Enhancements to the CPX I^2C Bus

Keith McCabe
California Polytechnic State University, San Luis Obispo

December 10, 2007

Abstract

The CPX Bus is a standardized CubeSat bus, consisting of a power subsystem, communications subsystem, and Command & Data handling (C&DH) subsystem. The CPX Bus uses three PIC18 microprocessors to accomplish both C&DH and comm functions, all of which are connected by a shared I^2C bus. This paper describes, in detail, the current state of the hardware and software that runs the CPX I^2C bus. Additionally, this paper proposes a software upgrade to the CPX I^2C bus, significantly increasing the modularity, robustness, and ease-of-use of the current software. The upgrade also introduces limited profiling of the I^2C bus and the ability to download this data to earth. Further, this paper describes how to add an IPC command to the CPX I^2C bus.

Contents

Chapter 1

Introduction

Since most CubeSats use low-power, low-speed microprocessors, simple serial buses, such as I^2C, SPI and UART, are commonly used as a main communications channel. Case in point: the CPX Bus uses I^2C as both a common bus for sensors as well as a means for communicating between the C&DH and communications processors. This paper describes the current Interprocessor Communications protocol and discusses its strengths and weaknesses. This paper goes on to design and implement changes in an attempt to address the weaknesses of the current design.

1.1 CubeSat Project Introduction

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

The short development cycle is made possible by the small mass and simple volume specifications of a CubeSat. With a simple, standardized satellite geometry, undergraduate university students have the opportunity to design, build, test, and actually see their satellite function in space, all within the time frame of an average college student (4-5 years). Additionally, the responsive nature of CubeSats has benefits to industry as well, and recent years have seen increased corporate involvement in the CubeSat community.

The CubeSat project has seen two successful launches in the past year, including the GeneSat launch in December of 2006, and the DNEPR Launch on April 17, 2007, which carried both CP3 and CP4 to orbit. The CubeSat community is currently looking forward to at least one launch in mid 2008.

1.2 CubeSats Developed at Cal Poly

1.2.1 CP1

CP1 marks Cal Poly's first foray as a CubeSat Developer; it was initially used to help the P-POD development team understand the issues that external CubeSat Developers were facing[3]. The CP1 core team was comprised of seven Cal Poly students from aerospace and electrical engineering; the software was written without any formal experience in software development.

Figure 1.1: CP1: Cal Poly's First Satellite

CP1's only flight model was lost on July 26th, 2007, when the launch vehicle failed.

1.2.2 CP2/CP4

After CP1's development finished, a new team was formed at Cal Poly dedicated solely to building CubeSats. The PolySat project was comprised of many members of the original CP1 team. CP2 was PolySat's effort to create a satellite with a standardized bus, usable for future CubeSats. The electronics and structure were developed from scratch by Cal Poly students. CP2 also marked the beginning of a dedicated software team. The satellite software is primarily written in C (as opposed to CP1, which used BASIC).

The first CP2 flight model was lost on July 26th, 2007, when the launch vehicle failed. However, the second model was manifested as CP4 on the second DNEPR launch, and successfully made it to oribt.

1.2.3 CP3

CP3's development began near the end of the CP2 development cycle. Many lessons from the development of CP2 went into CP3, resulting in several performance improvements.

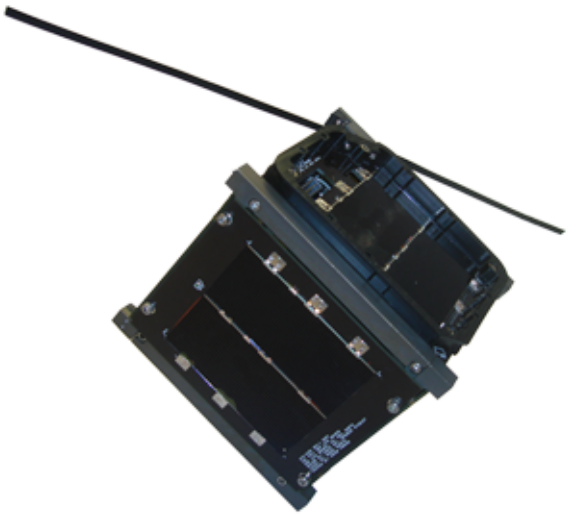

Figure 1.2: CP2: Cal Poly's Second Satellite

As you can see from figure 1.2 and figure 1.4, the two satellites are externally very similar, and are identified by the shape of their solar cells. The only major difference between CP2 and CP3 is the payload that each flew.

CP3 was launched on a DNEPR rocket on April 17, 2007 and has been working since.

Figure 1.3: CP4: Not Cal Poly's Fourth Satellite. Image taken by AeroCube-2 shortly after deployment

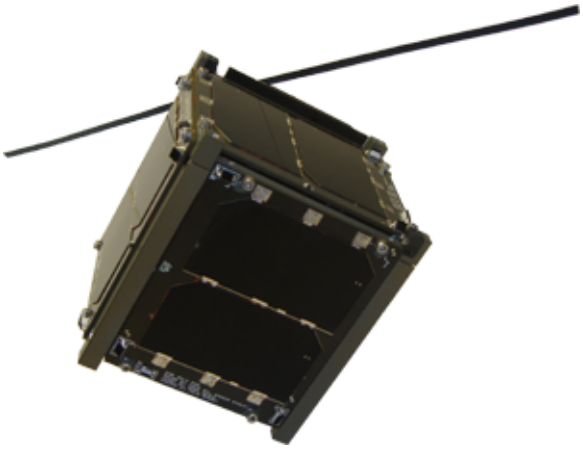

Figure 1.4: CP3: Cal Poly's Third Satellite

Chapter 2

CPX I^2C Bus

The CPX I^2C bus is the main communication channel for the CPX Satellite Bus. Digital bus data, with the exception of the battery monitors, uses the I^2C bus. The devices hanging off the I^2C bus include:

C&DH PIC	C&DH Board
COMM A PIC	C&DH Board
COMM B PIC	C&DH Board
EEPROM #1	C&DH Board
EEPROM #2	C&DH Board
Power ADC	Power Board
C&DH ADC	C&DH Board
Side Panel ADC	Side Panels
Magnetorquer/Magnetometer Controller	Side Panels

We can divide these devices into two distinct groups: "dumb" slaves, of which we have no control over the I^2C module for (such as the ADCs) and "smart" slaves, which we have to write I^2C code for. The latter group contains the comm controllers (which have identical code) and the payload processor, which changes from mission to mission.

