Distributed Microcontroller Network and Automated Test Harness for the Alaska Research Cubesat

By Gregg Christopher

Graduate Advisory Committee:

Chair:	Dr. Orion Lawlor, Computer Science
Members:	Dr. Chris Hartman, Computer Science
	Dr. Denise Thorsen, Electrical Engineering

M.S. PROJECT REPORT

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Alaska Fairbanks, 2011.

Table of Contents:

Abstract

This project presents the concept, architecture, requirements, design and initial implementation of a distributed embedded network intended for development of a satellite operating system. It includes an analysis of errors expected in the space environment modes of error simulation and methods of recovery. A first proof-of-concept implementation of the development environment and test harness are designed, fabricated and programmed. Sufficient command and data handling functionality is demonstrated to validate using software to induce simulated Single Event Upset (SEU) errors, detect error conditions, and recover via a reset mechanism.

Project Background

Cubesat Picosatellites

The Cubesat standard was initially established by Cal Poly San Louis Obispo as a common mechanical and functional specification that would allow a common launch device to deploy groups of satellites as ancillary payloads to large primary launches. One to three cubical satellites are built with sliding rails fitting this spring-loaded “pod.” As of this writing, Spring 2010, 35 cubesats have been launched, several others were lost during a failed launch and the pace of development and launches is growing rapidly. Of special interest to Alaska students is the capability of the Kodiak space port to place cubesats in high-inclination orbits visible to Alaska-based ground stations.

Cubesats are 10 cm on a side and must weight less than 1 kg at launch. They are subject to a variety of safety testing that varies by launch provider. Cubesats must follow appropriate regulations for radio communications.

Since the satellites are small, need not be high-performance, are relatively inexpensive ($40,000 for development and $40,000-$80,000 for launch are current typical costs) and operate in Low Earth Orbit (LEO) where they are shielded from the most severe radiation, cubesats can be a good student project for university aerospace programs.

The Cal Poly cubesat program acts as an intermediary to launch providers by defining standards for satellites and developing cubesat deployment hardware that is pre-qualified for a variety of launch vehicles and providers.[PPOD] This allows cubesat builders to bypass the complexities of dealing with the various flight validation requirements of many different potential launch providers by meeting a common standard published by Cal Poly. However, this may mean that in practice, a cubesat developer must be aware and meet both Cal Poly and launch provider requirements if there are other connections such a research grant constraints.

Since the first expected launch opportunity for the Alaska Research Cubesat is through a grant provided by NASA, the UAF program must also pass design review and flight validation testing defined by NASA, above and beyond the baseline cubesat requirements. These requirements also drive UAF cubesat design and implementation processes.

Space Systems Engineering Program

The Alaska Space Grant supports a Space Systems Engineering group with dedicated lab space and other resources. This interdisciplinary group is currently pursuing its first satellite launch, the Alaska Research Cubesat (ARC). The effort so far has included a graduate course, development of extensive analysis, requirements and design documents and two formal review proceedings before NASA staff.

The full set of documents for the ARC are not publicly available. Presented in this paper is an extremely abbreviated and generalized summary of those requirements.

One goal of this summary is to capture the critical concepts in a very nonspecific form, without including any details that could potentially subject the authors to liability without seeking State Department approval for this paper. Specifics of orbital mechanics, some details of the expected space environment, and other technical details are unspecified, but may motivate some of the project decision and discussion below.

Another goal is to avoid the encumbrance of documentation that would meet more formal NASA requirements, such as dozens of pages of requirements traceability matrices, large tables of acronyms and references to specific NASA design principle documentation. An attempt is made here to present the end result of the two-year research and design process in as concise and understandable a format as possible.

Overview of ARC Software Needs

Although the primary goal of the ARC project is to launch a functioning satellite, the flight firmware is not the only software development aspect. Satellite work is part of a coordinated effort to create a satellite development and testing lab, space flight hardware and operational ground station for satellite communication. Where the needs of these areas overlap, software effort should be reused to improve integration and avoid duplicate effort.

Flight Software

The flight software consists of the collected code present on all microcontroller, FPGA and other programmable elements that will be launched on the cubesat itself. A rudimentary operating system will handle communication between subsystems, command and data interfaces, task scheduling and prioritization. Subsystems require software for specific tasks such as radio communication, attitude determination and control and

Space systems have unique reliability requirements. Designing for reliability is critical and special types of system failure must be considered. Physical access to a satellite is impossible after launch, or even after delivery to a launch provider. It cannot be reset with certainty, faulty connections can not be unplugged and reconnected, and aside from also-experimental radio communication system, nothing about the satellite can even be observed directly after deployment. Comprehensive statistics have not been obtained, but between 25% - 50% of successfully launched cubesats have experienced complete failure attributed to the broad category of Command and Data Handling (CDH) error that could be attributed to faulty flight software.

Another key to this process is the realization that reliability engineering can be an endless task and specific risks and failure modes must be prioritized to avoid a bottomless pit of software effort. The ability to reprogram the system in flight is desired, but relegated to a wish list. This will certainly be present on future student or professional cubesats.

Ground Station and Mission Support

The ground station must perform satellite tracking and control of the antenna rotors to establish and maintain uplink and downlink capability. It must also perform proper encoding and decoding of radio signals into data packets, handle packet loss and capture and store received data. There must be a command and control interface allowing mission control to manage the satellite.

Received data is the product of the mission and must be made available for transfer. The ground station system may also be required to perform data interpretation and in order to sensibly automate command and control decisions that are dependent on the content of the data.

Validation testing of the flight software will require at least a representative implementation of the ground station in order to perform end-to-end testing. The ground station itself is subject to reliability requirements, as ground station errors can directly impact the satellite.

There are many Commercial Off-The-Shelf (COTS) choices for ground station subcomponents, including an international effort to create a federated network of ground stations called GENSO[GENSO], led by the European Space Agency and NASA. UAF has been granted an excellent ground station site by NOAA and hopes to make its ground station available for downlink by other cubesat efforts. In particular, a University of Michigan cubesat launched in summer 2010 from the Kodiak private space port.

Development Environment and Testing

Software for development includes collaboration tools, revision control and the toolsets required for specific hardware and programming languages. These are off-the-shelf, but require configuration, training and maintenance to be effective.

In order to perform validation testing, software may be required to automate and log test results. The project will be more adaptable to requirement and design changes as well as last-minute revisions if validation and regression testing are as streamlined as possible. Data logging, warehousing and analysis tools should be considered.

Distributed Hardware Architecture

Distributed System Design

The cubesat consists of discrete subsystems for each of the major system functions. These subsystems communicate via a common interface for command and data communication as well as system maintenance tasks.

In order to reduce overall complexity, each subsystem is required to be as identical as possible. The same microcontroller is used on all subsystems, all have the same electrical connections to the system bus, and the same internal resources are reserved for system tasks. This allows significant code re-use across subsystems and the development of a common codebase for deployment to all subsystems.

Subsystem Interface

Communication between subsystems prefers well-established serial protocols that have onboard hardware peripheral support. However, lines are reserved to implement custom data protocols, interrupts or a parallel channel. A switched bus also allows access to the hardware programming interface to permit supervisory circuits or in-situ reprogramming with or without a software bootloader.

This interface is physically implemented via stackthrough headers chosen for compatibility with the COTS power system purchased for flight, the Clyde Space power board. Pin assignments are selected to avoid interference with other commercial offerings (such as Pumpkin Space products), in case this or other products desire to mix and match those systems with ARC designs.

The physical arrangement of the pins on the header is located in Appendix 1: Physical Layout of Stackthrough Header Pins.

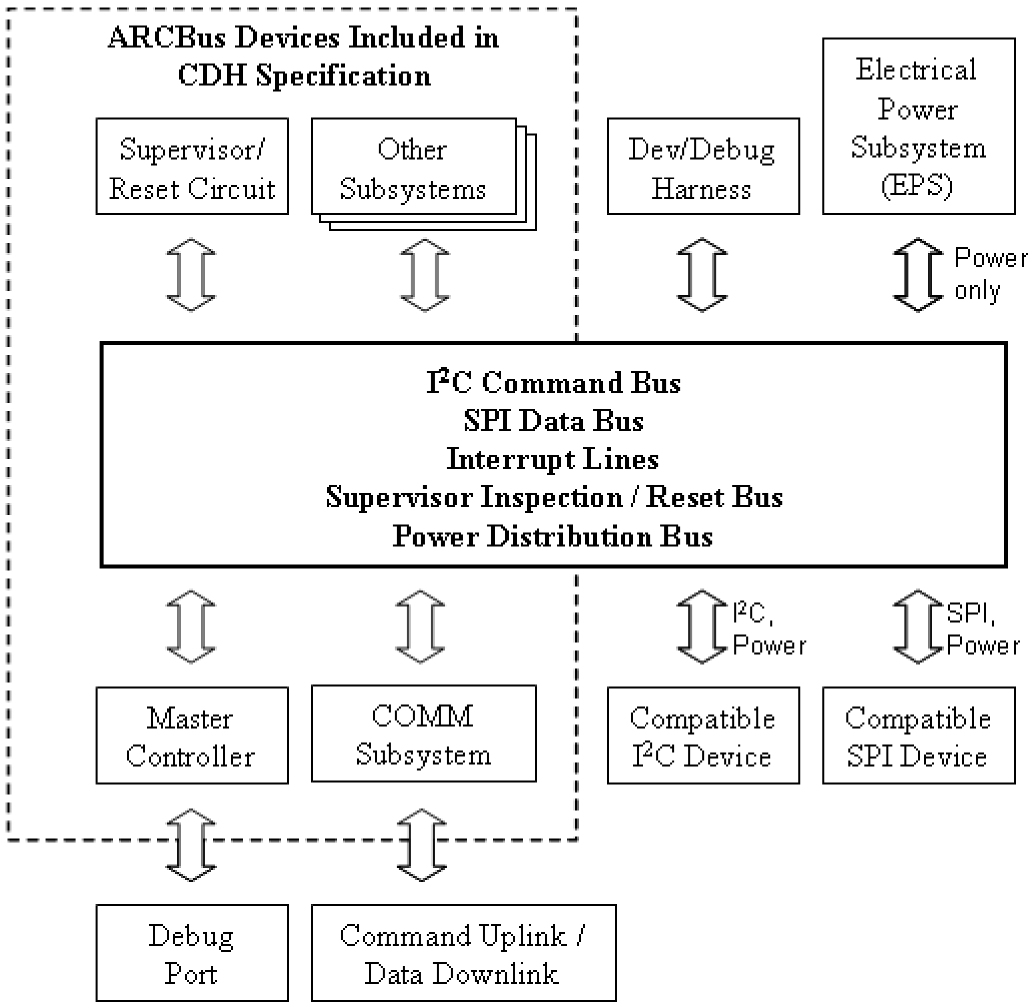
ARCBus Specification

Logical Signal Types

The ARCBus is a collection of data signals grouped into logical functional groups:

- I2C (2 lines): Primarily to exchange commands and responses between components.
- SPI (3 lines): Primarily to exchange blocks of data.
- Programming/Debugging/Reset signals (9 lines): These pins are normally electrically disconnected, but when the enable line for that component is asserted, they allow access to programming, power switch and status reporting functionality.
- Enable Switch Pins (8 lines): Each line is dedicated to enabling the programming pins on one component.
- Power: 5V , 3.3V, Battery Voltage and Ground

Bus Architectural Overview



ARCBus Devices:

ARC Bus Device: Any device that fully implements the ARCBus standard and participates fully in the system. Most ARC components are implemented as ARCBus devices that contain additional circuitry specific to that component.

Master Controller: An ARCBus device that is designated as having the primary task of coordination of command and data flow and maintenance of the system as a whole. This component contains the debug port which allows tethered communication with the satellite after final assembly, but before launch.

Peripheral Adapter: An ARCBus device that exposes the entire system as a peripheral to some external system. For example, the entire ARCBus can present itself as a USB peripheral to a more powerful and complicated mission-specific computer. The external interface to this peripheral may vary, as long as the adapter interfaces correctly with the bus.

Supervisor / Reset Circuit: This circuitry polls the system for error conditions and performs component resets or system resets

Other Items:

Dev/Debug Harness: This harness consists of cables that expose all signals of the ARCBus while allowing the satellite to be nearly completely assembled. Removal of this harness is one of the last steps to be performed after the satellite has passed all validation tests before the satellite is ready to be shipped.

ARCBus-Compatible I2C and SPI Devices: A non-ARCBus device may be attached to the appropriate signal lines, assuming that it follows certain operational protocols and passes additional tests beyond the requirements specified by the I2C and SPI protocols.

Power Supply: If it is not an ARCBus Device, the power supply must meet the standards to be ARCBus-compatible in order to be connected to the bus.

Expected Error Modes

Single Event Upsets

For the expected space environment and flight hardware, it is expected that ions or radiation will randomly induce electrical charges in the satellite hardware. While this type of event is not expected to damage hardware, it will cause bits in memory to change state randomly. The exact rate of error is extremely difficult to characterize, as it will vary depending on the manufacturing process of a particular device, the orbital details and space weather.

Based on reported conditions of similar satellites and predicted radiation energy levels, these upsets are expected to affect volatile memory only, not the flash memory used for program storage and long-term data storage. Further, it has been indicated that the charge pump circuitry that allows a flash memory write will fail at lower energy levels than those required to overwrite flash. However, since flash memory may be written to programmatically, attention must be paid to any mechanism that could malfunction (or spontaneously be created through stochastic memory changes) in a way to cause flash memory to be written.

For the purposes of this analysis, Single Event Upsets are assumed to be independent and uniformly random across affected memory, though this may not be true based on details of actual hardware used. Although the error mode is named the “Single” Event Upset and they are expected to occur sequentially as a poisson process, the analysis and mitigation of these events must assume that many could occur undetected before any observable change in state. A detected error state may include an arbitrary amount of corrupted memory and the possibility that some of it has been written to less volatile storage.

Latchups

A latchup event is one where a temporary electric structure is formed that causes a transistor to short to ground. This is typically caused by excess current and causes memory or a logical structure to be stuck in one state. There is no way to resolve this in software and the system must be powered down in order to clear the latchup. In some circumstances, a latchup can cause a risk of overheat from excessive current through an element. In the vacuum environment, cooling may be impaired, increasing the need to detect and clear this condition promptly.

Latchup events are much more difficult to emulate using the software-only techniques available for this project. A recovery method is provided by the power reset capability of the supervisory circuit, described below. Provisions for future work in testing and recovering from latchup events are included in the project implementation.

Traditional Software Error Modes

In addition to the special demands of the space environment, any software project is subject to the usual range of error modes. The design, development and testing processes must account for detection and recovery of errors induced by faulty programming or unexpected operational circumstances that induce soft errors in the satellite. This places additional emphasis on the need for the recovery mechanisms to be well thought out, simple, robust and thoroughly tested.

Error Recovery Model

The complete error recovery model for the satellite is based on first attempting soft recovery of known corrupt memory before performing a hard reset, and on first attempting to reset a single subsystem before resetting the entire satellite.

The desired order of recovery options is:

1. Per-subsystem soft error recovery. This would encompass scenarios where redundant memory storage or checksum-based approaches are sufficient to detect and recover from an error without any interaction with other subsystems. This might also include checking register states, such as those that control peripherals, against known correct values in non-volatile memory.
2. Guided recovery by external commands over the shared bus. This would require that all subsystems involved have working methods of communication, but would allow them to compare state information to attempt recovery.
3. Self-reset on command by bus master. Assuming that communication remains possible, an error has been detected, a subsystem can be ordered to perform a soft-reset. This is expected to resolve many non-latchup error modes, in cases where the reset capability has not been compromised.
4. Hard reset of subsystem(s) by the supervisor circuit. The supervisor circuit has a dedicated communication line to each subsystem. It can query each, and assuming that a sufficient majority analysis has been reached as to which subsystem is in error, that subsystem can be selectively reset.
5. Hard reset of the entire satellite by the supervisor circuit. This is expected to be highly likely to recover from a broad range of errors, as it allows power to be completely drained from all programmable elements. This reset is likely to be common in any case where communications have been interrupted,

since the subsystems would be unable to compare information that could lead to a diagnosis of a single malfunctioning subsystem. In particular, since the serial channels are shared, any subsystem with malfunctioning peripheral behavior is likely to induce this error recovery step.

Re-Scoping for MS Project

The discussion of goals and requirements above describes a system that may be infeasible to develop in the scope and context of a single Master's thesis. In a perfect world, this paper would present analysis of a finished and flight-ready set of software systems, but that goal proved unrealistic for the available time and resources. Additionally, the extent of Waterfall-style design work for the Cubesat has a tendency to encounter analysis paralysis that can be improved by smaller-scale, earlier development. With guidance from the graduate project committee, the scope of this project was restated to create an achievable first step that would also usefully contribute to the development of a satellite.

Revised Implementation Scope

With advisory guidance, the following goals were identified as most desirable while still being achievable within the scope of a Master's Thesis project. These were chosen to further the needs of the ARC project as a whole while still providing a defined "end point" to a project that could be arbitrarily large.

- The system should demonstrate the ability to exchange commands between subsystems, perform work based on those commands and respond with the result of the commands. Those commands should have clearly defined outcomes that can be tested for correctness. Each subsystem should be able to accept commands, report results, and be interrogated for status information by an external control and testing interface.
- Software simulation of error modes should be demonstrated on command, in such a way that errors are repeatable as representative of expected error modes as possible.
- The system should allow for manual and automated testing. This capability should primarily take the form of easily-modified PC software that can server both as the primary test control mechanism and the basis for a future test harness for more sophisticated testing and analysis.

An attempt was made to defer resolution of certain categories of work:

- Work on a complete command table able to handle all operations of the satellite, including mission phase management and specific subsystem operations. Though highly desirable, this is an unrealistic goal until a basic concept and framework have been proven to work.
- Complicated command interactions requiring queueing or prioritization logic. In the absence of subsystem prototypes, procedures of this type are likely to see many revisions up to and through the completion of this more limited project.
- Anything requiring more hardware development should be deferred. Due to unresolved limitations in the first iterations, some rapid-prototyping of hardware was required to reach functionality.

Revised Error Discussion

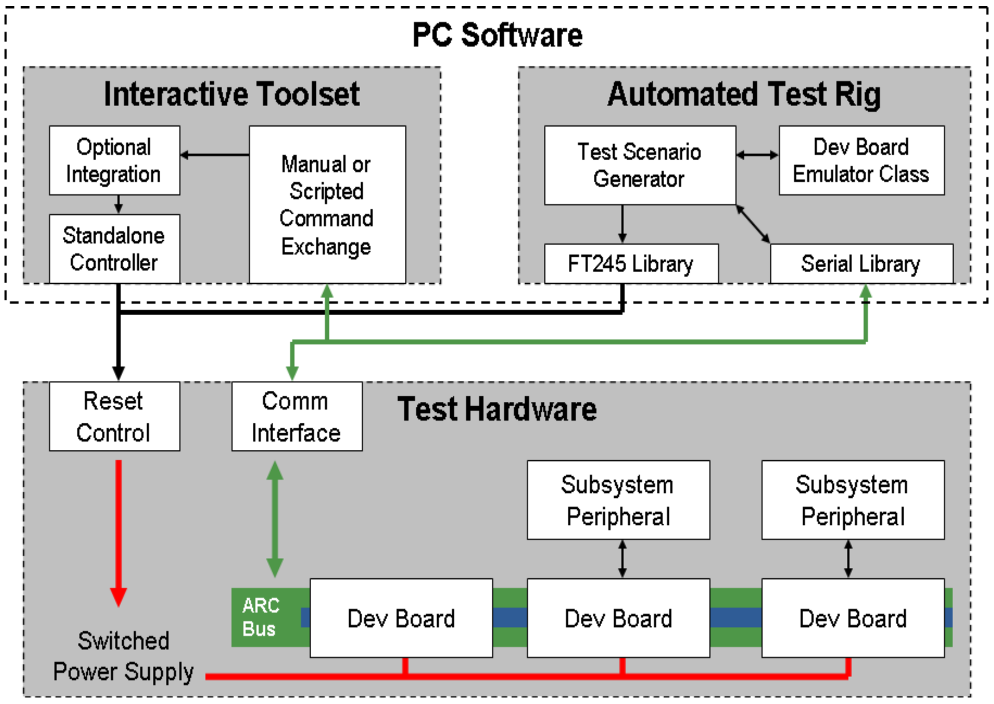
The revised scope is still adequate to test SEU and most firmware error modes, but does not have any direct provision for testing latchup. It is sufficient to test the complete sequence of planned error recovery methods with only software modifications. Other than those constraints, the revised scope does not impact the analysis of error causes, failure modes, or recovery methods.

