct
 *
 * PARAMETERS
 *  struct rtc_time *t
 *      pointer to the rtc_time struct to fill
 *
 * RETURNS:
 *  ERR_NONE on success
 *  I2C error code on failure
 */
char rtc_readTime(struct rtc_time *t)
{
    static union rtc_regs rregs;

    if ((err = rtc_readRegs(&rregs)) < 0) {
        return err;
    }

    return rtc_regsToTime(&rregs, t);
}

/*
 * rtc_timeToRegs(struct rtc_time *t, union rtc_regs *r)
 *
 * PARAMETERS:
 *  *t - the timestamp to place in the registers
 *  *r - pointer to the register union to fill
 *
 *  DESCRIPTION:
 *   Converts a given timestamp struct into registers for the RTC. These
 *   registers can then be transferred directly to the RTC to program a time
 *
 * RETURNS:
 *  ERR_NONE at all times
 */
char rtc_timeToRegs(struct rtc_time *t, union rtc_regs *r)
{
    unsigned char date, ten_date;
    memset(r, 0x00, sizeof(union rtc_regs));

    r->reg.second.b.second = t->second % 10;
    r->reg.second.b.ten_second = t->second / 10;

    r->reg.minute.b.minute = t->minute % 10;
    r->reg.minute.b.ten_minute = t->minute / 10;

    r->reg.hour.b.hour = t->hour % 10;
    if (t->hr_format == TWELVE_HOUR) {
        /* this is a hackish thing to do, but it avoids division and casting */
        r->reg.hour.b.ten_hour = (t->hour > 9);
        r->reg.hour.b.ampm = t->ampm;
    } else {
        if (t->hour >= 20) {
            r->reg.hour.b.ten_hour = 0;
            r->reg.hour.b.ampm = 1;
        } else if (t->hour >= 10) {
```

```
            r->reg.hour.b.ten_hour = 1;
            r->reg.hour.b.ampm = 0;
        } else {
            r->reg.hour.b.ten_hour = 0;
            r->reg.hour.b.ampm = 0;
        }
    }
    r->reg.hour.b.format = t->hr_format;

    r->reg.day.b.day = t->day;

    r->reg.date.b.date = t->date % 10;
    r->reg.date.b.ten_date = t->date / 10;

    date=t->date;
    date=t->date%10;
    ten_date=t->date/10;

    r->reg.month.b.month = t->month % 10;
    r->reg.month.b.ten_month = t->month / 10;
    r->reg.month.b.century = t->century;

    r->reg.year.b.year = t->year % 10;
    r->reg.year.b.ten_year = t->year / 10;

    return ERR_NONE;
}

/*
 * rtc_regsToTime(union rtc_regs *r, struct rtc_time *t
 *
 * PARAMETERS:
 *  *r - the registers to convert into a timestamp
 *  *t - the timestamp that is created
 *
 * DESCRIPTION:
 *  Converts an rtc_regs union into a timestamp for easier manipulation/reading
 *
 * RETURNS:
 *  ERR_NONE at all times
 */
char rtc_regsToTime(union rtc_regs *r, struct rtc_time *t)
{
    memset(t, 0x00, sizeof(struct rtc_time));

    t->second = r->reg.second.b.second + r->reg.second.b.ten_second * 10;
    t->minute = r->reg.minute.b.minute + r->reg.minute.b.ten_minute * 10;

    if (r->reg.hour.b.format) {
        /* 12 hour mode */
        t->hr_format = TWELVE_HOUR;
        t->hour = r->reg.hour.b.hour + r->reg.hour.b.ten_hour * 10;
        t->ampm = r->reg.hour.b.ampm;
    } else {
        /* 24 hour mode */
        t->hr_format = TWENTYFOUR_HOUR;
        t->hour =
            r->reg.hour.b.hour + r->reg.hour.b.ten_hour * 10 +
```

```
            r->reg.hour.b.ampm * 20;
        /* ampm is invalid in this case */
        t->ampm = TIME_MASK;
    }

    t->day = r->reg.day.b.day;
    t->date = r->reg.date.b.date + r->reg.date.b.ten_date * 10;
    t->month = r->reg.month.b.month + r->reg.month.b.ten_month * 10;
    t->century = r->reg.month.b.century;
    t->year = r->reg.year.b.year + r->reg.year.b.ten_year * 10;

    return ERR_NONE;
}

/*
 * rtc_setAlarm1(struct rtc_time *t)
 *
 * PARAMETERS:
 *  *t - the time to set an alarm for.
 *
 * DESCRIPTION:
 *  According to the data sheet, there are only certain combinations of masks
 *  that will work. if an invalid mask is encountered, ERR_RTC_INVALID_MASK is returned,
 *  otherwise 0 is returned.
 *  If a time is given, the alarm will go off when the current time matches that
 *  time. If the field is set to TIME_MASK, that field will always match.
 *
 *  RETURNS:
 *   ERR_NONE on success
 *   ERR_RTC_INVALID_MASK if the alarm given has an invalid mask
 *   I2C error code on I2C errors
 */
char rtc_setAlarm1(struct rtc_time *t)
{
    union rtc_regs r;
    unsigned char mask = 0, i;

    if (t->second != TIME_MASK) {
        r.reg.a1_second.b.second = t->second % 10;
        r.reg.a1_second.b.ten_second = t->second / 10;
        r.reg.a1_second.b.a1m1 = 0;
    } else {
        r.reg.a1_second.b.a1m1 = 1;
        mask = 1;
    }

    if (t->minute != TIME_MASK) {
        if (mask) {
            return ERR_RTC_INVALID_MASK;
        }

        r.reg.a1_minute.b.minute = t->minute % 10;
        r.reg.a1_minute.b.ten_minute = t->minute / 10;
        r.reg.a1_minute.b.a1m2 = 0;
    } else {
        r.reg.a1_minute.b.a1m2 = 1;
        mask = 1;
    }
```

```
    if (t->hour != TIME_MASK) {
        if (mask) {
            return ERR_RTC_INVALID_MASK;
        }

        r.reg.a1_hour.b.hour = t->hour % 10;
        if (t->hr_format == TWELVE_HOUR) {
            r.reg.a1_hour.b.ten_hour = (t->hour > 9);
            r.reg.a1_hour.b.ampm = t->ampm;
        } else {
            if (t->hour >= 20) {
                r.reg.a1_hour.b.ten_hour = 0;
                r.reg.a1_hour.b.ampm = 1;
            } else if (t->hour >= 10) {
                r.reg.a1_hour.b.ten_hour = 1;
                r.reg.a1_hour.b.ampm = 0;
            } else {
                r.reg.a1_hour.b.ten_hour = 0;
                r.reg.a1_hour.b.ampm = 0;
            }
        }
        r.reg.a1_hour.b.format = t->hr_format;
        r.reg.a1_hour.b.a1m3 = 0;
    } else {
        r.reg.a1_hour.b.a1m3 = 1;
        mask = 1;
    }

    if (t->date != TIME_MASK) {
        if (mask) {
            return ERR_RTC_INVALID_MASK;
        }
        r.reg.a1_daydate.b.daydate = t->date % 10;
        r.reg.a1_daydate.b.ten_date = t->date / 10;
        r.reg.a1_daydate.b.a1m4 = 0;
        r.reg.a1_daydate.b.dydt = 0;
//    } else if(t->day!=TIME_MASK) {
//        /* we can't do both a day and a date- they have to pick just one */
//        if(mask && t->date!=TIME_MASK)
//            return ERR_RTC;
//        r.reg.a1_daydate.b.daydate=t->day;
//        r.reg.a1_daydate.b.a1m4=0;
//        r.reg.a1_daydate.b.dydt=1;
    } else {
        r.reg.a1_daydate.b.a1m4 = 1;
    }