Interprocessor Communication refers to passing messages between "smart" slaves and the C&DH processor as it acts as the I^2C Master. While the I^2C specification does include support for multiple masters on the same bus, having a device act as both a master and addressable slave is very difficult. Thus, this paper assumes that there is a single master on the bus, being the C&DH controller.

2.1 Interprocessor Communication

The IPC code is built upon the modular I^2C libraries written by Jacob Farkas[3]. These abstract away most of the PIC-specific I^2C details, providing a byte-oriented interface to the I^2C master. The I^2C slaves need to provide their own ISR to use the bus.

The IPC protocol uses two I^2C transactions to accomplish a single message. First, the Master addresses the slave and writes the command byte as well as any associated data. Then, the Master immediately initiates a Master-Receive transaction, and reads the slave's response to the command.

Each side of the IPC is discussed in depth below.

2.1.1 Master

The Master side of the IPC is implemented in the function transferI2C(), in the file shared/cp2-i2c.c. The function's code in its entirety can be seen in appendix B.1.

The function uses four inputs: the two arguments, which are the slave address and IPC command to send, as well as two global buffers (i2cRxBuffer and i2cTxBuffer), which are used to as a data source or sink for transmit and receive, respectively. Additionally, the function uses two tables to map commands to their expected message recieve and transmit lengths.

First, the master uses the command byte to index into the command length table to determine how long the Master-Transmit should be. This table maps command numbers to their expected recieve and transmit lengths. Since the transmit length is pulled out of the table, it is not possible to have variable length Master-Transmit messages. Additionally, the recieve length is only used by the slave in some cases, and is not checked by the master.

Next, the master initiates the Master-Transmit. This is a fairly straightforward piece of code, except for the issue of checking slave acknowldgements. Due to hardware issues on CP2, CP2_CheckAck used to return an erroneous value, and was commented out. However, on the newer revisions of the C&DH board, this function now works correctly.

Once the Master-Transmit transaction has stopped, the master immediately starts a Master-Recieve transaction to the same slave. The master uses the first data byte as the transaction length, and reads that number of bytes from the bus. The master sends a NoACK pulse in response to the last data byte, in accordance with the I^2C specification.

2.1.2 Slave

Since the CPX bus will almost always include a communications system, I am using the comm controllers as an example slave. Other slaves on the CPX include payload processors and possibly side-panel controllers in the future.

The code that handles the IPC for the Comm controllers is displayed in appendix B.2. Since this is an Interrupt Service Routine (ISR), it is heavily intermixed with the PIC18's hardware I^2C module. For the purpose of this discussion, this paper will gloss over most of the gory hardware details. Consult the PIC18 datasheet for more information about the hardware module.

The ISR has five states: Master-Transmit after the address (I2C_STATE1), Master-Transmit after a data byte (I2C_STATE2), Master-Receive after the address (I2C_STATE3),

Master-Receive after a data byte (I2C_STATE4), and Master-NoACK (I2C_STATE5). The diagrams in figure 2.2 and figure 2.1, taken from the PIC18 Datasheet, display when interrupts are fired, and which state the ISR uses for that interrupt.

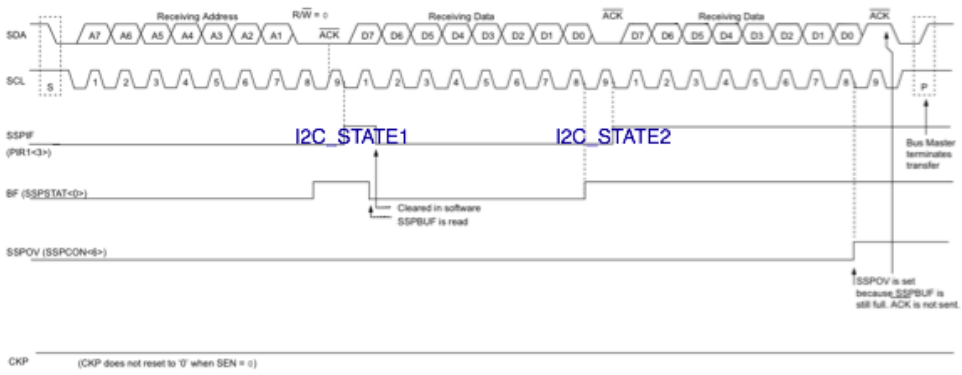

Figure 2.1: PIC18 I^{2}C ISR for a Master-Transmit transaction

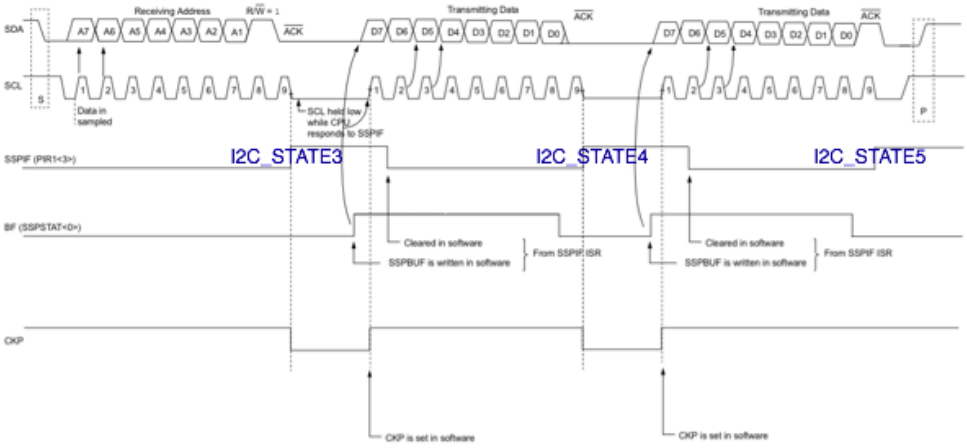

Figure 2.2: PIC18 I^{2}C ISR for a Master-Receive transaction

The two states handling Master-Transmits are straightforward, dumping the command byte into a special variable and the message into the i2cRxBuffer. Since no interrupt is

generated at the end of a Master-Transmit, processing of the command is postponed until the Master-Receive.