Test Bed Design

Since the supervisor circuit has not been designed, the implementation of this project will serve as the first prototype of the physical and logical operations that system must perform. The final form of the test harness will be the three available dev boards connected to a prototype control circuit that will stand in as the supervisor circuit. That circuit will in turn be connected to a PC that will host software controlling the entire test harness, emulating each dev board sufficient to test it, and performing control logic that is not yet present on the supervisor circuit.

Project Implementation

The implementation of this project consists construction of physical hardware, firmware for programmable elements of the hardware, and PC software. This project involved requirements, design and review aspects before any code could be written. These were required by the Space Grant before hardware design could commence and were a non-negotiable blocking factor to further development. The background and design sections above are a condensed summary of subsections of the requirement and design specifications produced during this process. Analysis of risk and timelines were also required, but are omitted from this report.



The interactive portion consists of three programs:

- An off-the-shelf serial communication tool, realterm.
- A tool to control the switching hardware and power supply.

- A programming framework to perform ad-hoc serial communication.

The automated portion of the test harness is constructed using components from the switching control program and the serial communication libraries. Since the libraries used happen to compile cleanly when used together, they are included in a single project. Communication code is included that allows the project to be broken out into multiple executable objects with messages passed between them in case incompatible libraries are adopted or precompiled tools are added to the system.

Toolchain Cost Comparison:

Choosing a compiler and debugger combination for embedded development may not be trivial. At the outset of this project, despite exhaustive testing, no working combination of compiler, hardware programmer/debugger and drivers was found for the 64-bit Windows environment on development machines, so the entire toolchain was run in a virtualized environment. While eventually feasible, this introduced another layer of difficulties. Eventually TI released improved drivers and compatibility improved.

Four compiler/debugger suites were considered for this project, included based on the fundamental criterion that the experimenter was able to get each of them to function at all.

mspgcc: gcc toolchain for MSP430

+ Free
+ Unlimited code size
+ One team member experienced and familiar with usage
- Most of the team were unable to effectively use the gdb debugger
- Unique precompiler conventions. Example code is unlikely to work.

IAR Embedded Workbench

+ Extremely commonly distributed
+ Reasonably simple build and debug process
+ Plentiful code examples for beginners
- code size limit
- “Request Quote” price for version with sufficient code size

TI Code Composer Studio (CCStudio)

+ Developed by device manufacturer with direct support
+ Reasonably simple build and debug process
+ Manufacturer provides code examples
- code size limit
- $795.00 per seat for version with sufficient code size

Rowley Crossworks

+ Reasonably simple build and debug process
+ One team member experienced and familiar with usage
- Unique precompiler conventions. Example code is unlikely to work.
- code size limit
+ $150/seat personal noncommercial license available

Based on the difficulty in using mspgcc and the prohibitive cost of the other two options, Rowley Crossworks was selected as the compiler and debugger software. To ensure broad compatibility, the TI FET programmer/ debugger hardware was chosen, but competing commercial and open hardware alternatives are rapidly evolving and should be monitored for maturity.

Hardware Development:

This project included participation in the requirements, design and fabrication processes of prototype hardware. From the first statement of requirements, roughly 18 months elapsed before a first iteration of hardware was in hand and usable for development. This was by far the most time-consuming aspect of in this project, though outside the scope of this report.

Dev Board Design

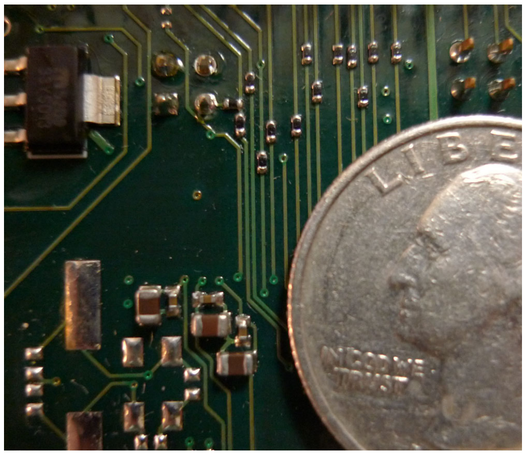
The initial goal was to produce a simple, experimenter-friendly carrier board for rapid prototype development. This “dev board” includes most of the common hardware elements that each subsystem of the satellite is intended to share, plus extra components such as a serial communication bridge, breakout areas for microcontroller pins, indicator lights, and jumpers to choose between optional hardware connections.

The schematic and layout of the dev board have not been approved for public release and can not be included here. They are considered proprietary information by the Alaska Space Grant, but might be made available on request. Please see contact information at http://spacegrant.alaska.edu for more information.

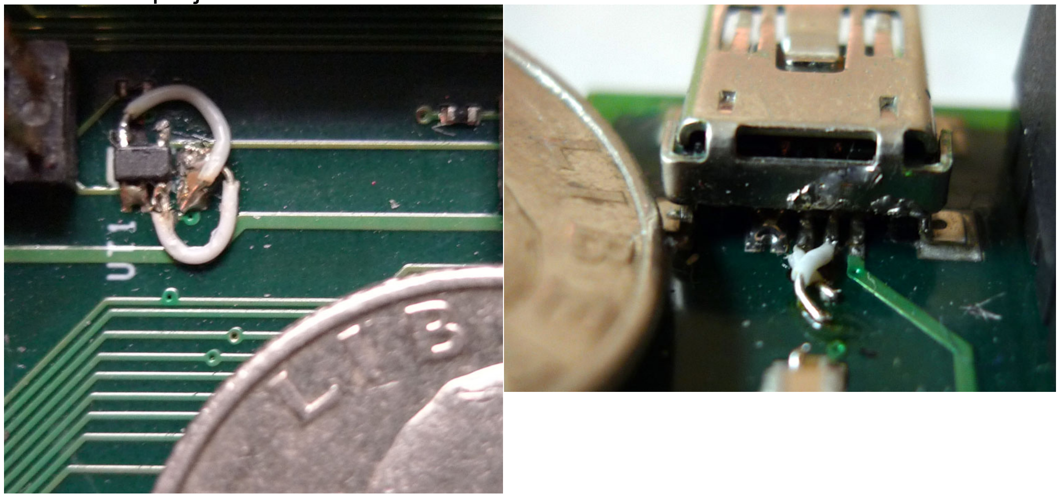
Given that these designs are simplistic by most industry standards, this may seem unusual to many readers, but this closed mode of development seems common in space systems engineering projects. This may be due to a combination of fear of United States arms trafficking laws and a lower level of penetration of open source methodologies that are more widespread elsewhere in academics. A desire to exert control over how these designs are represented and potentially reused has also been expressed.

Physical Assembly

Several parts and layout combinations in the dev board were especially small, and beyond the ability of anyone not expert in soldering. Because of this, a pick-and-place machine and reflow oven were required for board assembly. While the vacuum head and high-zoom cameras allow for precise and highly reliable assembly, these techniques are extremely slow and require a trained, careful and focused operator. The number of dev boards created is far fewer than hoped and they have not yet filled their potential as a commonly used prototyping tool.

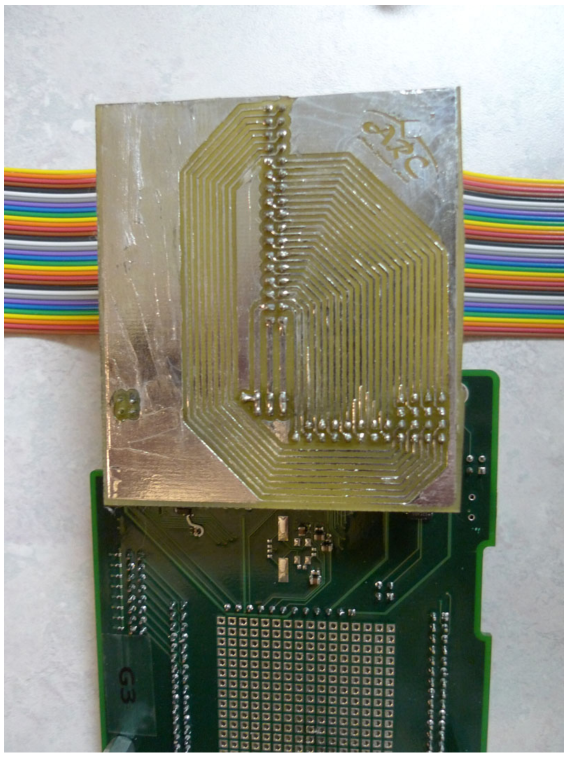

Detail image of dev board components.

One consequence of the waterfall-style (described as the “V Model”[VMODEL]) development methodology was that the first revision of the dev board was purchased in bulk, and instead of correcting errors discovered in the first version and purchasing corrected circuit boards, tiny jumper wires are soldered manually for every dev board produced. A few of the jumper soldering jobs have been accomplished only by the author and the lead Electrical Engineer on the project.


Detail images of wire jumper corrections.

Complete dev boards are designed to interconnect using stackthrough headers, identical to those that will be used in the final satellite assembly. However, this method of connection is both extremely high friction and difficult to disconnect, and leaves little to no room for physical access to the prototyping areas of each board.