    /* disable + clear alarms before doing anything else */
    if ((err = rtc_disableAlarms(0, 0)) < 0) {
        return err;
    }

    if ((err = rtc_clearAlarms(0, 0)) < 0) {
        return err;
    }

    /* set the alarm time */
```

```
    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(RTC_DS3231 | I2C_WRITE)) < 0) {
        return err;
    }

    /* start in the alarm1.seconds register */
    if ((err = CP2_WriteI2C(REG_A1_SECOND)) < 0) {
        return err;
    }

    for (i = REG_A1_SECOND; i <= REG_A1_DAYDATE; i++) {
        if ((err = CP2_WriteI2C(r.data[i])) < 0) {
            return err;
        }
    }

    if ((err = CP2_StopI2C()) < 0) {
        return err;
    }

    /* set the alarm interrupt bit */
    rtc_alm1_enable = 1;
    r.reg.control.b._eosc = 0;
    r.reg.control.b.intcn = 1;
    r.reg.control.b.a1ie = 1;
    r.reg.control.b.a2ie = rtc_alm2_enable;

    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(RTC_DS3231 | I2C_WRITE)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(REG_CONTROL)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(r.data[REG_CONTROL])) < 0) {
        return err;
    }

    if ((err = CP2_StopI2C()) < 0) {
        return err;
    }

    return ERR_NONE;
}

/*
 * rtc_setAlarm2(struct rtc_time *t)
 *
 * PARAMETERS:
 *  *t - the time to set an alarm for
```

```
 *
 * DESCRIPTION:
 *  alarm2 only accepts minutes, hours, and day/date
 *  refer to the notes on rtc_setAlarm1()
 *
 *  RETURNS:
 *   ERR_NONE on success
 *   ERR_RTC if there's a mask error
 *   I2C error code on I2C errors
 */
char rtc_setAlarm2(struct rtc_time *t)
{
    union rtc_regs r;
    unsigned char mask = 0, i;

    if (t->minute != TIME_MASK) {
        if (mask) {
            return ERR_RTC_INVALID_MASK;
        }

        r.reg.a2_minute.b.minute = t->minute % 10;
        r.reg.a2_minute.b.ten_minute = t->minute / 10;
        r.reg.a2_minute.b.a2m2 = 0;
    } else {
        r.reg.a2_minute.b.a2m2 = 1;
        mask = 1;
    }

    if (t->hour != TIME_MASK) {
        if (mask) {
            return ERR_RTC_INVALID_MASK;
        }

        r.reg.a2_hour.b.hour = t->hour % 10;
        if (t->hr_format == TWELVE_HOUR) {
            /* this is a hackish thing to do, but it avoids division and casting */
            r.reg.a2_hour.b.ten_hour = (t->hour > 9);
            r.reg.a2_hour.b.ampm = t->ampm;
        } else {
            if (t->hour >= 20) {
                r.reg.a2_hour.b.ten_hour = 0;
                r.reg.a2_hour.b.ampm = 1;
            } else if (t->hour >= 10) {
                r.reg.a2_hour.b.ten_hour = 1;
                r.reg.a2_hour.b.ampm = 0;
            } else {
                r.reg.a2_hour.b.ten_hour = 0;
                r.reg.a2_hour.b.ampm = 0;
            }
        }
        r.reg.a2_hour.b.format = t->hr_format;
        r.reg.a2_hour.b.a2m3 = 0;
    } else {
        r.reg.a2_hour.b.a2m3 = 1;
        mask = 1;
    }

    if (t->day != TIME_MASK) {
```

```
        /* we can't do both a day and a date- they have to pick just one */
        if (mask || t->date != TIME_MASK) {
            return ERR_RTC_INVALID_MASK;
        }

        r.reg.a2_daydate.b.daydate = t->day;
        r.reg.a2_daydate.b.a2m4 = 0;
        r.reg.a2_daydate.b.dydt = 1;
    } else if (t->date != TIME_MASK) {
        if (mask) {
            return ERR_RTC_INVALID_MASK;
        }

        r.reg.a2_daydate.b.daydate = t->date % 10;
        r.reg.a2_daydate.b.ten_date = t->date / 10;
        r.reg.a2_daydate.b.a2m4 = 0;
        r.reg.a2_daydate.b.dydt = 0;
    } else {
        r.reg.a2_daydate.b.a2m4 = 1;
    }

    /* set the alarm time */
    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(RTC_DS3231 | I2C_WRITE)) < 0) {
        return err;
    }

    /* start in the alarm1.seconds register */
    if ((err = CP2_WriteI2C(REG_A2_MINUTE)) < 0) {
        return err;
    }


    for (i = REG_A2_MINUTE; i <= REG_A2_DAYDATE; i++) {
        if ((err = CP2_WriteI2C(r.data[i])) < 0) {
            return err;
        }
    }

    if ((err = CP2_StopI2C()) < 0) {
        return err;
    }

    /* set the alarm interrupt bit */
    rtc_alm2_enable = 1;
    r.reg.control.b._eosc = 0;
    r.reg.control.b.intcn = 1;
    r.reg.control.b.a1ie = rtc_alm1_enable;
    r.reg.control.b.a2ie = 1;

    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(RTC_DS3231 | I2C_WRITE)) < 0) {
```

```
        return err;
    }

    if ((err = CP2_WriteI2C(REG_CONTROL)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(r.data[REG_CONTROL])) < 0) {
        return err;
    }

    if ((err = CP2_StopI2C()) < 0) {
        return err;
    }

    return ERR_NONE;
}

char rtc_disableAlarms(char a1, char a2)
{
    /* write to the control register */
    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(RTC_DS3231 | I2C_WRITE)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(REG_CONTROL)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(0x04 | ((a2 & 0x01) << 1) | (a1 & 0x01))) < 0) {
        return err;
    }

    if ((err = CP2_StopI2C()) < 0) {
        return err;
    }

    return ERR_NONE;
}

/*
 * rtc_checkOSF
 *
 * DESCRIPTION:
 *  checks the RTC oscillator stop flag and clears it if needed.
 *
 * PARAMETERS:
 *  none
 *
 * RETURNS:
 *  1 if the flag was set and cleared
 *  0 if the flag was not set
 */
char rtc_checkOSF(void)
```

```
{
    unsigned char startReg[]={REG_STATUS, 0};
    unsigned char stat = 0;

    /* we only want to return a 1 or a 0 for the status of the OSF.
     * returning an error value from an I2C transaction would mess things up
     * so we're just going to ignore I2C errors.
     */
    /* point us at the status register */
    if ((err = writeToSlave(RTC_DS3231, startReg, 1)) < 0) {
        return err;
    }

    /* read from the status register */
    if ((err = readFromSlave(RTC_DS3231, 1, &stat)) < 0) {
        return err;
    }

    if (stat & 0x80) {
        /* clear the OSF */
        startReg[1]=stat & 0x7F;

        if ((err = writeToSlave(RTC_DS3231, startReg, 2)) < 0) {
            return err;
        }
        return 1;
    } else {
        return 0;
    }
}

/*
 * rtc_clearAlarms(char a1, char a2)
 *
 * PARAMETERS:
 *  a1 - alarm1 clear flag
 *  a2 - alarm2 clear flag
 *
 * DESCRIPTION:
 *  Clears the alarm flags for the given alarms. 1 leaves the flag as is, 0
 *  clears the flag.
 *  Alarm flags must be cleared for another interrupt to occur. It is not
 *  possible to set an alarm flag, only clear it.
 *  For example, to clear alarm 1 but not alarm 2, call rtc_clearAlarms(0,1)
 *
 *  RETURNS:
 *  ERR_NONE on success
 *  I2C error code on I2C errors
 */
char rtc_clearAlarms(char a1, char a2)
{
    unsigned char stat;