The Master-Receive states each contain a switch statement based on the command byte sent in the previous Master-Transmit transaction. Each case loads up the data variable, which is written to the I^2C register. During the Master-Receive after a data byte, most states set a bit in either the commTxFlags or commPayloadTxFlags variables. These are the mechanism with which the ISR communicates with the main loop of the comm controller; these variables are checked, and action is taken based on which bits are set.

2.2 Problems

This protocol assumes that, for a given command, the slave controller can come up with a response nearly immediately. In the comm controller's case, this is true; the main loop is not involved in loading data for the Master-Receive. However, for slaves that operate under a command-process-respond paradigm, our IPC requires at least four transactions– the Master-Transmit/Receive pair for the command, and the Master-Transmit/Receive pair once the data has been processed and made available. The first Master-Receive and the second Master-Transmit serve no purpose, other than being required by the IPC protocol.

In order to support the "immediate response" functionality required by our IPC, the comm controller intermixes the IPC commands with the low-level I^2C ISR. This makes the ISR difficult to maintain; adding a new IPC command requires adding it to two different switch statements, as well as making space in the flags, and adding handlers in the main loop for those flags.

Additionally, this protocol contains no safeguards against bit-flips on the wire. While lab testing has not shown significant error rates, it is difficult to predict the results in the harsh space environment, with large temperature changes and exposure to radaition. Due to the on-orbit failure of CP4, there is considerable external pressure to measure how often erroneous I^2C transactions happen on-orbit.

2.3 On-Orbit Performance

Both CP3 and CP4 have been in orbit for over six months as of this writing. The Cal Poly Operations team has been experiencing difficulties uplinking to both satellites, due to the transceivers being less sensitive than we had anticipated. After upgrading and tuning our groundstation, we experienced somewhat-acceptable uplink rates to both satellites.

CP4 suffered a failure several months ago and can no longer communicate meaningful telemetry to earth. The current hypothesis is that the I^2C bus has been rendered inoperable, either by hardware or software failure, and the comm controllers can no longer communicate with the C&DH processor. It is unknown whether the problem will ever fix

itself, but the Cal Poly team continues to contact CP4 to asses the health of the comm subsystem.

Although CP3's primary mission remains unfulfilled, the satellite bus continues to operate. We continue to receive data dumps, and all systems are operating well within normal parameters without noticeable degradation.

Chapter 3

Modifications

3.1 Changes

3.1.1 One Message, One Transaction

By moving to one I^2C transaction per message, we can address the issue of wasting transactions for certain types of slaves. The on-the-wire representation is also simpler, which aids in debugging when using an oscilloscope or logic analyzer.

On the downside, the comm controller interface loses efficiency; since messages are no longer "bidirectional", there is extra overhead to manually write to and subsequently read from the comm controller. However, the comm controller API stays nearly identical, as most of the changes necessary are underneath the preexisting cdh-comm API. Since the satellite's performance is more than acceptable at this time, the slight performance hit is not expected to cause any problems.

3.1.2 Error Detection

In order to detect bit flips on the bus, an 8-bit CRC was added to each message. Since the maximum buffer size on either end is 256 bytes, an 8-bit CRC was deemed to be sufficient. The ATM CRC-8 Polynomial[4] was chosen for the CRC calculation, though switching polynomials is trivial.

The error detection is the basis for calculating message acceptance and rejection statistics. Every time a message is passed between the processors, both sides increment the message count. If the CRC fails, the rejected count is incremented. If the CRC succeeds, the message is marked as accepted. Hardware failures, such as a slave failing to Ack or other, more esoteric problems, are counted separately. These statistics can be retrieved via a groundstation command.

3.1.3 Redesigned API

Since the large switch statements in the comm controller's ISR were difficult to maintain, and the use of global buffers on the C&DH side introduced a lot of stupid bugs into new code, both APIs were redesigned.

On the master side, there are two new functions: readFromCPU() and writeToCPU(). They read from and write to a slave CPU, respectively. These functions also take care of calculating the message CRC, in the case of writeToCPU(), and checking it against the received data, in the case of readFromCPU(). In certain cases, readFromCPU will attempt to re-request the data if it receives a message with a bad CRC.

For the slave side, there are now two functions: writeI2C() and readI2C(). However, due to the asynchronous nature of an I^2C slave device, these functions require some explanation.

readI2C() has an associated global flag, i2cMessageReceived, which indicates when "fresh" data will be returned. Calling readI2C() with i2cMessageRecieved false will return the last message the bus recieved, although this situation is prone to concurrency issues and not recommended. If the master attempts to write to the slave while i2cMessageReceived is true, the data will be silently discarded.

writeI2C() simply sets the slave response to the specified message. The slave response is reset back to the default message once the specified response has been fully written out in a Master-Receive transaction. Since the slave normally sets the slave response as a result of receiving a message from the master, there is no need for the slave to be able to check whether the response has been "used" or not.

3.2 System Impacts

As a result of the changes, there was both a redistribution of responsibility between masters and slaves, as well as some subtle performance changes. Specifically, the C&DH controller now manually handles the Comm controller's I^2C response. The comm controller's responses can be diagramed with a state machine, figure 3.1.