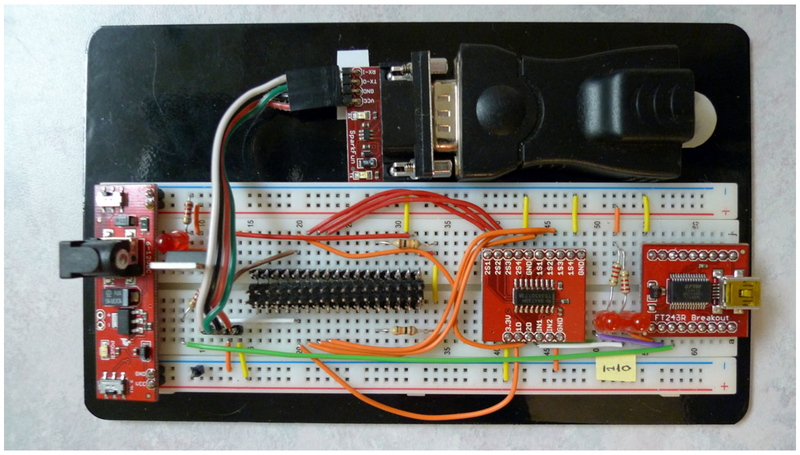
A side-by-side layout would be much more usable for prototyping and testing tasks. Commonly available ribbon cable connectors are sufficient for connecting the boards, but the dev boards are not laid out in a way conducive to this arrangement. Given the other demands on the time of other project members and the extremely long turnaround on board design, requesting a change to the dev boards or an extra board was infeasible. To overcome this, simple carrier boards were designed using freeware schematic and layout software and etched to bare copper boards using toner transfer methods. This rapid prototyping capability proved to be essential to overcoming bottlenecks in the availability of hardware development resources on the project.


Hand-etched carrier circuit board connecting a dev board to the ribbon cable.

Switching and Control Circuit

The Supervisor Circuit that will manage power resets for error recovery. A prototype of the final circuit has not yet been developed, so a temporary stand-in is required to fill these functions for the test harness.

Use another dev board for the circuit, using the peripherals on the dev board for communication channels and general-purpose I/O pins to control power switching and multiplexed communication channels. This would have the advantage of being able to re-use code from the firmware from the other dev boards. If the dev boards weren’t so difficult to populate and solder corrective jumper wires, this would have been the most attractive option.
Use another microcontroller break-out board and breadboard to create the circuit. This could be another MSP430 or another convenient chip. This could potentially be simpler than using a complete dev board, but offer fewer reuse options.
Use off-the-shelf parts that specialize in the required capabilities and do not require firmware. This offloads potential complexity to PC software, but reduces the time invested in embedded development. For the initial test harness, this option was chosen as the requiring the most minimal time investment.


Control Circuit Prototype with COTS Parts

Firmware Development:

One portion of this project was to develop firmware for the ARC development boards using the MSP430F2618 microcontroller. This firmware serves as a proof-of-concept and exploration of basic methods that will serve as the groundwork for future satellite development.

Availability of Prior Work and Examples

Unlike many programming projects, a simple internet search is unlikely to turn up adequate example code for a particular problem. Reference examples provided by hardware manufacturers and compiler vendors tend to be extremely simplistic, because more complicated examples lose general applicability or may not be approachable. Even if references are found for a particular microcontroller line, they may be for a different serial peripheral, work only with one compiler, or use peripherals in a different context.

The best available resources tended to be other students or professors. It is possible that the accumulation of experience with this development environment can lead to better knowledge transfer for future students. However, given rapid changes in microcontrollers, most current code faces a significant risk of obsolescence even in the short term.

This challenge indicates a greater need for more frequent development iterations, more exchange of information between team members and more testing. The best immediate response to this observation is to communicate it back to the team to be included in risk, cost and time estimates.

Firmware Structure

The firmware is intended to eventually be a model for Cubesat flight firmware, so an initial goal is to use as few resources and introduce as little complexity as possible. The firmware model can be roughly divided into four categories:

- Initialization and configuration of memory and registers, especially the control registers for peripherals and associated interrupts.
- Interrupt handling routines.
- Utility functions for routine operations, including peripheral management.
- A main event loop.

If possible, all of the configuration settings required to use a peripheral in a given mode are collected into one place and structured in such a way that they can be mixed and matched for different needs. Some peripheral-handling functions, such as the UART routines, are at a high level of maturity and have been broken out into separate header and source files for group use. Others, such as the I2C handling routines, are not yet good enough to redistribute in any finished form.

The interrupt handling routines cannot be as easily encapsulated, as the same vector may be used by multiple peripherals in varying states. Interrupts are kept as minimal as possible, reading or writing to predefined dedicated memory buffers and setting a flag for the main loop to later handle further processing. One unresolved issue with peripheral interrupts is the resolution of error and wait states. At present, the team has not yet settled on best-practice principles for these situations.

Each peripheral requires interface code to be written along with appropriate data structures. Additionally, a few simple functions are provided to create testable capabilities and as a stand-in for eventual mature satellite development. These structures are responsible for managing the state of each peripheral and the state of the microcontroller as a whole.

Finally, an extremely simple main loop handles flags raised by communication peripherals or timers. Display LED states are managed here, as well as processing actions that must occur in a specific sequence, such as sending a command to another board via I2C after external communication has been received via UART. Commands and external communication can be triggered asynchronously from within the main loop as well.

Low-Power Modes

The current firmware implementation does not take advantage of low power modes because of the complexity of potential interactions between multiple peripherals operating simultaneously. Each would need to manage its own set of registered interrupts for leaving low power mode without disabling the set required for the others. This test harness does not yet include a power minimization component.

With correct power management, MSP430 power consumption tends to be linearly proportional to the speed of the microcontroller and the amount of time spent in operation. This makes it possible to “pay as you go” for uC power. One satellite engineer at the 2009 Cal Poly Cubesat workshop advised against over-analysis of this area, though. If there is a 5 watt transmitter module in operation, the relative gain from saving micro- or milliwatt consumption of power can be insignificant.

Standard Libraries

To the surprise of this investigator, mature and reliable implementations of many ANSI C libraries were provided with the CrossWorks compiler. Once appropriate low-level supplementary code had been written, such as a putchar() implementation, relatively complicated tasks such as string handling and formatted output could be written using familiar language constructs. These libraries would work (in theory) even on the lower-end microcontrollers in the MSP430 series, provided enough memory space. This may be a common and expected experience for experienced embedded developers, but for a computer scientist unfamiliar with the product, it was a welcome discovery.

Selection of a Minimal Command Set

One goal of the project firmware was to perform actions representative of the actual tasks that satellite subsystems will perform. A small set of commands were chose for this purpose.

The command set for the test harness firmware should be as minimal as possible while still demonstrating the ability to:

- Demonstrate basic responsiveness.
- Store and retrieve arbitrary sets of data.
- Demonstrate communication between development boards.
- Exchange data between boards.
- Demonstrate the ability to act on commands sent between boards.
- Perform work on command.
- Simulate single-event-upset events on command.

Assuming that the firmware can

The command set used for test runs and observations of error rates was:

Command	Action
Set LED Pattern	Immediately change the indicator light pattern variable and update the port controlling the indicator lights.
Read LED Pattern	Report the value of the indicator light pattern variable.
Remote Set LED	Send a command over the bus to a specified board to set its LED pattern.
Set Work Area Buffer Segment	Set the contents of the specified portion of the data buffer.
Read Work Area Buffer Segment	Report the contents of the specified portion of the data buffer.
Perform Calculation on Buffer	Perform a predetermined calculation on the entire data buffer.
Send Buffer Segment to Other Board	Send a segment of the data buffer with a given offset and length to another board.
Flip Bits (Intentionally corrupt memory)	Intentionally corrupt volatile memory by performing an XOR of the designated location with a bit mask. Typically used for one bit at at time.

Switching Control Circuit Prototype

The switching control circuit serves as both a test harness and as a stand-in for the Supervisor Circuit that will be responsible for restoring the satellite when errors are detected. Three options for creating this circuit were considered:

- Use another dev board for the circuit, using the peripherals on the dev board for communication channels and general-purpose I/O pins to control power switching and multiplexed communication channels. This would have the advantage of being able to re-use code from the firmware from the other dev boards. If the dev boards weren't so difficult to populate and solder with corrective jumper wires, this would have been the most attractive option.
- Use another microcontroller break-out board and breadboard to create the circuit. This could be another MSP430 or another convenient chip. This could potentially be simpler than using a complete dev board, but offer fewer reuse options.
- Use off-the-shelf parts that specialize in the required capabilities and do not require firmware. This offloads potential complexity to PC software, but reduces the time invested in embedded development. For the initial test harness, this option was chosen as the requiring the most minimal time investment. Fortunately, the FT245[FTDI] device drivers proved easy to use.

PC Software Development:

Serial Communications

In the Windows en environment, there are a large number of serial libraries and tools available. For this project, a serial library that is simple to use, free and that works with all of the hardware drivers in use has not yet been identified. An ideal candidate would also be open-source.

The current library, CSerialPort[CSERIAL], is not the ideal choice because it does not work with all of the virtual com port drivers in use elsewhere, but it was sufficient for the inital test harness implementation.

Switching Hardware Control

The switching hardware control relied on PC software-controlled GPIO pins provided by the Fortunately, the manufacturer provided libraries were easy to use and easy to integrate with the other portions of the codebase. A simple console-based program was produced to control the multiplexing of serial communications and the power reset. This program can be used either interactively or by passing control characters to its input stream.

Simple Automated Test Program

Automated test case generation is fairly straightforward. Commands can be sent according to a pre-generated list or generated randomly, with or without a fixed seed. First, the switching control hardware is instructed to connect to a specified dev board. Then the command is sent via the serial interface and the harness enters an idle mode for a fixed time period. If no acknowledgement is received from the targeted board, then it is assumed to have entered a fault state and a reset is performed.

A data structure records the expected state of selected memory on each microcontroller so that the test harness can maintain an independent simulation of the satellite function. This facilitates both to validate SEU simulation and as a software debugging mechanism. This mechanism allows unit testing of individual subsystems and regression testing against known good previous system configurations.