    /* write in the status masked with the enable bits */
    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

```

```
    if ((err = CP2_WriteI2C(RTC_DS3231 | I2C_WRITE)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(REG_STATUS)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C((a2 & 0x01) << 1 | a1 & 0x01)) < 0) {
        return err;
    }

    if ((err = CP2_StopI2C()) < 0) {
        return err;
    }

    return ERR_NONE;
}

/*
 * rtc_readTemp(int *temp)
 *
 * PARAMETERS:
 *  int *temp - variable to store the temperature reading in
 *
 * DESCRIPTION:
 *  Reads the temperature from the internal temperature sensor on the DS3231
 *  NOTE: the RTC only does a temperature grab every 64 seconds. if you're
 *        polling this frequently and not noticing a change, that's probably
 *        the reason.
 *
 *  RETURNS:
 *  ERR_NONE on success
 *  I2C error code on I2C errors
 */
char rtc_readTemp(int *temp) {
    unsigned char startReg = REG_RTC_TEMP_MSB;
    char temps[2] = {0, 0};
    int t=0;
    *temp=0;

    if ((err = writeToSlave(RTC_DS3231, &startReg, 1)) < 0) {
        return err;
    }

    if ((err = readFromSlave(RTC_DS3231, 2, (unsigned char*)temps)) < 0) {
        return err;
    }

    *temp = temps[0];
    *temp = (*temp<<2) | (temps[1]>>6);

    t=*temp;

    return ERR_NONE;
}

/*
```

```
 * initControl()
 *
 * PARAMETERS:
 *  void
 *
 * DESCRIPTION:
 *  Sets up the RTC for us to use it. This really only ever needs to be called
 *  once since the RTC is on a battery backup that lasts just about forever,
 *  but it doesn't hurt to do all this stuff every time we restart just to make
 *  sure
 *
 * RETURNS:
 *  0 on success
 *  I2C error code on I2C errors
 */
char initRTC(void)
{
    union rtc_regs rtc;

    /* enable the oscilator (0 is enable) */
    rtc.reg.control.b._eosc = 0;

    /* enables interrupts even in battery backup mode */
    rtc.reg.control.b.bbsqi = 1;

    /* (1) enables interrupts rather than a square wave */
    rtc.reg.control.b.intcn = 1;

    /* we don't use the rate select, but leave it at its default */
    rtc.reg.control.b.rs1 = 1;
    rtc.reg.control.b.rs2 = 1;

    /* disable alarms */
    rtc_alm1_enable = rtc_alm2_enable = 0;
    rtc.reg.control.b.intcn = 1;
    rtc.reg.control.b.a1ie = 0;
    rtc.reg.control.b.a2ie = 0;

    /* write the control bits */
    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(RTC_DS3231 | I2C_WRITE)) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(REG_CONTROL)) < 0) {
        return err;
    }
    if ((err = CP2_WriteI2C(rtc.data[REG_CONTROL])) < 0) {
        return err;
    }

    /* turn off all the alarm flags */
    if ((err = CP2_WriteI2C(0x00)) < 0) {
        return err;
    }
```

```
    if ((err = CP2_StopI2C()) < 0) {
        return err;
    }

    /* the OSF flag is on at poweron- check it to turn it off */
    rtc_checkOSF();

    return ERR_NONE;
}

#ifdef RTC_TEST
/*
 * rtcInterrupt(void)
 *
 * PARAMETERS:
 *  void
 *
 * DESCRIPTION
 *  Timer interrupt handler
 *
 * RETURNS:
 *  void
 */
#pragma interrupt rtcInterrupt
void rtcInterrupt(void)
{
    /* SANITY: make sure we were called by an interrupt */
    if (!INTCONbits.INT0F)
        return;

    /* toggle pin 24 to check the timing */
    LATAbits.LATA0 = !LATAbits.LATA0;

    /*
     * we're being interrupted, but we don't have any alarms set. this means
     * the square wave generator is on. call initControl() to set up things
     * the way we like 'em
     * XXX: disabled for now because calling I2C in interrupts is a bad idea
     if(!rtc_alm1_enable && !rtc_alm2_enable) {
     initControl();
     return;
     }
     */

    rtc_clearAlarms(0, 0);

    INTCONbits.INT0F = 0;
    return;
}

#pragma code

/*
 * interruptHandler()
 *
 * PARAMETERS:
 *  void
```

```
 *
 * DESCRIPTION:
 *  This method is called when an interrupt occurs
 *
 * RETURNS:
 *  void
 */
#pragma interrupt interruptHandler
void interruptHandler()
{
    if (INTCONbits.INT0F) {
        rtcInterrupt();
    }
}

#pragma code

#pragma code high_vector=0x08
void isr(void)
{
    _asm GOTO interruptHandler _endasm
}

#pragma code
void main(void)
{
    union rtc_regs rtc_in1, rtc_in2;
    union rtc_regs rtc_out;
    struct rtc_time alm, set_time;
    char err=0;
    int temp=0;

    if (err = CP2_OpenI2C(MASTER, SLEW_OFF))
        return;
    SSPADD = SSPADD_CLOCK;

    /* initalize the correct control bits */
    err = initRTC();

    set_time.second = 42;
    set_time.minute = 1;
    set_time.hour = 8;
    set_time.ampm = PM;
    set_time.hr_format = TWELVE_HOUR;
    set_time.day = Friday;
    set_time.date = 20;
    set_time.month = April;
    set_time.year = 25;
    set_time.century = 0;

    err = rtc_writeTime(&set_time);

    err = rtc_readRegs(&rtc_in1);
    Delay10KTCYx(100);
    err = rtc_readRegs(&rtc_in2);

    memset(&set_time, 0x00, sizeof(struct rtc_time));
```

```
    rtc_readTime(&set_time);

    /* enable PIC interrupt pin from rtc */
    RCONbits.IPEN = 0;  /* disable interrupt priority */
    INTCONbits.GIE = 1; /* enable interrupts globally */
    INTCONbits.PEIE = 1;        /* enable peripheral interrupts */
    INTCON2bits.INTEDG0 = 0;    /* falling edge trigger */
    INTCONbits.INT0IE = 1;      /* enable interrupt 0 */
    TRISAbits.TRISA0 = 0;       /* make RA0 an output pin */
    TRISBbits.TRISB0 = 1;       /* make INT0 an input pin */

    alm.second = alm.minute = alm.hour = alm.day = alm.date = TIME_MASK;
    if (err = rtc_setAlarm1(&alm) < 0) {
        while (1) ;
    }

    /* spin in circles and wait for interrupts */
    while (1) {
        //err = rtc_readTemp(&temp);
    }

    return;
}
#endif
```

7.4 Real Time Clock

```
/*
 * rtc.h
 *
 * DS3231 RTC driver
 *
 * Author: Jacob Farkas
 * $Id: rtc.h,v 1.1 2004/12/17 20:37:32 kleveque Exp $
 */
#ifndef _RTC_H
#define _RTC_H

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

#include "cp2-common.h"
#include "cp2-i2c.h"
#include "structs.h"

// enable to set up RTC main loop and interrupt handlers for testing
#undef RTC_TEST

#define TWELVE_HOUR     1
#define TWENTYFOUR_HOUR 0
#define AM              0
#define PM              1