Performance-wise, one of the most common scenarios to be affected is retrieving a groundstation command from the comm controller. Since the main comm loop needs to detect that an I^2C command has been recieved and process it, responses to the IPC_COMM_LOAD_* commands are not immediately available for reading. The C&DH controller is forced to busy-wait and poll the comm controller for the correct command byte.

The default message for the comm controller requires more time to transmit now, as demonstrated in figure 3.2. As shown by the markers, there is a significant pause between bytes, caused by the comm controller using clock stretching while it prepares the next byte. This is due to the comm controller calculating the CRC for each byte in the ISR. One possible fix is to employ a CRC lookup table and fix the range of the comm status byte.

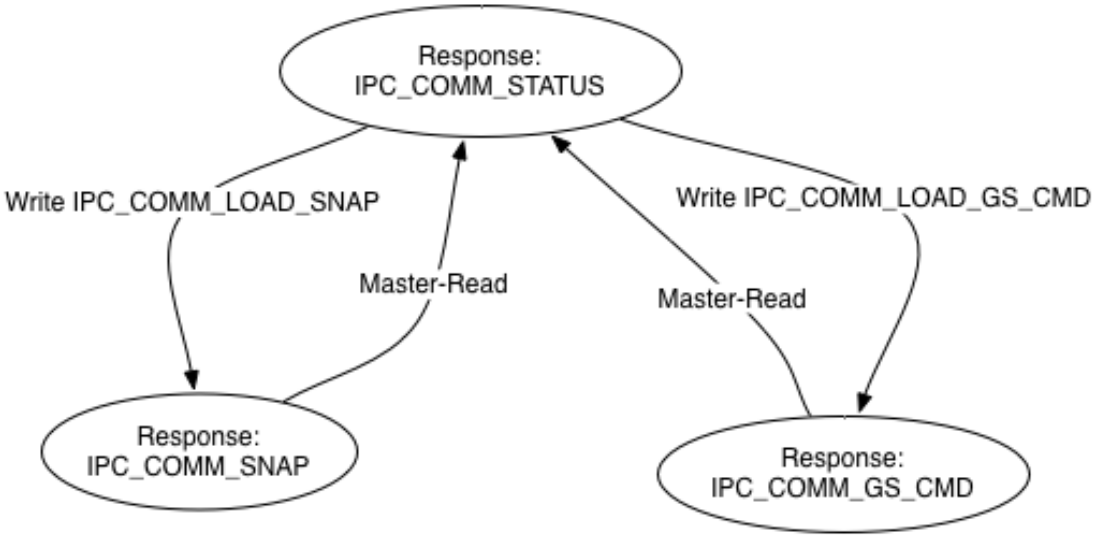

Figure 3.1: Comm response state

On a better note, there is now a single "funnel" for I^2C commands on the comm controllers. Also, with support for variable-length Master-Transmit transactions, the number of bus commands has been greatly reduced. With fixed-size Master-Transmit transactions, the bus was forced to have a separate I^2C command to go with each AX.25 Packet type the C&DH wanted to send to earth. Now, there is a single "Transmit this data" (IPC_COMM_TX_DATA) I^2C command that wraps the packet type and any associated data.

In order to prevent race conditions, the comm I^2C ISR was modified slightly. It now interrupts on start and stop bits, in addition to all of the previous states. This allows the comm controller to definitively know when it is safe to move data in and out of the I^2C buffers.

3.3 How To Add Commands

3.3.1 IPC Commands & TNC Packet Types

While Jacob Farkas' description of how to add a groundstation command[3] is still mostly correct, the steps to add an IPC command have changed signficantly. As an example, let's add the necessary commands to allow the C&DH processor to retrieve the I^2C statistics from the comm controller. Additionally, let's have the C&DH return the statistics in response to an uplinked command.