Some commands exist primarily to induce and check for error states. Based on a model of the addressable satellite memory, bits to flip are chosen to simulate SEU errors. Currently this model is uniformly random, but may be informed by more extensive analysis of error distributions in the future. For example, SEU upset of control channels could result in errors concentrated in some area of memory.

The test harness collects the responses from subsystems to each command and summary statistics of command results. As useful metrics are developed to express satellite functionality, these recorded results are expected to increase in usefulness.

Error Observation and Analysis

Observed Error Modes

Aside from recoverable errors in the reported data buffers, by the most commonly observed error mode was that the embedded network becomes non-responsive. This condition is extremely difficult to automatically diagnose. Human intervention is required at that point to attempt to characterize the error. A hardware debugger may be attached to one of the microcontrollers, but it is possible that timing-dependent behaviors may occur differently in that case. In many cases, whether an error mode is attributable to a bit flip event or an inherent weakness in the firmware implementation is an open question.

Methods of Error Measurement

In the absence of methods of soft recovery from errors, each experimental run terminates at the point where the system fails to respond to further commands and must be reset.

Because the test harness in this project represents a point in time for the cubesat development, the actual numbers produced are far less valuable than its contribute to implementation and debugging. As the satellite firmware reaches maturity across several components, these metrics will be tuned to expected upset rates for the space environment and become the primary measurement of expected satellite performance.

The test harness records:

- Number and type of commands sent until failure.
- Number of simulated SEU disruptions until failure.
- Distribution of commands sent, summary counts and command sequence.
- Approximate temporal distribution of commands, to selectively force command queueing, conflicting or dependent command resolution and other time-dependent conditions.

Probable Actual Sources of Error

Errors were mass-produced intentionally by the testing process. For a relatively immature research product such as this project, it is reasonable to conjecture that other sources of error occur, mixed in with the intentionally simulated results. The bulk of errors were not investigated individually. A random sampling of observed errors were analyzed and a rough characterization of observed error categories follows:

Induced Errors

Known Firmware Weaknesses

- At the time of this project, some elements of the serial peripheral firmware implementation are known to be incomplete or lack appropriate error handling. This became especially apparent during higher-speed testing where the microcontrollers would be expected to be forced to wait or arbitrate use of the shared bus.

- The serial peripherals were intentionally configured to run at much less than their maximum speed (50 KHz or less). This was done to ease diagnosis with inexpensive logic analyzer hardware as well as reduce the chance of signal integrity issues due to long wires in the test bed. However, this also introduced much more frequent bus collisions and artificially long waits for the microcontroller CPU while the serial peripheral completed a transaction. These combined tended to exaggerate the impact of weaknesses in the interrupt and state handling.
- This is one of several ways in which the value of the automated testing harness is demonstrated, as can is difficult and tedious to manually test issues such as these.

Firmware bugs

- This project does not represent a mature, commercially viable level of reliability, much less the extreme reliability required for space operations. It is reasonable to expect that undiscovered bugs in the initial firmware contribute significantly to observed error rates. Given the frequency at which government and commercial projects experience errors that are either uncategorized or lumped in the general category of “software malfunction,” the ability to load-test candidate systems and expose these errors may be of greater value to the success of the project than the stated purpose of simulating single event upset errors.

Control Hardware Timing and Stability

- The control hardware appeared to have some switching latency, which may be related to the relatively crude breadboard techniques or poor understanding of proper electrical design for the components used. It was possible to induce a higher rate of error by shortening the wait to send or receive signals after switching the control circuitry to address a different board.

UART Errors

- The frequencies used by the microcontroller peripheral and USB-to-UART bridge chip are approximate, sometimes off by as much as one to three percent. This depends on such factors as compromises in clock and peripheral configuration and factory calibration of the microcontroller. Simple software measures such as sentinel values to reject noise and checksums were employed, but a thorough analysis of UART-induced error was not performed.

Conclusion/Results:

This project included extensive group and individual work aimed at both arriving at the first implementation of the satellite test and development harness and at the larger goal of successfully flying UAF’s first satellite. It would have been far easier to produce a project like this one individually, without requirements for design review proceedings for any major decision or document and without producing the extensive and proprietary requirements and design documents required for NASA review and to meet the Space Grant goal of preparing students to work in the government and commercial aerospace fields.

At the inception of this project, no software or hardware architecture had yet been evaluated and all major technical decisions leading up to its present state had to be researched and evaluated. This project included creating and promoting the concept of the development boards and their original statement of requirements. It included participation in the design and layout review process for them as well. Nearly all analysis of expected error modes and planning of recovery mechanisms was left up to this project to plan and complete.

Because the Space Grant did not provide any hardware that could be used for development or presentation, parts were sourced and purchased out-of-pocket. Once purchased, fabrication of the dev boards was up to the individuals who needed them, so the requisite tools were learned and three boards were fabricated and tested.

Only after all of this work was completed could any substantial software be written. Because a development environment was not yet available, programming hardware, a compiler and other parts and tools such as a logic analyzer were also purchased and incorporated into the test harness capabilities.

The firmware test bed successfully demonstrated all of the intended commands in interactive testing with and without debugging hardware attached. One important discovery was just how easily failure can be induced in a stress-test of not-yet-mature firmware. Though familiar to the engineering department at UAF, resources on how to develop extremely highly reliable software for the MSP430 processor were difficult to obtain. Even the most senior and knowledgeable team members had only a vague concept of how to properly implement I^2C communication, and that task was part of this project.

Important Lessons Learned

- Design for ease-of-prototyping. The smallest possible components, tightly packed, might be optimal for a production design, but are hardest to reliably assemble. Including too many capabilities in a single, non-modularized unit only multiplies the number of things that can go wrong and the difficulty of diagnosis when something does go wrong.
- Many modern software engineering methodologies have not reached broad adoption in some areas of aerospace engineering. Interdisciplinary developers must take care to understand the perspective that all team members will bring to the design process.
- The final Cubesat software as specified for the Alaska Research Cubesat will require significant effort to achieve completion and reliability. This may sound like a trivial observation, but this project revealed an interesting way to assess project effort. by choosing one aspect of the project, and pursuing it in depth independent of other aspects, a more full understanding of the complexity and difficulty can be acquired. This understanding may translate usefully to other parts of the project. This also suggests an interesting general method of effort and cost estimation, by picking a narrow “slice” of the available work and pursuing it alone to a higher level of maturity than the rest of the project, the probably effort required for related work can be more accurately estimated!
- The kind of prior work and examples that are commonly assumed to be available for many software development projects may be significantly less extensive or harder to find for embedded or aerospace projects.
- Automated testing can reveal errors in firmware that are difficult to detect even after many hours of interactive, human-driven testing.

Future work

1. Merge the test firmware code with work done by other team members and converge as quickly as possible on a shared code base for future development.
2. Continue work on all of the highest priority satellite Command and Data Handling tasks, such as a more complete command set, a final framework and code base for the firmware, and many other needs.
3. Once the simulated single-event upset methods have achieved a high level of confidence, experiment with locally available sources of radiation to see if upset events can be induced and whether the resulting error modes substantially agree with those produced in software.
4. Identify a better serial communication library or toolset for the PC software. This turned out to be

surprisingly difficult challenge in the Windows environment. Serial communication has become a niche capability with a surprisingly error-prone field of tools. It seems simple at first to just invoke UART as a communication channel, but it might be worthwhile to reconsider it in favor of USB serial communication for future work.

5. Start testing real satellite subsystems in preparation for a trip to space!

References and Related Links

[PPOD] Jordi Puig-Suari, Clark Turner, William Ahlgren, “Development of the Standard CubeSat Deployer and a CubeSat Class PicoSatellite” Retrieved 11/30/2011: http://www.cubesat.org/images/Papers/cubesat_paper.pdf

[GENSO] GENSO introduction: http://www.genso.org/

[ITAR] International Traffic in Arms Regulations: http://en.wikipedia.org/wiki/ITAR

[CSERIAL] CSerialPort v1.27 An MFC class for Win32 serial ports: http://www.naughter.com/serialport.html

[VMODEL] V-Model: http://en.wikipedia.org/wiki/V-model

[FTDI] Future Technology Devices International Ltd: FT245R - USB FIFO IC: http://www.ftdichip.com/Products/ICs/FT245R.htm

I.Galysh , K. Doherty , J. McGuire , H.Heidt , D. Niemi , G. Dutchover, “CubeSat: Developing a Standard Bus for Picosatellites” Retrieved 11/30/2011: http://www.cubesat.org/images/Papers/stensat_hist.pdf

Ryan Nugent, Riki Munakata, Alexander Chin, Roland Coelho, Jordi Puig-Suari, “The CubeSat: The Picosatellite Standard for Research and Education” Retrieved 11/30/2011: http://www.cubesat.org/images/More_Papers/cps2008.pdf

Alaska Space Grant Program: http://spacegrant.alaska.edu/

The CubeSat Project: http://www.cubesat.org/

Rowley Associates, CrossWorks for MSP430 - C Compiler for MSP430 Microcontrollers: http://www.rowley.co.uk/msp430/

MSP430 Microcontroller - Texas Instruments: http://www.ti.com/lsds/ti/microcontroller/16-bit_msp430/overview.page?DCMP=MCU_other&HQS=Other+IL+msp430

Appendix 1: Physical Layout of Stackthrough Header Pins

Prospective UAF Cubesat PC/104 Pin Designations

	EN1 2	EN3 4	EN5 6	EN7 8	10	12	14	16	18	20	TORQ 22	TORQ 24	5V 26
Header 2 (2x26)	EN0 1	EN2 3	EN4 5	EN6 7	9	11	13	15	17	19	TORQ 21	TORQ 23	5V 25
	PRG 2	TCK 4	TDI/TC 6	BSL 8	P1.1 10	P1.3 12	P1.5 14	P1.7 16	I2Cdat 18	TX_UA 20	MISO 22	i2cON 24	PXX 26
Header 1 (2x26)	PWRO 1	!RST/N 3	TMS 5	TDO/T 7	P1.0 9	P1.2 11	P1.4 13	P1.6 15	I2CCLI 17	RX_UA 19	SPICLI 21	MOSI 23	PXX 25