#define TIME_MASK       0xFF
```

```
/* REGISTERS */
#define REG_SECOND      0x00
#define REG_MINUTE      0x01
#define REG_HOUR        0x02
#define REG_DAY         0x03
#define REG_DATE        0x04
#define REG_MONTH       0x05
#define REG_YEAR        0x06
#define REG_A1_SECOND   0x07
#define REG_A1_MINUTE   0x08
#define REG_A1_HOUR     0x09
#define REG_A1_DAYDATE  0x0A
#define REG_A2_MINUTE   0x0B
#define REG_A2_HOUR     0x0C
#define REG_A2_DAYDATE  0x0D
#define REG_CONTROL     0x0E
#define REG_STATUS      0x0F
#define REG_AGING       0x10
#define REG_RTC_TEMP_MSB    0x11
#define REG_RTC_TEMP_LSB    0x12

enum month {
    January,
    Feburary,
    March,
    April,
    May,
    June,
    July,
    August,
    September,
    October,
    November,
    December,
};

enum day {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday,
};

/* FUNCTION PROTOTYPES */
char rtc_readRegs(union rtc_regs *);
char rtc_writeTime(struct rtc_time *);
char rtc_readTime(struct rtc_time *);

char rtc_regsToTime(union rtc_regs *, struct rtc_time *);
char rtc_timeToRegs(struct rtc_time *, union rtc_regs *);

char rtc_setAlarm1(struct rtc_time *);
char rtc_setAlarm2(struct rtc_time *);
char rtc_disableAlarms(char, char);
```

```
char rtc_clearAlarms(char, char);
char rtc_checkOSF(void);
char rtc_readTemp(int *);

char initRTC(void);

void rtcInterrupt(void);
void interruptHandler(void);

#endif
```

7.5 Real Time Clock Structure

```
/* RTC */
struct rtc_time {
    unsigned char second;
    unsigned char minute;
    unsigned char hour;
    unsigned char ampm;         /* am or pm for 12 hour */
    unsigned char hr_format;    /* 24 (0) or 12 (1) hour format */
    unsigned char day;          /* day of the week */
    unsigned char date;         /* date of the month */
    /* the following three fields are not used in setting an alarm */
    unsigned char month;
    unsigned char year;
    unsigned char century;
};

union rtc_regs {
    unsigned char data[17];
    struct {
        union {
            unsigned char data;
            struct {
                unsigned second:4;
                unsigned ten_second:3;
                unsigned PADDING:1;
            } b;
        } second;

        union {
            unsigned char data;
            struct {
                unsigned minute:4;
                unsigned ten_minute:3;
                unsigned PADDING:1;
            } b;
        } minute;

        union {
            unsigned char data;
            struct {
                unsigned hour:4;
                unsigned ten_hour:1;
                /* in 24 hour mode, this is another 10 hours */
```

```
            unsigned ampm:1;
            /* sets 12/24 hour mode (hi/lo) */
            unsigned format:1;
            unsigned PADDING:1;
        } b;
    } hour;

    union {
        unsigned char data;
        struct {
            unsigned day:3;
            unsigned PADDING:5;
        } b;
    } day;

    union {
        unsigned char data;
        struct {
            unsigned date:4;
            unsigned ten_date:2;
            unsigned PADDING:2;
        } b;
    } date;

    union {
        unsigned char data;
        struct {
            unsigned month:4;
            unsigned ten_month:1;
            unsigned PADDING:2;
            unsigned century:1;
        } b;
    } month;

    union {
        unsigned char data;
        struct {
            unsigned year:4;
            unsigned ten_year:4;
        } b;
    } year;

    union {
        unsigned char data;
        struct {
            unsigned second:4;
            unsigned ten_second:3;
            unsigned a1m1:1;
        } b;
    } a1_second;

    union {
        unsigned char data;
        struct {
            unsigned minute:4;
            unsigned ten_minute:3;
            unsigned a1m2:1;
        } b;
```

```
        } a1_minute;

        union {
            unsigned char data;
            struct {
                unsigned hour:4;
                unsigned ten_hour:1;
                /* in 24 hour mode, this is another 10 hours */
                unsigned ampm:1;
                /* sets 12/24 hour mode (hi/lo) */
                unsigned format:1;
                unsigned a1m3:1;
            } b;
        } a1_hour;

        union {
            unsigned char data;
            struct {
                unsigned daydate:4;
                unsigned ten_date:2;
                unsigned dydt:1;
                unsigned a1m4:1;
            } b;
        } a1_daydate;

        union {
            unsigned char data;
            struct {
                unsigned minute:4;
                unsigned ten_minute:3;
                unsigned a2m2:1;
            } b;
        } a2_minute;

        union {
            unsigned char data;
            struct {
                unsigned hour:4;
                unsigned ten_hour:1;
                /* in 24 hour mode, this is another 10 hours */
                unsigned ampm:1;
                /* sets 12/24 hour mode (hi/lo) */
                unsigned format:1;
                unsigned a2m3:1;
            } b;
        } a2_hour;

        union {
            unsigned char data;
            struct {
                unsigned daydate:4;
                unsigned ten_date:2;
                unsigned dydt:1;
                unsigned a2m4:1;
            } b;
        } a2_daydate;

        union {
```

```
            unsigned char data;
            struct {
                unsigned a1ie:1;
                unsigned a2ie:1;
                unsigned intcn:1;
                unsigned rs1:1;
                unsigned rs2:1;
                unsigned bbsqi:1;
                unsigned PADDING:1;
                unsigned _eosc:1;
            } b;
        } control;

        union {
            unsigned char data;
            struct {
                unsigned a1f:1;
                unsigned a2f:1;
                unsigned PADDING:5;
                unsigned osf:1;
            } b;
        } status;

        union {
            unsigned char data;
            struct {
                unsigned rout0:1;
                unsigned rout1:1;
                unsigned ds0:1;
                unsigned ds1:1;
                unsigned tcs0:1;
                unsigned tcs1:1;
                unsigned tcs2:1;
                unsigned tcs3:1;
            } b;
        } trickle_charge;
    } reg;
};
```

7.6 I^{2}C Library

```
/*
 * cp2-i2c.h
 * Author: Jacob Farkas <jfarkas@calpoly.edu>
 * $Id: cp2-i2c.h,v 1.11 2005/04/09 17:48:31 jfarkas Exp $
 */

/* NOTE: if you add any IPC_* commands below, be sure to add them
 *       in order as well as updating cp2-i2c.c
 */

/* Format:
 * #define IPC_CPU_COMMANDNAME CNUM
 *             CPU = {COMM,PAYLOAD}
 *             COMMANDNAME = descriptive name
```

```
 *      CNUM (hex) = next sequential command number for the chosen CPU
 *
 * #define IPC_{CPU}_COMMANDNAME_TX_LENGTH     COUNT
 *             COUNT = bytes you expect the master (CDH) to send you
 *
 * #define IPC_{CPU}_COMMANDNAME_RX_LENGTH     COUNT
 *             COUNT = bytes of data you will send back to the master in response
 */

#ifndef _CP2I2C_H
#define _CP2I2C_H

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

#include "cp2-common.h"
#include "cp2-errors.h"

/* COMM STATUS BYTE BITS */
#define COMM_STAT_EN_PL     0x80    /* EN_PL pin */
#define COMM_STAT_SEL_RF    0x40    /* SEL_RF pin */
#define COMM_STAT_SEL_RX    0x20    /* SEL_RX pin */
#define COMM_STAT_SEL_TX    0x10    /* SEL_TX pin */
#define COMM_STAT_XCVR_MODE 0x08    /* transceiver mode: RX=0,TX=1 */
#define COMM_STAT_READY     0x04    /* ready to receive data from cdh? */
#define COMM_STAT_XCVR_CAL  0x02    /* transceiver calibrated? */
#define COMM_STAT_CMD_RECVD 0x01    /* valid command received? */