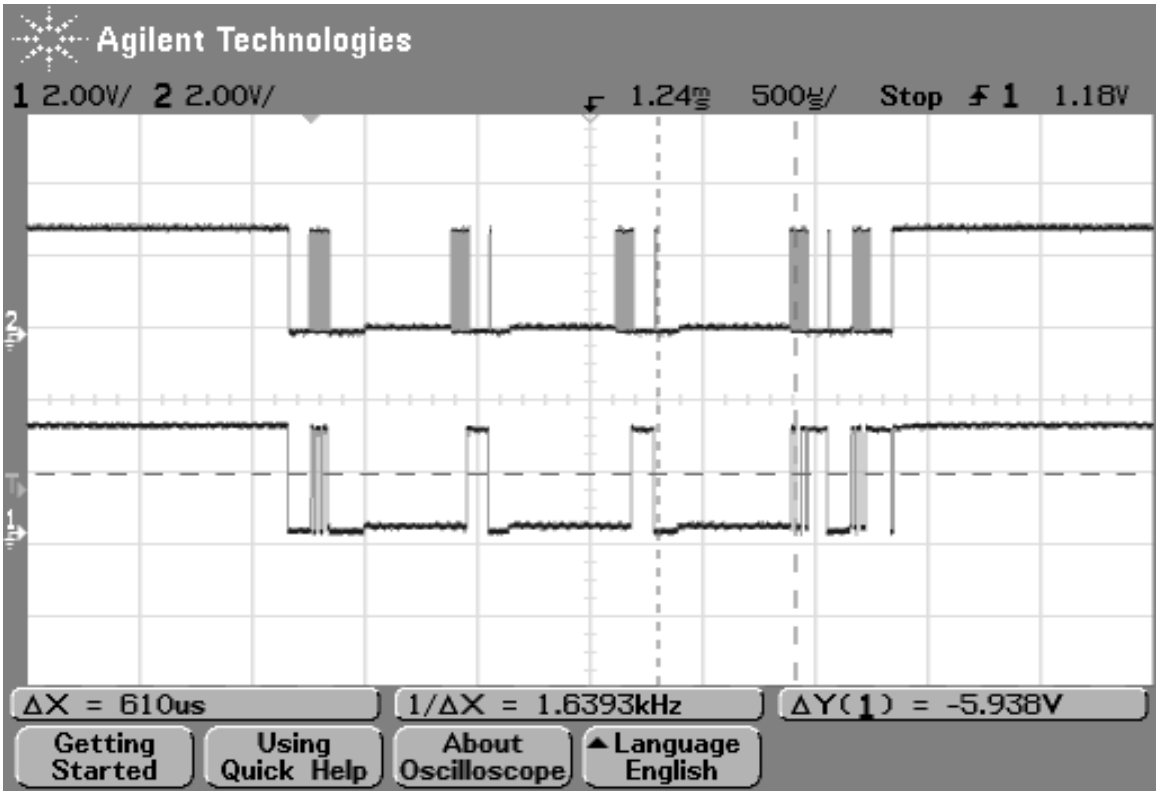

Figure 3.2: Comm Default Response on the Oscilloscope

IPC Command definitions are stored in cp2-i2c.h. Since this is going to be a bidirectional transaction, we'll need two commands. Using the supplied naming conventions, we'll use the names IPC_COMM_LOAD_I2C_STATS and IPC_COMM_I2C_STATS.

Listing 3.1: New command definitions

```
68 #define IPC_COMM_TX_DATA                0x10
69 #define IPC_COMM_TX_BEACON              0x11 // Transmits data, + CW
70
71 #define IPC_COMM_LOAD_I2C_STATS         0x12
72 #define IPC_COMM_I2C_STATS              0x13
```

The pair of commands we're adding fit the "load/retrieve" paradigm, in which the master sends an IPC_COMM_LOAD_XXXX command, pauses, and then attempts to read the resulting IPC_COMM_XXXX command. Since this is a common pattern, we can use the generic getValueFromComm() function, which implements this pattern. Because getValueFromComm() is the "raw" interface to comm controller properties, it would be best to wrap it in an identifiable name, such as getI2CStatsFromComm. The code for this is in listing 3.2

Listing 3.2: New command definitions

```
88  char getValueFromComm(unsigned char cmdArg, unsigned char expectedCmd,
89                        unsigned char *buf)
90  {
91      uint8 cmd = 0, dataLen = 0, retries;
92      commWait();
93
94      if ((err = writeToCPU(PIC_COMM, cmdArg, msgBuf, 0)) != ERR_NONE) {
95          return err;
96      }
97
98      // XXX: Do we need a delay here?
99      for (retries = 0; retries < COMM_GET_RETRIES; retries++) {
100         Delay1KTCYx(20);
101
102         if ((err = readFromCPU(PIC_COMM, &cmd, msgBuf, &dataLen, 1)) !=
103                 ERR_NONE) {
104             return err;
105         }
106
107         if (cmd == expectedCmd) {
108             break;
109         }
110     }
111
112     if (retries == COMM_GET_RETRIES) {
113         return ERR_I2C_BAD_MSG;
114     }
115
116     memcpy(buf, (void *)msgBuf, dataLen);
117     return ERR_NONE;
118 }
119
120 // ... snip for brevity ...
```

```
121
122 char getI2CStatsFromComm(unsigned char *data)
123 {
124     return getValueFromComm(IPC_COMM_LOAD_I2C_STATS, IPC_COMM_I2C_STATS, data);
125 }
```

We need to add support for the necessary commands on the comm processor. The I^2C command funnel is in the function handleMessage() in comm-main.c. Add a case to the switch statement for the command IPC_COMM_LOAD_I2C_STATS.

Listing 3.3: Adding command to the comm funnel

```
388 void handleMessage() {
389     static uint8 cmd, dataLen;
390     if ((err = readI2C(&cmd, msgBuf, &dataLen)) != ERR_NONE) {
391         return;
392     }
393
394     switch (cmd) {
395     case IPC_COMM_LOAD_GS_CMD:
396         COMM_INT = 0;
397         writeI2C(IPC_COMM_GS_CMD, &uplinkCommand.command,
398                                         uplinkCommand.len + 1);
399         break;
400
401     case IPC_COMM_LOAD_SNAP:
402         writeI2C(IPC_COMM_SNAP, (void *)&commSensorSnapData,
403                     sizeof(commSensorSnapData));
404         break;
405
406     case IPC_COMM_ACK_CMD:
407         tncTxPacket(PACKET_CMD_ACK, 0, 0);
408         break;
409
410     case IPC_COMM_NACK_CMD:
411         tncTxPacket(PACKET_CMD_NACK, 0, 0);
412         break;
413
414     case IPC_COMM_SET_TX_POWER:
415         cc1000WriteRegister(CC1000_PA_POW, msgBuf[0]);
416         break;
417
418     case IPC_COMM_TX_BEACON:
419         cwTransmitBeacon();
420         tncTxPacket(PACKET_COMM_ID, 0, 0);
421         tncTxPacket(msgBuf[0], msgBuf + 1, dataLen);
422         break;
423
424     case IPC_COMM_TX_DATA:
425         tncTxPacket(msgBuf[0], msgBuf + 1, dataLen);
426         break;
427
428     case IPC_COMM_LOAD_I2C_STATS:
429         ((uint16 *)msgBuf)[0] = accepted;
430         ((uint16 *)msgBuf)[1] = rejected;
431         ((uint16 *)msgBuf)[2] = totalRead;
432         ((uint16 *)msgBuf)[3] = totalSent;
```

```
433         writeI2C(IPC_COMM_I2C_STATS, msgBuf, sizeof(uint16) * 4);
434         break;
435
436     default:
437         break;
438     }
439 }
```

Now that the comm controllers will respond with the correct data, let's add the commands necessary to transmit the data to the ground. All packets transmitted from the satellite have a packet type; this is a part of the comm controller's TNC, and is described in depth in Chris Noe's paper on the comm subsystem[2]. So, we want to add a new packet type for I^2C statistics. These are located in comm-tnc, and the new command has been added in listing 3.4.