	3V3 28	GND_ 30	GND_ 32	PPSw 34	SepSv 36	PXX 38	PXX 40	BatCo 42	BatCo 44	BatBu 46	TORQ 48	TORQ 50	TORQ 52
Header 2 (2x26)	3V3 27	GND_ 29	PXX 31	PPSw 33	SepSv 35	PXX 37	PXX 39	BatCo 41	BatCo 43	BatBu 45	TORQ 47	TORQ 49	TORQ 51
	PXX 28	PXX 30	5VUSI 32	PXX 34	PXX 36	PXX 38	PXX 40	PXX 42	PXX 44	PXX 46	TORQ 48	TORQ 50	TORQ 52
Header 1 (2x26)	PXX 27	PXX 29	PXX 31	PXX 33	PXX 35	PXX 37	PXX 39	i2cDat 41	i2cCLI 43	PXX 45	TORQ 47	TORQ 49	TORQ 51

Header 1 Pin Designations

I2C Command Bus Signals		
17	I2CCLK	Clock Signal for the I2C command bus
18	I2Cdat	Data signal for the I2C command bus
SPI Data Bus Signals		
22	MISO	Master-In, Slave-Out SPI line
23	MOSI	Master-Out, Slave-In SPI line
21	SPICLK	SPI clock signal
Interrupt Bus and Bootstrap Loader Pins		
8	BSL	Bootstrap Loader RX (P2.2 on the MSP430F261x)
9-16	P1.[0-7]	Interrupt bus pins. P1.1, pin 8, is also used for BSL TX
JTAG Programming/Debugging/BSL Enable Pins		
1	PWROFF	Power
2	PRG	Reserved for future or non-MSP programming pins
3	!RST/NMI	Reset pin
4	TCK	JTAG test clock
5	TMS	JTAG test mode select
6	TDI/TCLK	JTAG test data/clock input
7	TDO/TDI	JTAG test data out programing data in
UART Ground Communication		
19	RX_UART	UART RX channel
20	TX_UART	UART TX channel

Header 2 Pin Designations

Per-Board JTAG/BSL Enable Switch Pins		
1	EN0	Enable 0: Master Controller
2	EN1	Enable 1: Communication
3	EN2	Enable 2: Vibration/Thermal
4	EN3	Enable 3: ACS/ADS
5	EN4	Enable 4: Camera
6	EN5	Enable 5: [Potential Torquer 1]
7	EN6	Enable 6: [Potential Torquer 2]
8	EN7	Enable 7: [No projected use yet]

Reserved for Potential Torquer Firing Signals		
47-52	TORQUE	Control signals for torquer firing circuits
Reserved to Avoid Designated-Purpose Pumpkin Pins		
25-31, 33-40, 42, 44-46	PXX	Pumpkin uses this pin for something other than GPIO, to avoid hypothetical conflicts, use last.
Clyde Space Board Pin Designations		
24	i2cON	Selected for i2c communications
32	5VUSB	5V rail from USB connector
41	i2cDat	I2C bidirectional data
43	i2cCLK	I2C bidirectional clock

Reserved for Potential Torquer Firing Signals		
21-24, 47-52	TORQUE	Control signals for torquer firing circuits
Reserved to Avoid Designated-Purpose Pumpkin Pins		
25-31, 33-40, 42, 44-46	PXX	Pumpkin uses this pin for something other than GPIO, to avoid hypothetical conflicts, use last.
Clyde Space Board Pin Designations		
25-26	5V_BUS	5B Bus
27-28	3.3V BUS	3.3V Bus
29	GND_RF	Analog RF Ground
30	GND_DIGI	Digital Logic Ground
32	GND_HIPC	High-Power Systems Ground
33-34	VBAT(PP_	Pull Pin normally connected switch
35-36	VBAT(SS_	Separation Switch normally connected pin
41-44	VBAT(PP_	Common point for pull pin and separation switch
45-46	BAT_BUS	Output Battery Bus