/* PAYLOAD STATUS BYTES */
#define PAYLOAD_STATUS_PENDINGTEST  0x80
#define PAYLOAD_STATUS_LEAVEMEALONE 0x40
#define PAYLOAD_STATUS_SNAPSHOTRDY  0x20
#define PAYLOAD_STATUS_CLEARALARMS  0x10
#define PAYLOAD_STATUS_RUNNINGTEST  0x08
#define PAYLOAD_STATUS_BUFFERFULL   0x04

/* COMM PIC COMMANDS */
#define IPC_COMM_STATUS                 0x00
#define IPC_COMM_STATUS_TX_LENGTH       0
#define IPC_COMM_STATUS_RX_LENGTH       1

#define IPC_COMM_SENSOR_SNAP            0x01
#define IPC_COMM_SENSOR_SNAP_TX_LENGTH  0
#define IPC_COMM_SENSOR_SNAP_RX_LENGTH  4

#define IPC_COMM_TX_BEACON              0x02
#define IPC_COMM_TX_BEACON_TX_LENGTH    98
#define IPC_COMM_TX_BEACON_RX_LENGTH    1   // status byte

#define IPC_COMM_TX_DATA                0x03
#define IPC_COMM_TX_DATA_TX_LENGTH      98
#define IPC_COMM_TX_DATA_RX_LENGTH      1

/* 225 bytes = 1 sequence number + 25 snaps * 9 bytes/snap */
#define IPC_COMM_TX_PAYLOAD_DATA            0x04
#define IPC_COMM_TX_PAYLOAD_DATA_TX_LENGTH  226
#define IPC_COMM_TX_PAYLOAD_DATA_RX_LENGTH  1
```

```
#define IPC_COMM_TX_PAYLOAD_TEST            0x05
#define IPC_COMM_TX_PAYLOAD_TEST_TX_LENGTH  14
#define IPC_COMM_TX_PAYLOAD_TEST_RX_LENGTH  1

#define IPC_COMM_TX_RTC_TIME            0x06
#define IPC_COMM_TX_RTC_TIME_TX_LENGTH  10  /* sizeof struct rtc_time */
#define IPC_COMM_TX_RTC_TIME_RX_LENGTH  1

#define IPC_COMM_GET_COMMAND            0x07
#define IPC_COMM_GET_COMMAND_TX_LENGTH  0
#define IPC_COMM_GET_COMMAND_RX_LENGTH  25

#define IPC_COMM_ACK_COMMAND            0x08
#define IPC_COMM_ACK_COMMAND_TX_LENGTH  0
#define IPC_COMM_ACK_COMMAND_RX_LENGTH  1

#define IPC_COMM_NACK_COMMAND           0x09
#define IPC_COMM_NACK_COMMAND_TX_LENGTH 0
#define IPC_COMM_NACK_COMMAND_RX_LENGTH 1

#define IPC_COMM_GET_TNC_MODE           0x0A
#define IPC_COMM_GET_TNC_MODE_TX_LENGTH 1
#define IPC_COMM_GET_TNC_MODE_RX_LENGTH 1

#define IPC_COMM_SET_TX_POWER               0x0B
#define IPC_COMM_SET_TX_POWER_TX_LENGTH 1
#define IPC_COMM_SET_TX_POWER_RX_LENGTH 1

/* 200 = 8 adcs snaps * 25 bytes/snap */
#define IPC_COMM_TX_ADCS_DUMP               0x0C
#define IPC_COMM_TX_ADCS_DUMP_TX_LENGTH     200
#define IPC_COMM_TX_ADCS_DUMP_RX_LENGTH     1

/* PAYLOAD PIC COMMANDS */
#define IPC_PAYLOAD_STATUS              0x80
#define IPC_PAYLOAD_STATUS_SIZE_TX      0
#define IPC_PAYLOAD_STATUS_SIZE_RX      1

#define IPC_PAYLOAD_SENSOR_SNAP         0x81
#define IPC_PAYLOAD_SENSOR_SNAP_SIZE_TX 0
#define IPC_PAYLOAD_SENSOR_SNAP_SIZE_RX 6

#define IPC_PAYLOAD_TEST_SNAP           0x82
#define IPC_PAYLOAD_TEST_SNAP_SIZE_TX   0
#define IPC_PAYLOAD_TEST_SNAP_SIZE_RX   9

#define IPC_PAYLOAD_NEW_TEST            0x83
#define IPC_PAYLOAD_NEW_TEST_SIZE_TX    16
#define IPC_PAYLOAD_NEW_TEST_SIZE_RX    0

#define IPC_PAYLOAD_GET_TEST            0x84
#define IPC_PAYLOAD_GET_TEST_SIZE_TX    0
#define IPC_PAYLOAD_GET_TEST_SIZE_RX    16

#define IPC_PAYLOAD_RESTART             0x86
#define IPC_PAYLOAD_RESET_SIZE_TX       0
#define IPC_PAYLOAD_RESET_SIZE_RX       0
```

```
#define IPC_PAYLOAD_CANCEL              0x87
#define IPC_PAYLOAD_CANCEL_SIZE_TX      0
#define IPC_PAYLOAD_CANCEL_SIZE_RX      0

#define IPC_PAYLOAD_NUM_RESET           0x88
#define IPC_PAYLOAD_NUM_RESET_TX        0
#define IPC_PAYLOAD_NUM_RESET_RX        1

/* IPC buffer */
#define IPC_BUF_MAX                256

/* RETRY AND DELAYS */
/* the following macros define the number of times to retry a poll before
 * failing with an error code. this prevents us from ending up in an infinite
 * loop.
 * XXXX_RETRY is the number of times to retry the poll before failing
 * XXXX_DELAY is the Delay10TCYx() amount to delay between each poll
 *  (NOTE: this value is an unsigned char - if you pass a value more than
 *   255 you're going to get unexpected results)
 *  If this value is undef'ed, no delay is called
 */
#define IDLE_RETRY              250
#undef IDLE_DELAY

#define START_RETRY             250
#undef START_DELAY

#define ACK_RETRY               250
#undef ACK_DELAY

#define WRITE_RETRY             250
#undef WRITE_DELAY

#define READ_RETRY              250
#undef READ_DELAY

#define SSPADD_CLOCK    0x09

#define I2C_READ        0x01
#define I2C_WRITE       0x00

/* I2C device addresses */
#define AD_1039         0xCA
#define FLASH_AT24C     0xA0
#define DIGI_AD5245     0x58
#define RTC_DS3231      0xD0

/* IPC device addresses */
#define PIC_COMM            0x5A
#define PIC_COMM_DISABLED   0x5D
#define PIC_PAYLOAD         0x6A
#define PIC_CDH             0x7A

/* enable serial port and configures */
#define SSPENB                  0x20
/* I2C Slave mode, 7-bit address */
#define SLAVE_7                 6
/* I2C Slave mode, 10-bit address */
```

```
#define SLAVE_10                7
/* I2C Master mode */
#define MASTER                  8
/* slew rate disabled for 100kHz mode */
#define SLEW_OFF                0xC0
/* slew rate enabled for 400kHz mode */
#define SLEW_ON                 0x00

/*******************************
 * I2C
 *******************************/
// See ANXXX from microchip for more detail on the I2C state machine
#define I2C_STATE1  0x09    // Master write, prev byte was address
#define I2C_STATE2  0x29    // Master write, prev byte was data
#define I2C_STATE3  0x0C    // Master read, prev byte was address
#define I2C_STATE4  0x2C    // Master read, prev byte was data
#define I2C_STATE5  0x28    // Master NACK