Listing 3.4: Adding a new packet type.

```
49 #define PACKET_COMM_ID       0x0B
50 #define PACKET_ADCS_DUMP     0x0C
51
52 #define PACKET_I2C_STATS     0x0D
```

Since the comm now supports the correct packet type, all that's left is to tie everything together. After following Jacob Farkas' instructions, you should have added a case to the switch statement in cdh-exec_cmds.c, similar to the one below. To respond to the command, we simply (ab)use the commandData buffer to retrieve the I^2C statistics from the currently active comm, fill it with the C&DH I^2C statistics as well, and then use commTransmit, which passes the message over I^2C and into the PACKET_* funnel (which is the code in listing 3.3.

Listing 3.5: Adding the handler for the Groundstation command.

```
1531         // --------------------
1532         // Return the I2C Stats
1533         // --------------------
1534         case CMD_CDH_GET_I2C_STATS:
1535             if (cdhCurrentState == NORMAL_OPS) {
1536                 if ((err = getI2CStatsFromComm(commandData)) == ERR_NONE) {
1537
1538                     ((uint16 *)commandData)[4] = accepted;
1539                     ((uint16 *)commandData)[5] = rejected;
1540                     ((uint16 *)commandData)[6] = totalRead;
1541                     ((uint16 *)commandData)[7] = resent;
1542                     ((uint16 *)commandData)[8] = totalSent;
1543                     ((uint16 *)commandData)[9] = hwFailures;
1544
1545                     commandAck();
1546                     commTransmit(PACKET_I2C_STATS, commandData,
1547                                  sizeof(uint16) * 10, 0);
1548                 }
1549                 else {
1550                     commandNack(DEFAULT_NACK_ERROR_CODE);
1551                     logError(err);
```

```
1552                    }
1553                }
1554                else {
1555                    commandNack(DEFAULT_NACK_ERROR_CODE);
1556                }
1557                break;
```

Chapter 4

Future Work

4.1 On Orbit Characterization

Since our satellite currently does not have the capability to upload code, testing the changes described in this document are currently pending another launch. The next launch carrying a CP satellite is currently scheduled for Summer '08; it is up to the operations team for that satellite to gather the data on how the bus is performing in its target environment.

4.2 Uploadable Code

Although the CubeSat community continues to move towards having at least one launch per year, putting a CubeSat in orbit is still an expensive and time-consuming endeavour. In order to get the most out of the hardware on orbit, being able to modify at least a portion of the runnable code on the satellite would increase the usefulness of the bus.

4.3 Arbitrary Interface for Modifying Parameters

Modifying a parameter of the satellite, such as a rate for taking data, is a very common operation. As it stands now, we have separate commands for each modifiable parameter, although they share a lot of common code. By specifying a generic interface, it would be possible to cut down on a lot of "boilerplate" code and reduce maintenance costs of the embedded software. One possibility is to use raw memory locations, and write tools that determine a symbol's location in memory using the linker output. Alternatively, a static structure could be defined that specifies which parameters are modifiable.

Appendix A

Acknowledgements

- Jacob Farkas, Chris Noe, and Kyle Leveque for getting me involved in this whole mess.
- The entire MSTL Crew, past and present, for making the lab a great place to work.
- My family, for their unending support.
- Dr. Jordi Puig-Suari, for going crazy each and every day.
- Dr. John Bellardo, for his guidance and understanding.

Appendix B

Source Code Excerpts

All code for this project can be pulled out of the PolySat svn from /Home/kmccabe/i2c-branch/ @ rev 8491.

B.1 transferI2C()

Listing B.1: Old TransferI2C Function frame

```
1  /*
2   * FUNCTION:
3   *  transferI2C
4   *
5   * DESCRIPTION:
6   * transferI2C() implements our generic I2C communication protocol
7   * it consists of 2 steps
8   * 1: the master writes a single byte command to the slave, along with any
9   *         data associated with that command (up to 256 bytes, currently).
10  *    the data is stored in buf, and is txlength long
11  * 2: the master reads from the slave. first byte read is the length of the
12  *    complete i2c transaction, followed by 1 or more bytes of data
13  *
14  * comment: jfarkas and cnoe discussion leads us to the decision to implement
15  * transferI2C separately from read/writeToSlave. the reason is that the cmd
16  * byte must be the first byte of the write transaction to the slave, followed
17  * immediately by the transaction data (if any). writeToSlave doesn't allow for
18  * this without two separate transactions, which won't work with our IPC.
19  *
20  * PARAMETERS:
21  *  unsigned char addr
22  *      Device address to write to
23  *
24  *  unsigned char command
25  *      Device command. Determines what is read/written
26  *
27  * RETURNS:
28  *  0 on success
29  *  I2C error code otherwise
```

```
30   */
31  int transferI2C(unsigned char addr, unsigned char command)
32  {
33      unsigned char txLength, rxLength;
34      unsigned char i;
35      unsigned char *destBuffer;
36
37      // clear RX buffer
38      memset(i2cRxBuffer, 0x00, IPC_BUF_MAX);
39
40      if (command & 0x80) {
41          txLength = payloadCommandTable[command & 0x7F][0];
42      } else {
43          txLength = commCommandTable[command][0];
44      }
45
46      // first, write the command + command data to the slave
47      if ((err = CP2_StartI2C()) < 0) {
48          return err;
49      }
50
51      if ((err = CP2_WriteI2C(addr | I2C_WRITE)) < 0) {
52          CP2_StopI2C();
53          return err;
54      }
55
56      if ((err = CP2_CheckAckI2C()) < 0) {
57          CP2_StopI2C();
58          return err;
59      }
60
61      if ((err = CP2_WriteI2C(command)) < 0) {      // command byte
62          CP2_StopI2C();
63          return err;
64      }
65
66      if ((err = CP2_CheckAckI2C()) < 0) {
67          CP2_StopI2C();
68          return err;
69      }
70
71      for (i = 0; i < txLength; i++) {
72          if ((err = CP2_WriteI2C(i2cTxBuffer[i])) < 0) {
73              CP2_StopI2C();
74              return err;
75          }
76  //      if(i=CP2_CheckAckI2C())
77  //          return i;
78      }
79      CP2_StopI2C();
80
81      // read back
82      if ((err = CP2_StartI2C()) < 0) {
83          CP2_StopI2C();
84          return err;
85      }
86
87      if ((err = CP2_WriteI2C(addr | I2C_READ)) < 0) {
```

```
88          CP2_StopI2C();
89          return err;
90      }
91
92      if ((err = CP2_CheckAckI2C()) < 0) {
93          CP2_StopI2C();
94          return err;
95      }
96
97      /* XXX: what happens if we issue a stop condition while the slave is still
98       * transferring? Will the slave pick up on the stop condition and stop, or
99       * will it keep trying to send data?
100      */
101     if (!(err = CP2_ReadI2C(&rxLength))) {
102         if (rxLength > 0 && rxLength <= IPC_BUF_MAX) {
103             if ((err = CP2_AckI2C()) < 0) {
104                 CP2_StopI2C();
105                 return err;
106             }
107
108             for (i = 0; i < rxLength; i++) {
109                 if ((err = CP2_ReadI2C(i2cRxBuffer + i)) < 0) {
110                     CP2_StopI2C();
111                     return err;
112                 }
113
114                 if (i != rxLength - 1) {
115                     if ((err = CP2_AckI2C()) < 0) {
116                         CP2_StopI2C();
117                         return err;
118                     }
119                 }
120             }
121         }
122     } else {
123         CP2_StopI2C();
124         return err;
125     }
126
127     CP2_NoAckI2C();
128     CP2_StopI2C();
129
130     return ERR_NONE;
131 }
```