Appendix 2: Source Code

```
// main.c for the unified test harness firmware, combining capabilities from
// the separate firmware development projects for master and slave units.

// This code targets the MSP430F2618 device specifically.

// Requires the Rowley Crossworks MSP430 compiler and related libraries

// Gregg Christopher, 2011

//#include <__cross_studio_io.h>
#include <msp430x26x.h>
#include <stdio.h>
#include <string.h>

#include "arc_uart_uca1.h"

#define MY_ID 1

int x, y, z;
short timer_flag;

unsigned char RXData;
unsigned char RXCompare;
unsigned char TXData;
unsigned char TXByteCtr;

unsigned char LEDstate;
unsigned short data_in;

#define WORK_BUF_SIZE 30

unsigned char work_buf[WORK_BUF_SIZE];

#define I2c_RX_BUF_LEN 32

enum { I2C_IDLE = 0,
       I2C_TX,
       I2C_RX,
       I2C_COMPLETE,
       I2C_ERROR };

// ggg: This is a VERY poor way to set up a command table. Replace it with a registration process, or machine-
generated code.
enum { EXT_CMD_SETLED   = 0x00,
       EXT_CMD_GETLED   = 0x01,
       EXT_CMD_ORDERLED = 0x02,
       EXT_CMD_SETBUF   = 0x03,
       EXT_CMD_READBUF  = 0x04,
       EXT_CMD_DOCALC   = 0x05,
       EXT_CMD_SENDBUF  = 0x06,
       EXT_CMD_FLIPBIT  = 0x07 };

enum { BUS_CMD_SETLED   = 0x00,
       BUS_CMD_SETBUF   = 0x01 };


unsigned char *I2Cdat;                         // Pointer to buffer for I2C data TX
unsigned short I2C_len = 0, I2Cidx = 0;             // Length and current data index for I2C data TX buffer
unsigned short I2Cstate = I2C_IDLE;                 // State of UCB1 I2C peripheral
unsigned char I2C_RX_buf[I2c_RX_BUF_LEN];           // I2C RX buffer
unsigned short I2C_RX_dat;                          // Flag indicating a complete transmission has been read
into the I2C RX buffer

void arc_I2C_setup(void){
  UCB0CTL1 |= UCSWRST;                              // Put UCB1 into reset state

  // Register Set Up
  UCB0CTL0 = UCMM | UCMST | UCMODE_3 | UCSYNC;      // Multi Master, Master, I2C Mode, synchronous mode
  UCB0CTL1 |= UCSSEL_2;                             // Clock source for I2C unit is SMCLK
  UCB0BR0=0x40;                                     // Set prescaler to run at ~50 kB / s for 16 Mhz SMCLK
  UCB0BR1=0x01;
  UCB0I2COA = 0x30 + MY_ID;                         // Own Address is calculated

  P3SEL |= 0x06;                                    // Assign P3 I2C pins to USCI_B0

  UCB0CTL1 &= ~UCSWRST;                             // Bring UCB1 out of reset state

  // ggg: temporary code needs a better implementation for handling bad I2C initial states
  // check if bus is busy
  if(UCB0STAT&UCBBUSY){
        //bit bang the bus until no device has hold of it
        while(1); //trap for debugger
        //I2C_reset();
  }

  UCB0I2CIE |= UCNACKIE;                            // Enable not-acknowledge interrupt
  UC0IE |= UCB0RXIE;                                // Enable RX interrupt
  UC0IE |= UCB0TXIE;                                // Enable TX interrupt

}

unsigned short i2c_tx(unsigned short addr, const unsigned char *dat, unsigned short len){
  if(len==0) return -1;                             // Check for zero length

  while(UCB0STAT&UCBBUSY);                          // Wait for the bus to become free
  UCB0I2CSA = addr;                                 // set slave address
  UCB0CTL1 |= UCTR;                                 // set transmit mode
  UC0IFG &= ~UCB0TXIFG;                             // Clear TX interrupt flag
  UC0IE &= ~UCB0RXIE;                               // Disable RX interrupt
  //UC0IE |= UCB0TXIE;                              // Enable TX interrupt
  I2Cidx = 0;                                       // Set buffer index
  I2Cstate = I2C_TX;                                // Clear done flag
  I2C_len = len;                                    // Set length
  I2Cdat = (unsigned char*)dat;                     // Set TX data buffer pointer
  UCB0CTL1|=UCTXSTT;                                // Generate start condition
  // Go to LPM0 during transaction (ggg: why is this a good idea?)
  while(I2Cstate != I2C_COMPLETE && I2Cstate != I2C_ERROR){
        __bis_SR_register(LPM0_bits + GIE);
  }
  // ggg: we probably don't want to hang the CPU while transferring. This should be changed to a handoff with
callbacks for error handling.
  while (UCB0CTL1 & UCTXSTP);                       // Wait for stop condition to be sent
  // TO DO: check I2Cstate for error and handle accordingly (ggg: need conversation with Jesse about this and
other I2C exception states!)
  I2Cstate = I2C_IDLE;                              // Indicate that the peripheral is no longer used
  UC0IE |= UCB0RXIE;                                // Enable RX interrupt
  return I2Cidx;                                    // Return number of bytes sent
}
```

```
void do_calc() {
  unsigned int ii;

  for(ii = 0; ii < WORK_BUF_SIZE; ii++) {
        work_buf[ii]++;
  }

}

void corrupt_memory(unsigned int loc, unsigned short flip_mask) {
  char *p = (char *)loc;
  printf("Change location %04X by mask %02X\r", loc, flip_mask);
  *p ^= flip_mask;
}

int ext_command(char *cmd_str, unsigned int cmd_len) {
  unsigned int ii;
  char send_buf[10];
  unsigned int send_len;
  unsigned short flip_mask;
  if(cmd_len < 1) return 0;
  //if(cmd_str[0] != '!' || cmd_str[1] != '#') return 0;
  switch(cmd_str[0]) {
  case EXT_CMD_SETLED:
        LEDstate = cmd_str[1];
        P7OUT = (LEDstate & 0x7F) | (P7OUT & 0x80);
        break;
  case EXT_CMD_GETLED:
        printf("LED State: %02X\r", LEDstate);
        break;
  case EXT_CMD_ORDERLED:
        I2C_RX_buf[0] = BUS_CMD_SETLED;
        I2C_RX_buf[1] = cmd_str[2];
        if(2 != i2c_tx(0x30 + cmd_str[1], I2C_RX_buf, 2)) {
        printf("Failed to send LED change command.\r");
        return 0;
        }
        break;
  case EXT_CMD_SETBUF:
        for(ii = 0; ii < cmd_str[2]; ii++) work_buf[cmd_str[1] + ii] = cmd_str[3 + ii];
        break;
  case EXT_CMD_READBUF:
        for(ii = 0; ii < cmd_str[2]; ii++) printf("%02X ", work_buf[cmd_str[1] + ii]);
        printf("\r");
        break;
  case EXT_CMD_DOCALC:
        do_calc();
        break;
  case EXT_CMD_SENDBUF:
        I2C_RX_buf[0] = BUS_CMD_SETBUF;
        I2C_RX_buf[1] = cmd_str[2];
        I2C_RX_buf[2] = cmd_str[3];
        for(ii = 0; ii < cmd_str[3]; ii++) I2C_RX_buf[ii+3] = work_buf[cmd_str[2] + ii];
        ii += 3;
        if(ii != i2c_tx(0x30 + cmd_str[1], I2C_RX_buf, ii)) {
        printf("Failed to send LED change command.\r");
        return 0;
        }
        break;
  case EXT_CMD_FLIPBIT:
        ii = cmd_str[1];
        ii = ii << 8;
        ii += cmd_str[2];
        flip_mask = cmd_str[3];
        flip_mask = flip_mask << 8;
        flip_mask += cmd_str[4];
        corrupt_memory(ii, flip_mask);
        break;
  default:
        return 0;
        break;
  }
  return 1;
}

int i2c_command() {
  int ii;

  switch(I2C_RX_buf[0]) {
  case 0:
        LEDstate = I2C_RX_buf[1];
        P7OUT = (LEDstate & 0x7F) | (P7OUT & 0x80);
        break;
  case 1:
        for(ii = 0; ii < I2C_RX_buf[2]; ii++) work_buf[I2C_RX_buf[1] + ii] = I2C_RX_buf[ii+3];
        break;
  default:
        return 0;
  }
  return 1;
}



void main(void)
{
  unsigned char aptr;
  volatile int i;
  unsigned int ii;

  WDTCTL = WDTPW + WDTHOLD;                         // Stop WDT
  if (CALBC1_16MHZ ==0xFF || CALDCO_16MHZ == 0xFF)
  {
        while(1);                                   // If calibration constants erased do not load, trap CPU!!
  }

  BCSCTL3=XCAP_0;
  //setup clocks

  //set DCO to 16MHz from calibrated values
  DCOCTL=0;
  BCSCTL1=CALBC1_16MHZ;
  DCOCTL=CALDCO_16MHZ;

  init_uart_uca1();

  // Set up Timer 1 for 1 Hz-ish operation for human-friendly event timing.
  TACTL=TASSEL_1|TACLR; //|ID_3;
  TACCR0 = 32768/2;
  TACCTL0 = CCIE;
  TACTL |= MC_1;
  timer_flag = 0;

  P1DIR |= 0x01;                             // Set P1.0 to output direction
```

```
//P1IE |= 0x010;                          // P1.4 interrupt enabled
//P1IES |= 0x010;                         // P1.4 Hi/Lo edge
//P1IFG &= ~0x010;                        // P1.4 IFG cleared

P7DIR = 0xFF;                             // P7.x output
P7OUT = 0x00;

arc_I2C_setup();

P1OUT = 0x01;

z = y = x = 0;

__enable_interrupt();

for(i=0;i<32000;i++);
for(i=0;i<32000;i++);
for(i=0;i<32000;i++);
aptr=0x3a;

data_in = 0;
LEDstate = 0x00;
uart_len = 1;

//printf("Hello World.\r");
//printf("Initial LED state: %02x\r", LEDstate);

while(1) {

      if(P1IN & 0x10) {
      //P7OUT |= 0x02;
      }

      // If a terminated string is waiting in the input buffer, process it.
      if(RxBuf.term_str) {
      //printf("Got a terminated string.\r");
      uart_len = 0;
      for(uart_len = 0; uart_len < RxBuf.term_str; uart_len++) uart_in[uart_len] = RxBuf.buf[uart_len];
      RxBuf.term_str = 0;
      //while(RxBuf.term_str == 1) uart_in[uart_len++] = arc_getc();
      uart_in[uart_len] = '\0';
      //printf("String: %s\r", uart_in);
      if(!ext_command(uart_in, uart_len)) {
      printf("Unable to handle command.\r");
      }
      //printf("Transmitting Command: %02X\r", uart_in[0]);
      }

      if(I2C_RX_dat) {
      i2c_command();
      //LEDstate = I2C_RX_buf[0];
      //P7OUT = (LEDstate & 0x7F) | (P7OUT & 0x80);
      I2C_RX_dat = 0;
      }

      if(timer_flag) {
      P7OUT ^= 0x80;
      timer_flag = 0;
      //LEDstate = uart_in[0];
      //if(P7OUT & 0x80) printf("Board %d is alive. Last LED state: %X\r", MY_ID, LEDstate);

      TXByteCtr = 1;                      // Load TX byte counter
```

```
        if(MY_ID == 0) {
        if(uart_len != i2c_tx(0x32, uart_in, uart_len)) {
        //if(1 != i2c_tx(0x32, uart_in, 1)) {
        //error sending data
        P7OUT |= BIT2;
        P7OUT &= ~BIT1;
        printf("Errror Writing Address Pointer\r\n");
        }
        }
        }
  }
  //__bis_SR_register(LPM0_bits + GIE);      // Enter LPM0, interrupts enabled

}

void Port_1(void) __interrupt[PORT1_VECTOR]
{
  //P7OUT = 0x10;
  //P1OUT ^= 0x01;                           // P1.0 = toggle
  P1IFG &= ~0x010;                           // P1.4 IFG cleared
}

//void USCIAB0TX_ISR(void) __interrupt[USCIAB0TX_VECTOR]
//{
  //RXData = UCB0RXBUF;                      // Get RX data
  //__bic_SR_register_on_exit(CPUOFF);       // Exit LPM0
//}


void hsec_timer(void) __interrupt[TIMERA0_VECTOR]{
  timer_flag=1;
}


/*void USCIAB0TX_ISR(void) __interrupt[USCIAB0TX_VECTOR]
{
  P7OUT |= 0x08;
  if(UC0IFG&UCB0TXIFG) {
        P7OUT |= 0x10;
        if (TXByteCtr)                              // Check TX byte counter
        {
        P7OUT |= 0x20;
        UCB0TXBUF = TXData;                  // Load TX buffer
        TXByteCtr--;                         // Decrement TX byte counter
        }
        else
        {
        UCB0CTL1 |= UCTXSTP;                 // I2C stop condition
        IFG2 &= ~UCB0TXIFG;                  // Clear USCI_B0 TX int flag
        //__bic_SR_register_on_exit(CPUOFF);        // Exit LPM0
        }
  }
}*/

void I2C_dat(void) __interrupt[USCIAB0TX_VECTOR]{
  unsigned char flags = UC0IFG & (UC0IE);
  unsigned char I2Cstatus = UCB0STAT;

  if(I2Cstate == I2C_TX) {
        // Master Receiving from slave
        if(flags & UCB0RXIFG){                      // Check if data was received