/* DATA */
static int ret;

extern unsigned char i2cTxBuffer[];
extern unsigned char i2cRxBuffer[];

/* PROTOTYPES */
#define I2C_IDLE()    if(ret=CP2_IdleI2C()) { return ret; }

char CP2_OpenI2C(unsigned char, unsigned char);
char CP2_CloseI2C(void);

char CP2_IdleI2C(void);

char CP2_StartI2C(void);
char CP2_StopI2C(void);
char CP2_RestartI2C(void);

char CP2_AckI2C(void);
char CP2_NoAckI2C(void);
char CP2_CheckAckI2C(void);
char CP2_AckPollI2C(unsigned char);

char CP2_WriteI2C(unsigned char);
char CP2_ReadI2C(unsigned char*);

char readFromSlave(unsigned char address, int readLength, unsigned char* buffer);
char writeToSlave(unsigned char address, unsigned char data[], int dataLength);
int transferI2C (unsigned char addr, unsigned char command);

#endif
```

7.7 I^{2}C Library

Note: `transferI2C` was written by Chris Noe

```
/*
```

```
 * cp2-i2c.c
 * Author: Jacob Farkas <jfarkas@calpoly.edu>
 * $Id: cp2-i2c.c,v 1.6 2005/02/21 02:03:27 cnoe Exp $
 *
 * TODO:
 * -change all functions to the error handling specefications
 */

#include "cp2-i2c.h"

#pragma udata I2CTXBUF
unsigned char i2cTxBuffer[IPC_BUF_MAX];
#pragma udata

#pragma udata I2CRXBUF
unsigned char i2cRxBuffer[IPC_BUF_MAX];
#pragma udata

/* drunken yak */
/* global err, used to return error codes */
char err;

// this table maps commands to their expected TX/RX lengths
#define IPC_COMM_COMMAND_COUNT      13
#define IPC_PAYLOAD_COMMAND_COUNT   5
#define IPC_COMMAND_TABLE_WIDTH     2

unsigned char
  commCommandTable[IPC_COMM_COMMAND_COUNT][IPC_COMMAND_TABLE_WIDTH] = {
    {IPC_COMM_STATUS_TX_LENGTH,             IPC_COMM_STATUS_RX_LENGTH},
    {IPC_COMM_SENSOR_SNAP_TX_LENGTH,        IPC_COMM_SENSOR_SNAP_RX_LENGTH},

    {IPC_COMM_TX_BEACON_TX_LENGTH,          IPC_COMM_TX_BEACON_RX_LENGTH},
    {IPC_COMM_TX_DATA_TX_LENGTH,            IPC_COMM_TX_DATA_RX_LENGTH},
    {IPC_COMM_TX_PAYLOAD_DATA_TX_LENGTH,    IPC_COMM_TX_PAYLOAD_DATA_RX_LENGTH},
    {IPC_COMM_TX_PAYLOAD_TEST_TX_LENGTH,    IPC_COMM_TX_PAYLOAD_TEST_RX_LENGTH},
    {IPC_COMM_TX_RTC_TIME_TX_LENGTH,        IPC_COMM_TX_RTC_TIME_RX_LENGTH},

    {IPC_COMM_GET_COMMAND_TX_LENGTH,        IPC_COMM_GET_COMMAND_RX_LENGTH},
    {IPC_COMM_ACK_COMMAND_TX_LENGTH,        IPC_COMM_ACK_COMMAND_RX_LENGTH},
    {IPC_COMM_NACK_COMMAND_TX_LENGTH,       IPC_COMM_NACK_COMMAND_RX_LENGTH},

    {IPC_COMM_GET_TNC_MODE_TX_LENGTH,       IPC_COMM_GET_TNC_MODE_RX_LENGTH},
    {IPC_COMM_SET_TX_POWER_TX_LENGTH,       IPC_COMM_SET_TX_POWER_RX_LENGTH},

    {IPC_COMM_TX_ADCS_DUMP_TX_LENGTH,       IPC_COMM_TX_ADCS_DUMP_RX_LENGTH},
};

unsigned char
  payloadCommandTable[IPC_PAYLOAD_COMMAND_COUNT][IPC_COMMAND_TABLE_WIDTH] = {
    {IPC_PAYLOAD_STATUS_SIZE_TX,        IPC_PAYLOAD_STATUS_SIZE_RX},
    {IPC_PAYLOAD_SENSOR_SNAP_SIZE_TX,   IPC_PAYLOAD_SENSOR_SNAP_SIZE_RX},
    {IPC_PAYLOAD_TEST_SNAP_SIZE_TX,     IPC_PAYLOAD_TEST_SNAP_SIZE_RX},
    {IPC_PAYLOAD_NEW_TEST_SIZE_TX,      IPC_PAYLOAD_NEW_TEST_SIZE_RX},
    {IPC_PAYLOAD_GET_TEST_SIZE_TX,      IPC_PAYLOAD_GET_TEST_SIZE_RX},
};

/*
```

```
 * FUNCTION:
 *  CP2_OpenI2C
 *
 * DESCRIPTION:
 *  Sets up a PIC for I2C mode.
 *
 * PARAMETERS:
 *  unsigned char mode
 *      PIC master or slave mode (SSPCON1)
 *
 *  unsigned char slew
 *      PIC slew rate setting (SSPSTAT)
 *
 * RETURNS:
 *  0 always
 */
char CP2_OpenI2C(unsigned char mode, unsigned char slew)
{
    /* reset to power on states */
    SSPSTAT &= 0x3F;
    SSPCON1 = 0x00;
    SSPCON2 = 0x00;

    /* select serial mode */
    SSPCON1 |= mode;
    /* slew rate on/off */
    SSPSTAT |= slew;

    /* set SCL (PORTC,3) pin to input */
    DDRCbits.RC3 = 1;
    /* set SDA (PORTC,4) pin to input */
    DDRCbits.RC4 = 1;

    /* enable synchronous serial port */
    SSPCON1 |= SSPENB;

    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_CloseI2C
 *
 * DESCRIPTION:
 *  Turns off the PIC I2C mode. The I2C pins are turned back into general I/O
 *   pins.
 *  We shouldn't ever use this method.
 *
 * PARAMETERS:
 *  none
 *
 * RETURNS:
 *  ERR_NONE always
 */
char CP2_CloseI2C(void)
{
    SSPCON1 &= 0xDF;
```

```
    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_IdleI2C
 *
 * DESCRIPTION:
 *  Polls the PIC pins IDLE_RETRY times until an idle condition is achieved
 *  All of the following I2C methods call IdleI2C, so you shouldn't have to
 *   call this method yourself
 *
 *  IdleI2C checks for bus collisions
 *
 * PARAMETERS:
 *  none
 *
 * RETURNS:
 *  0 if an idle condition was achieved
 *  I2C error code otherwise
 */
char CP2_IdleI2C(void)
{
    unsigned int idlectr;

    for (idlectr = 0; idlectr < IDLE_RETRY; idlectr++) {
        if (PIR2bits.BCLIF) {
            return ERR_I2C_BCLIF;
        }

        /* check the necessary bits for an idle condition */
        if (!((SSPCON2 & 0x1F) | (SSPSTATbits.R_W))) {
            return ERR_NONE;
        }
#ifdef IDLE_DELAY
        Delay10TCYx(IDLE_DELAY);
#endif
    }