B.2 Comm I^{2}C ISR

Listing B.2: Old Comm I^{2}C ISR frame

```
1  /**
2   * ISR for I2C address match.
3   */
4  #pragma interrupt i2cISR
5  void i2cISR(void)
6  {
7      static unsigned int idlectr;            // I2C retry counter
8
```

```
9       static unsigned char i2cBufferIndex;    // Index into current I2C buffer
10      static unsigned char i2cCommand;        // Last I2C command recv'd
11      static unsigned char commandReceived;   // Have we received a command?
12
13      extern unsigned char commSensorSnapData[];
14
15      unsigned char i;
16      unsigned char data;
17
18      // Record that we've received an i2c request
19      i2cActivityDetect = TRUE;
20
21      // Examine S, RW, DA and BF to determine I2C state
22      switch (SSPSTAT & 0x2D) {
23          // ---------------------------------------------
24          // State 1: Master Write, previous byte was address
25          // ---------------------------------------------
26          // S = 1, RW = 0, DA = 0, BF = 1
27          case I2C_STATE1:
28              i2cBufferIndex = 0;     // Reset buffer index
29              i2cCommand = 0;         // Reset last command
30              commandReceived = FALSE;    // Reset cmd recvd
31              i = 0;                  // Reset loop iterator
32              data = SSPBUF;          // Dummy read SSPBUF to clear BF
33              break;
34
35          // ---------------------------------------------
36          // State 2: Master Write, previous byte was data
37          // S = 1, RW = 0, DA = 1, BF = 1
38          // ---------------------------------------------
39          case I2C_STATE2:
40              // Store command byte separately from data
41              if (commandReceived) {
42                  i2cRxBuffer[i2cBufferIndex++] = SSPBUF;
43              } else {
44                  commandReceived = TRUE; // Command received
45                  i2cCommand = SSPBUF;    // Store command
46              }
47              break;
48
49          // ---------------------------------------------
50          // State 3: Master Read, previous byte was address
51          // S = 1, RW = 1, DA = 0, BF = 0
52          // ---------------------------------------------
53          // Description: Return length of command data
54          // ---------------------------------------------
55          case I2C_STATE3:
56              switch (i2cCommand) {
57                  case IPC_COMM_STATUS:
58                      data = IPC_COMM_STATUS_RX_LENGTH;
59                      break;
60
61                  case IPC_COMM_SENSOR_SNAP:
62                      i2cBufferIndex = 0;
63                      data = IPC_COMM_SENSOR_SNAP_RX_LENGTH;
64                      break;
65
66                  case IPC_COMM_TX_DATA:
```

```
67                      data = IPC_COMM_TX_DATA_RX_LENGTH;
68                      break;
69
70                  case IPC_COMM_TX_BEACON:
71                      data = IPC_COMM_TX_BEACON_RX_LENGTH;
72                      break;
73
74                  case IPC_COMM_TX_PAYLOAD_DATA:
75                      data = IPC_COMM_TX_PAYLOAD_DATA_RX_LENGTH;
76                      break;
77
78                  case IPC_COMM_TX_PAYLOAD_TEST:
79                      data = IPC_COMM_TX_PAYLOAD_TEST_RX_LENGTH;
80                      break;
81
82                  case IPC_COMM_TX_RTC_TIME:
83                      data = IPC_COMM_TX_RTC_TIME_RX_LENGTH;
84                      break;
85
86                  case IPC_COMM_GET_COMMAND:
87                      i2cBufferIndex = 0;
88                      data = 1 + uplinkCommand.len; // command + command data bytes
89                      break;
90
91                  case IPC_COMM_ACK_COMMAND:
92                      data = IPC_COMM_ACK_COMMAND_RX_LENGTH;
93                      break;
94
95                  case IPC_COMM_NACK_COMMAND:
96                      data = IPC_COMM_NACK_COMMAND_RX_LENGTH;
97                      break;
98
99                  case IPC_COMM_GET_TNC_MODE:
100                     data = IPC_COMM_GET_TNC_MODE_RX_LENGTH;
101                     break;
102
103                 case IPC_COMM_TX_ADCS_DUMP:
104                     data = IPC_COMM_TX_ADCS_DUMP_RX_LENGTH;
105                     break;
106
107                 case IPC_COMM_SET_TX_POWER:
108                     data = IPC_COMM_SET_TX_POWER_RX_LENGTH;
109                     break;
110
111
112                 default:
113                     data = 0;
114                     break;
115             }
116
117             // wait for bufferfull to clear (= read complete)
118             for (idlectr = 0; idlectr < READ_RETRY; idlectr++) {
119                 if (!SSPSTATbits.BF)
120                             break;
121                 // XXX: what to do if buffer full hasn't cleared?
122                 // only happens if master doesn't read
123             }
124
```

```
125             // we timed out waiting for bufferfull
126             if (idlectr == READ_RETRY) {
127                 // XXX: reset state? pins?
128                 // log error
129             }
130 
131             SSPBUF = data;      // Buffer next byte
132             break;
133 
134         // ------------------------------------------------
135         // State 4: Master Read, previous byte was data
136         // S = 1, RW = 1, DA = 1, BF = 0
137         // ------------------------------------------------
138         case I2C_STATE4:
139             // Update current pin values