    /* this is where error codes start their life */
    if (SSPCON2 & 0x10) {
        return ERR_I2C_NO_ACK;
    }

    if (SSPCON2 & 0x08) {
        return ERR_I2C_NO_RECV;
    }

    if (SSPCON2 & 0x04) {
        return ERR_I2C_NO_STOP;
    }

    if (SSPCON2 & 0x02) {
        return ERR_I2C_NO_RSTRT;
    }

    if (SSPCON2 & 0x01) {
        return ERR_I2C_NO_STRT;
```

```
    }

    if (SSPSTATbits.R_W) {
        return ERR_I2C_NO_XMIT;
    }

    return ERR_I2C_UNKNOWN;
}

/*
 * FUNCTION:
 *  CP2_StartI2C
 *
 * DESCRIPTION:
 *  Starts an I2C transaction on the bus
 *
 * PARAMETERS:
 *  none
 *
 * RETURNS:
 *  ERR_NONE if the start condition succeeded
 *  I2C error code otherwise
 */
char CP2_StartI2C(void)
{
    unsigned int idlectr;

    I2C_IDLE();

    /* send the bus start condition. this should be cleared by hardware */
    SSPCON2bits.SEN = 1;

    /* XXX: if we have a bus collision interrupt enabled we need to change this
     * code
     */
    /* the start condition could have generated a bus collision- check */
    for (idlectr = 0; idlectr < START_RETRY; idlectr++) {
        if (!PIR2bits.BCLIF) {
            break;
        }

        /* clear the bus collision flag */
        PIR2bits.BCLIF = 0;

#ifdef START_DELAY
        Delay10TCYx(START_DELAY);
#endif

        I2C_IDLE();

        /* try the start condition again */
        SSPCON2bits.SEN = 1;
    }
    /* if we couldn't successfully grab hold of the bus an I2C_ERR_BCLIF will
     * be passed up from here
     */
    I2C_IDLE();
```

```
    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_StopI2C
 *
 * DESCRIPTION:
 *  Sends an I2C stop condition
 *
 * PARAMETERS:
 *  none
 *
 * RETURNS:
 *  0 if the stop condition succeeded
 *  I2C error code otherwise
 */
char CP2_StopI2C(void)
{
    /* initiate stop condition - automatically cleared by hardware */
    SSPCON2bits.PEN = 1;
    I2C_IDLE();

    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_RestartI2C
 *
 * DESCRIPTION:
 *  Sends an I2C restart condition
 *
 * PARAMETERS:
 *  none
 *
 * RETURNS:
 *  0 if the restart condition succeeded
 *  I2C error code otherwise
 */
char CP2_RestartI2C(void)
{
    /* initiate restart condition - cleared by hardware */
    SSPCON2bits.RSEN = 1;
    I2C_IDLE();

    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_AckI2C
 *
 * DESCRIPTION:
 *  Sends a master recieve acknowledge on the I2C bus
 *
 * PARAMETERS:
 *  none
```

```
 *
 * RETURNS:
 *  0 if the acknowledge succeeded
 *  I2C error code otherwise
 */
char CP2_AckI2C(void)
{
    /* enable acknowledge bit (active low) */
    SSPCON2bits.ACKDT = 0;

    /* initiate the acknowledge sequence on I2C */
    SSPCON2bits.ACKEN = 1;
    I2C_IDLE();

    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_NoAckI2C
 *
 * DESCRIPTION:
 *  Sends a noack on the I2C bus
 *
 * PARAMETERS:
 *  none
 *
 * RETURNS:
 *  0 if the noack succeeded
 *  I2C error code otherwise
 */
char CP2_NoAckI2C(void)
{
    /* set the acknowledge bit */
    SSPCON2bits.ACKDT = 1;

    /* initiate the acknowledge sequence */
    SSPCON2bits.ACKEN = 1;
    I2C_IDLE();

    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_CheckAckI2C
 *
 * DESCRIPTION:
 *  Checks for an acknowledge from the slave
 *
 * PARAMETERS:
 *  void
 *
 * RETURNS:
 *  0 if the slave acknowledged
 *  I2C error code otherwise
 */
char CP2_CheckAckI2C(void)
```

```
{
    if (SSPCON2bits.ACKSTAT) {
        return ERR_I2C_NO_ACK;
    }

/* this idle isn't really necessary */
    I2C_IDLE();

    return ERR_NONE;
}

/*
 * FUNCTION:
 *  CP2_AckPollI2C
 *
 * DESCRIPTION:
 *  Polls a device until it acknowledges
 *
 * PARAMETERS:
 *  unsigned char dev
 *      The device address to poll
 *
 * RETURNS:
 *  0 when the device acknowledges
 *  I2C error code otherwise
 */
char CP2_AckPollI2C(unsigned char dev)
{
    unsigned int idlectr;

    /* we expect errors to occur in here- ignore everything except bus
     * collisions */
    if (ERR_I2C_BCLIF == CP2_StartI2C()) {
        return ERR_I2C_BCLIF;
    }

    for (idlectr = 0; idlectr < ACK_RETRY; idlectr++) {
        if (ERR_I2C_BCLIF == CP2_WriteI2C(dev)) {
            return ERR_I2C_BCLIF;
        }

        /* check for an acknowledge */
        if (!(SSPCON2bits.ACKSTAT)) {
            CP2_StopI2C();
            return ERR_NONE;
        }

        /* restart and try again */
        if (ERR_I2C_BCLIF == CP2_RestartI2C()) {
            return ERR_I2C_BCLIF;
        }

#ifdef ACK_DELAY
        Delay10TCYx(ACK_DELAY);
#endif
    }

    /* the device wouldn't respond */
```

```
    return ERR_I2C_ACK_RETRY;
}

/*
 * FUNCTION:
 *  CP2_WriteI2C
 *
 * DESCRIPTION:
 *  Writes the given byte to the I2C bus
 *
 * PARAMETERS:
 *  unsigned char data
 *      The data byte to write to I2C
 *
 * RETURNS:
 *  ERR_NONE on success
 *  I2C error code otherwise
 */
char CP2_WriteI2C(unsigned char data)
{
    unsigned int idlectr;

    /* put the char into the outgoing buffer */
    SSPBUF = data;

    for (idlectr = 0; idlectr < WRITE_RETRY; idlectr++) {
        if (!(SSPSTATbits.BF)) {
            /* the buffer has flushed */
            I2C_IDLE();
            return ERR_NONE;
        }
#ifdef WRITE_DELAY
        Delay10TCYx(WRITE_DELAY);
#endif
    }

    return ERR_I2C_BF;
}

/*
 * FUNCTION:
 *  CP2_ReadI2C
 *
 * DESCRIPTION:
 *  Reads a byte from I2C and puts it in the given char*
 *
 * PARAMETERS:
 *  unsigned char *data
 *      Pointer to a char to fill with data
 *
 * RETURNS:
 *  0 on success
 *  I2C error code otherwise. On error, *data is also set to 0
 */
char CP2_ReadI2C(unsigned char *data)
{
    unsigned int idlectr;

```

```
    /* reset the data being passed in */
    *data=0;

    /* turn on the recieve enable bit */
    SSPCON2bits.RCEN = 1;

    for (idlectr = 0; idlectr < READ_RETRY; idlectr++) {
        /* when SSPSTATbits.BF is empty, the read is complete */
        if (SSPSTATbits.BF) {
            *data = SSPBUF;
            I2C_IDLE();

            return ERR_NONE;
        }
#ifdef READ_DELAY
        Delay10TCYx(READ_DELAY);
#endif
    }

    return ERR_I2C_READ_TIMEOUT;
}

/*
 * FUNCTION:
 *  readFromSlave
 *
 * DESCRIPTION:
 *  Reads a given number of bytes from a device
 *
 * PARAMETERS:
 *  unsigned char address
 *      Device address to read from
 *
 *  int readLength
 *      Number of bytes to read
 *
 *  unsigned char *buffer
 *      Buffer to put the read bytes into. This must be at least readLength
 *       long.
 *
 * RETURNS:
 *  ERR_NONE on success
 *  I2C error code otherwise
 */
char readFromSlave(unsigned char address, int readLength, unsigned char *buffer)
{
    unsigned int idlectr;
    unsigned char *cptr;
    int i;
    char err;

    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(address | I2C_READ)) < 0) {
        CP2_StopI2C();
        return err;
```

```
    }

//  I2C capacitance causes this to return error code incorrectly
//      if(err = CP2_CheckAckI2C()) {
//              CP2_StopI2C();
//              return err;
//      }

    for (i = 0; i < readLength; i++) {
        cptr = buffer + i;
        if ((err = CP2_ReadI2C(cptr)) < 0) {
            return err;
        }

        if (i != readLength - 1) {
            if ((err = CP2_AckI2C()) < 0) {
                return err;
            }
        }
    }
    CP2_StopI2C();

    return err;
}

/*
 * FUNCTION:
 *  writeToSlave
 *
 * DESCRIPTION:
 *  Writes a given number of bytes to a device
 *
 * PARAMETERS:
 *  unsigned char address
 *      Device address to write to
 *
 *  unsigned char data[]
 *      Buffer to write from. This must be at least dataLength long.
 *
 * int dataLength
 *      Number of bytes to write
 *
 * RETURNS:
 *  0 on success
 *  I2C error code otherwise
 */
char writeToSlave(unsigned char address, unsigned char data[], int dataLength)
{
    int index;

    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }
    if ((err = CP2_WriteI2C(address | I2C_WRITE)) < 0) {
        CP2_StopI2C();
        return err;
    }
    if ((err = CP2_CheckAckI2C()) < 0) {
```

```
        CP2_StopI2C();
        return err;
    }

    for (index = 0; index < dataLength; index++) {
        if ((err = CP2_WriteI2C(data[index])) < 0) {
            CP2_StopI2C();
            return err;
        }
        // CheckACK?
    }

    CP2_StopI2C();

    return err;

}

/*
 * FUNCTION:
 *  transferI2C
 *
 * DESCRIPTION:
 * transferI2C() implements our generic I2C communication protocol
 * it consists of 2 steps
 * 1: the master writes a single byte command to the slave, along with any
 *         data associated with that command (up to 256 bytes, currently).
 *    the data is stored in buf, and is txlength long
 * 2: the master reads from the slave. first byte read is the length of the
 *    complete i2c transaction, followed by 1 or more bytes of data
 *
 * comment: jfarkas and cnoe discussion leads us to the decision to implement
 * transferI2C separately from read/writeToSlave. the reason is that the command
 * byte must be the first byte of the write transaction to the slave, followed
 * immediately by the transaction data (if any). writeToSlave doesn't allow for
 * this without two separate transactions, which won't work with our IPC.
 *
 * PARAMETERS:
 *  unsigned char addr
 *      Device address to write to
 *
 *  unsigned char command
 *      Device command. Determines what is read/written
 *
 * RETURNS:
 *  0 on success
 *  I2C error code otherwise
 */
int transferI2C(unsigned char addr, unsigned char command)
{
    unsigned char txLength, rxLength;
    unsigned char i;
    unsigned char *destBuffer;

    // clear RX buffer
    memset(i2cRxBuffer, 0x00, IPC_BUF_MAX);

    if (command & 0x80) {
```

```
        txLength = payloadCommandTable[command & 0x0F][0];
    } else {
        txLength = commCommandTable[command][0];
    }

    /* first, write the command + command data to the slave */
    if ((err = CP2_StartI2C()) < 0) {
        return err;
    }

    if ((err = CP2_WriteI2C(addr | I2C_WRITE)) < 0) {
        CP2_StopI2C();
        return err;
    }

    if ((err = CP2_CheckAckI2C()) < 0) {
        CP2_StopI2C();
        return err;
    }

    if ((err = CP2_WriteI2C(command)) < 0) {      /* command byte */
        CP2_StopI2C();
        return err;
    }

    if ((err = CP2_CheckAckI2C()) < 0) {
        CP2_StopI2C();
        return err;
    }

    for (i = 0; i < txLength; i++) {
        if ((err = CP2_WriteI2C(i2cTxBuffer[i])) < 0) {
            CP2_StopI2C();
            return err;
        }
//              if(i=CP2_CheckAckI2C())
//                      return i;
    }
    CP2_StopI2C();

    // read back
    if ((err = CP2_StartI2C()) < 0) {
        CP2_StopI2C();
        return err;
    }

    if ((err = CP2_WriteI2C(addr | I2C_READ)) < 0) {
        CP2_StopI2C();
        return err;
    }

    if ((err = CP2_CheckAckI2C()) < 0) {
        CP2_StopI2C();
        return err;
    }

    /* XXX: what happens if we issue a stop condition while the slave is still
     * transferring? Will the slave pick up on the stop condition and stop, or
```

```
     * will it keep trying to send data?
     */
    if (!(err = CP2_ReadI2C(&rxLength))) {
        if (rxLength > 0 && rxLength <= IPC_BUF_MAX) {
            if ((err = CP2_AckI2C()) < 0) {
                CP2_StopI2C();
                return err;
            }

            for (i = 0; i < rxLength; i++) {
                if ((err = CP2_ReadI2C(i2cRxBuffer + i)) < 0) {
                    CP2_StopI2C();
                    return err;
                }

                if (i != rxLength - 1) {
                    if ((err = CP2_AckI2C()) < 0) {
                        CP2_StopI2C();
                        return err;
                    }
                }
            }
        }
    } else {
        CP2_StopI2C();
        return err;
    }

    CP2_NoAckI2C();
    CP2_StopI2C();

    return ERR_NONE;
}
```

7.8 I^{2}C Slave Code

Note: Most of this design is credited to Chris Noe

```
unsigned char i2cActivityDetect;
unsigned char i2cCommand;       // Last I2C command recv'd
unsigned char commandReceived;   // Have we received a command?
static unsigned char i2cBufferIndex;    // Index into current I2C buffer
static unsigned char i;

void i2cISR(void) {
    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
            sendingSnap=NULL;
            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) {
                i2cRxBuffer[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 IPC_PAYLOAD_ROTATE:
                    data=IPC_PAYLOAD_ROTATE_RX;
                    break;
                default:
                    break;
            }

            while (SSPSTATbits.BF)      /* Wait for xmit buffer to empty */
            { }

            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 IPC_PAYLOAD_ROTATE:
                    data=NO_DATA_FILLER;
                    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;
}
```

Bibliography

[1] Heidt, H., Puig-Suari, J., Moore, A.S., Nakasuka, S., Twiggs, R.J., CubeSat: A New Generation of Picosatellite for Education and Industry LowCost Space Experimentation", Proceedings of the Utah State University Small Satellite Conference, Logan, UT, August 2001, pp. 1-2, 6.

[2] Schaffner, Jake A, "The Electronic System Design, Analysis, Integration, and Construction of the Cal Poly State University CP1 CubeSat". 16th AIAA/USU Conference on Small Satellites

[3] Noe, Chris. "Design and Implementation of the Communications Subsystem for the Cal Poly CP2 Cubesat Project" Cal Poly Senior Project.

[4] Day, Chris. "The Design of an Efficient, Elegant, and Cubic Pico-Satellite Electronics System" Cal Poly Masters Thesis

[5] Toorian, Armen. Personal interview. June 13, 2005.