140             commStatus &= 0b00001111;     // clear upper 4 bits
141             commStatus |= (SEL_TX_READ << 4);
142             commStatus |= (SEL_RX_READ << 5);
143             commStatus |= (SEL_RF_READ << 6);
144             commStatus |= (EN_PL_READ << 7);
145 
146             // Return command data, if any
147             // all transmit commands set "not ready" until we transmit data currently in buffer
148             switch (i2cCommand) {
149                 case IPC_COMM_TX_BEACON:
150                     commTxFlags |= BEACON_WAITING;
151                     data = commStatus;
152                     commStatus &= ~COMM_STAT_READY;
153                     break;
154 
155                 case IPC_COMM_TX_DATA:
156                     commTxFlags |= CDH_DATA_WAITING;
157                     data = commStatus;
158                     commStatus &= ~COMM_STAT_READY;
159                     break;
160 
161                 case IPC_COMM_TX_ADCS_DUMP:
162                     commTxFlags |= ADCS_DUMP_WAITING;
163                     data = commStatus;
164                     commStatus &= ~COMM_STAT_READY;
165                     break;
166 
167                 case IPC_COMM_TX_PAYLOAD_TEST:
168                     commTxFlags |= PAYLOAD_TEST_WAITING;
169                     data = commStatus;
170                     commStatus &= ~COMM_STAT_READY;
171                     break;
172 /*
173                 case IPC_COMM_TX_PAYLOAD_DATA:
174                     commTxFlags |= PAYLOAD_DATA_WAITING;
175                     data = commStatus;
176                     commStatus &= ~COMM_STAT_READY;
177                     break;
178 */
179                 case IPC_COMM_TX_RTC_TIME:
180                     commTxFlags |= RTC_TIME_WAITING;
181                     data = commStatus;
182                     commStatus &= ~COMM_STAT_READY;
```

```
183                     break;
184
185                 case IPC_COMM_SET_TX_POWER:
186                     cc1000TxPower = i2cRxBuffer[0];
187                     commFlags |= UPDATE_TX_POWER;
188                     data = commStatus;
189                     break;
190
191                 case IPC_COMM_ACK_COMMAND:
192                 case IPC_COMM_NACK_COMMAND:
193                     if (i2cCommand == IPC_COMM_ACK_COMMAND) {
194                         commTxFlags |= ACK_WAITING;
195                     } else {
196                         commTxFlags |= NACK_WAITING;
197                     }
198
199                     commStatus &= ~COMM_STAT_CMD_RECVD; // clear command received status
200                     COMM_INT = 0;                       // cdh has responded to our COMM_INT
201                     data = commStatus;
202                     commStatus &= ~COMM_STAT_READY;
203                     break;
204
205                 case IPC_COMM_STATUS:
206                     data = commStatus;
207                     break;
208
209                 case IPC_COMM_SENSOR_SNAP:
210                     data = commSensorSnapData[i2cBufferIndex];
211                     i2cBufferIndex++;
212                     break;
213
214                 case IPC_COMM_GET_COMMAND:
215                     if (i2cBufferIndex == 0) {
216                         data = uplinkCommand.command;
217                     } else {
218                         data = uplinkCommand.data[i2cBufferIndex - 1];
219                     }
220
221                     i2cBufferIndex++;
222                     break;
223
224
225                 default:
226                     i = 0;
227                     break;
228             }
229
230             // wait for bufferfull to clear (= read complete)
231                 for (idlectr = 0; idlectr < READ_RETRY; idlectr++) {
232                     if (!SSPSTATbits.BF) {
233                         break;
234                     }
235                     // XXX: what to do if buffer full hasn't cleared
236                     // only happens if master doesn't read
237                 }
238
239                 // we timed out waiting for bufferfull
240                 if (idlectr == READ_RETRY) {
```

```
241                     // XXX: reset state? pins?
242                     // log error
243                 }
244
245             SSPBUF = data;      // Buffer next byte
246             break;
247
248         // ------------------------------------------------
249         // State 5: Master NACK
250         // S = 1, RW = 0, DA = 1, BF = 0
251         // ------------------------------------------------
252         case I2C_STATE5:
253             // Post command cleanup
254             i2cCommand = 0;         // Reset command
255             commandReceived = 0;    // Reset cmd recvd
256             i2cBufferIndex = 0;     // Reset index
257             i = 0;                  // Reset loop iterator
258             break;
259     }
260
261
262     // Release SCL to free the bus
263     SSPCON1bits.CKP = 1;
264
265     // Clear interrupt flag
266     PIR1bits.SSPIF = 0;
267 }
268
269 #pragma code
```

Bibliography

[1] Cubesat Specification. Available at http://cubesat.atl.calpoly.edu/media/CDS_rev10.pdf.

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

[3] Farkas, Jacob. "CPX: Design of a Standard Cubesat Sotware Bus" Cal Poly Senior Project.

[4] A. Leon-Garcia and I. Widjaja, "Communication Networks: Fundamental Concepts and Key Architectures" New York: McGraw-Hill, 2004.

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