        I2Cdat[I2Cidx] = UCB0RXBUF;                   // Receive data
        I2Cidx++;                                     // Increment index
        if(I2C_len == (I2Cidx+1)) {                   // Check if this is the 2nd to last byte
        UCB0CTL1 |= UCTXSTP;                          // Generate stop condition
        }                                             // One more interrupt to go
        if(I2C_len >= I2Cidx) {                       // Check if this was the last byte
        I2Cstate = I2C_COMPLETE;                      // Set return value
        LPM4_EXIT;                                    // Exit low power mode
        }
        }

        // Master Sending to Slave
        if(flags & UCB0TXIFG) {                       // Check if data needs to be transmitted
        if(I2C_len > I2Cidx){                         // Check if there are more bytes
        UCB0TXBUF = I2Cdat[I2Cidx];                   // Receive data
        I2Cidx++;                                     // Increment index
        } else {
        UCB0CTL1|=UCTXSTP;                            // Generate stop condition
        I2Cstate=I2C_COMPLETE;                        // Return value
        UC0IFG &= ~UCB0TXIFG;                         // Clear interrupt flag
        LPM4_EXIT;                                    // Exit low power mode
        }
        }
  }                                                   // End I2C_TX Handling
  if(I2Cstate == I2C_IDLE) {
  if(I2Cstatus & UCSTTIFG) {                          // Start condition received
        I2Cstate = I2C_RX;                            // Note that peripheral has begun receiving
        I2C_len = 0;                                  // Start writing at the beginning of the buffer
        UCB0I2CIE |= UCSTPIE;
  }
  }
  if(I2Cstate == I2C_RX) {
        // Slave Receiving from master
        I2C_RX_buf[I2C_len++] = UCB0RXBUF;            // Get RX data
        //data_in = 1;
        if(I2Cstatus & UCSTPIFG) {                            // Stop condition received
        //UC0IFG &= ~UCSTPIFG;                                // Clear interrupt flag
        I2Cstate = I2C_IDLE;                          // Note that receiving is complete
        I2C_RX_dat = 1;                               // Indicate that writing to the RX buffer is complete
        UCB0I2CIE &= ~UCSTPIE;
        }
  }

}

void I2C_state(void) __interrupt[USCIAB0RX_VECTOR]{
  unsigned char flags = UC0IFG & (UC0IE);
  unsigned char I2Cstatus = UCB0STAT;
  //clear state bits that were read
  UCB0STAT&=~I2Cstatus;

  if(I2Cstatus & UCNACKIFG) {                         // Not-Acknowledge received
        UCB0CTL1 |= UCTXSTP;                          // Generate stop condition
        I2Cstate = I2C_ERROR;                         // set ERROR return value
        LPM4_EXIT;                                    // Exit low power mode
  }
  // ggg These belong in the tx vector, already above. Learn why.
  if(flags & UCSTPIFG) {                      // Stop condition received
        I2Cstate = I2C_IDLE;                          // Note that receiving is complete
        I2C_RX_dat = 1;                               // Indicate that writing to the RX buffer is complete
  }
  if(I2Cstatus & UCSTTIFG) {                          // Start condition received
```

```
        I2Cstate = I2C_RX;                          // Note that peripheral has begun receiving
  }
  if(I2Cstatus & UCALIFG) {
        // Arbitration Lost, not used in master mode
        // ggg: Need conversation on this topic with people who know the peripheral better
  }
  if(I2Cstate == I2C_RX) {
        // Slave Receiving from master
        I2C_RX_buf[I2C_len++] = UCB0RXBUF;          // Get RX data
        //data_in = 1;
        if(I2Cstatus & UCSTPIFG) {                          // Stop condition received
        //UC0IFG &= ~UCSTPIFG;                              // Clear interrupt flag
        I2Cstate = I2C_IDLE;                        // Note that receiving is complete
        I2C_RX_dat = 1;                             // Indicate that writing to the RX buffer is complete
        }
  }
}

void USCI0RX_ISR(void) __interrupt[USCIAB1RX_VECTOR]
{
  while (!(IFG2&UCA1TXIFG));                // USCI_A0 TX buffer ready?

  if(RxBuf.len == 0) {
        if(RxBuf.p2 == RxBuf.start_sentinel[0] && RxBuf.p1 == RxBuf.start_sentinel[1]) {
        RxBuf.p1 = RxBuf.p2 = 0;
        RxBuf.len = UCA1RXBUF;
        RxBuf.in = 0;
        } else {
        RxBuf.p2 = RxBuf.p1;
        RxBuf.p1 = UCA1RXBUF;
        }
  } else {
        RxBuf.buf[RxBuf.in++] = UCA1RXBUF;
        if(RxBuf.in == RxBuf.len) {
        RxBuf.term_str = RxBuf.len;
        RxBuf.len = 0;
        RxBuf.in = 0;
        }
  }

  //RxBuf.buf[RxBuf.in] = UCA1RXBUF;                // Pass incoming UART traffic to the input buffer
  //if(RxBuf.buf[RxBuf.in] == STRING_TERMINATOR) RxBuf.term_str = 1;  // Note if a string has been terminated
  //RxBuf.in = (RxBuf.in+1)%RX_SIZE;                // Increment the buffer pointer
}

void USCI0TX_ISR(void) __interrupt[USCIAB1TX_VECTOR]
{
  unsigned int t;
  if(IFG2&UCA1TXIFG) {
        if(TxBuf.out == TxBuf.in) {
        TxBuf.done=1;
        //UC1IE &= ~UCA1TXIE;
        UC1IFG &= ~UCA1TXIFG;
        } else {
        UCA1TXBUF = TxBuf.buf[TxBuf.out];
        TxBuf.out = (TxBuf.out+1)%TX_SIZE;
        }
  }
}
```

```
// arc_uart_uca1.h for the unified test harness firmware.

// This code targets the MSP430F2618 device specifically.

// Requires the Rowley Crossworks MSP430 compiler and related libraries

// Gregg Christopher, 2011

//UART stuff
#define RX_SIZE   64
#define TX_SIZE   128
//#define STRING_TERMINATOR '\r'


//TX buffer structure
struct Tx{
  unsigned int in;
  volatile unsigned int out;
  char done;
  char buf[TX_SIZE];
};

//RX buffer structure
struct Rx{
  volatile unsigned int in;  // Index of the next open location in the buffer.
  unsigned int out;           // Index of the next byte to be read in. (Buffer is empty when in == out)
  char term_str;              // Flag indicating that a null string terminator resides somewhere in the unread
buffer.
  char start_sentinel[10];
  char p1, p2;
  unsigned int len;
  char buf[RX_SIZE];
};

extern struct Tx TxBuf;
extern struct Rx RxBuf;
extern char uart_in[RX_SIZE];
extern unsigned int uart_len;

void init_uart_uca1();
int arc_getc(void);
int txchar(unsigned char c);
int __putchar(int ch);
```

```
// arc_uart_uca1.c for the unified test harness firmware.

// This code targets the MSP430F2618 device specifically.

// Requires the Rowley Crossworks MSP430 compiler and related libraries

// Gregg Christopher, 2011

#include <msp430x26x.h>
#include <stdio.h>

#include "arc_uart_uca1.h"

struct Tx TxBuf;
struct Rx RxBuf;
char uart_in[RX_SIZE];
```

```
unsigned int uart_len;

void init_uart_uca1() {
  memset(&TxBuf,0,sizeof(TxBuf));
  memset(&RxBuf,0,sizeof(RxBuf));
  TxBuf.in = TxBuf.out = 0;
  TxBuf.done = 0;
  RxBuf.in = RxBuf.out = 0;
  RxBuf.term_str = 0;

  // Set up UCA1 for 9600 baud UART
  UCA1CTL1=UCSWRST;
  P3SEL |= BIT6|BIT7;                        // P3.6,7 = USCI_A1 TXD/RXD
  UCA1CTL0 = 0;
  UCA1CTL1 |= UCSSEL_1;                      // SMCLK
  UCA1BR0 = 3;                               // 32,768 => 9600
  UCA1BR1 = 0;                               // 32,768 => 115200
  UCA1MCTL = UCBRS_3;                        // Modulation UCBRSx = 3
  //take UCA1 out of reset mode
  UCA1CTL1 &= ~UCSWRST;                      // **Initialize USCI state machine**
  UC1IE|=UCA1TXIE|UCA1RXIE;                  // Enable USCI_A1 RX interrupt

  strcpy(RxBuf.start_sentinel, "!#");
  RxBuf.len = 0;
}



//queue byte to get transmitted
int txchar(unsigned char c){
  unsigned int t;
  int res=c;
  unsigned short state;
  //disable interrupt
  state=__disable_interrupt();
  //check if transmitting
  if(TxBuf.done) {
        //bypass queue for first byte if not transmitting
        UCA1TXBUF=c;
        //clear done flag
        TxBuf.done=0;
        //UC1IE |= UCA1TXIE;
  //queue byte
  } else {
        //get next input index
        t=(TxBuf.in+1)%TX_SIZE;
        //check if next input index equals output index
        if(t!=TxBuf.out) {
        //not equal, room in buffer
        //put char in buffer
        TxBuf.buf[TxBuf.in]=c;
        //set next input index
        TxBuf.in=t;
        } else {
        //buffer full
        res=EOF;
        }
  }
  //enable interrupt
  __set_interrupt(state);
  //return result
  return res;
}

//get byte from buffer return if buffer is empty
int arc_getc(void) {
  int c;
  unsigned int t;
  unsigned short state;
  //check for bytes
  if(RxBuf.in==RxBuf.out){
        RxBuf.term_str = 0;
        //no bytes return EOF
        return EOF;
  } else {
        //disable interrupt while byte is retrieved
        state=__disable_interrupt();
        //get output index
        t=RxBuf.out;
        //get char from buffer
        c=(unsigned char)(RxBuf.buf[t++]);
        //set new output index
        RxBuf.out=(t)%RX_SIZE;
        //re enable interrupt
        __set_interrupt(state);
  }
  //return result
  return c;
}

//make printf send over UART
int __putchar(int ch) {
  return txchar(ch);
}
```

```
// Highly simplified test harness for firmware proofing.
// Requires the CSerial library and FTDI Ft245 libraries and drivers.

// Gregg Christopher, 2011

#include "stdafx.h"
#include "SerialPort.h"
#include "conio.h"
#include <windows.h>
#include "ftd2xx.h"
#include "FTChipID.h"

#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <vector>
using std::vector;
#include <string>
using std::string;
#include <cstdlib>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

#include <iostream>
```

```
using namespace std;

FT_HANDLE ftHandle;
FT_STATUS ftStatus;

const int WORK_BUF_SIZE = 30;
const int SAFETY_SWITCH_TIME = 100;

int ft245_write(unsigned char c) {
  DWORD written;

  char txbuf[10];
  txbuf[0] = c;
  ftStatus = FT_Write(ftHandle, txbuf, 1, &written);
  if (ftStatus == FT_OK) {
        //printf("FT245 Written:  %d %02X\n", written, (unsigned char)txbuf[0]);
  } else {
        printf("FT245 NOT OK %X\n", written);
        exit(0);
  }
  return 0;
}

class dev_board {
public:
  unsigned char led_val;
  unsigned char ft_addr;
  unsigned char i2c_addr;
  unsigned char work_buf[WORK_BUF_SIZE];

  dev_board() {
        led_val = 0;
  }

  void write_LED(CSerialPort &port, unsigned char l) {
        Sleep(SAFETY_SWITCH_TIME);
        ft245_write(ft_addr);
        Sleep(SAFETY_SWITCH_TIME);

        led_val = l;
        unsigned char command[10];
        command[0] = 0x21;
        command[1] = 0x23;
        command[2] = 0x02;
        command[3] = 0x00;
        command[4] = l;

        int x = port.Write(command, 5);
        printf("LED set written: %d: ", x);
        for(int ii = 0; ii < 5; ii++) printf("%02X ", command[ii]);
        printf("\n");
  }

  int check_LED(CSerialPort &port) {
        Sleep(SAFETY_SWITCH_TIME);
        ft245_write(ft_addr);
        Sleep(SAFETY_SWITCH_TIME);

        char command[100];
        char response[100];
        char exp_response[100];
```

```
      command[0] = 0x21;
      command[1] = 0x23;
      command[2] = 0x01;
      command[3] = 0x01;

      sprintf(exp_response, "LED State: %02X", led_val);

      int x = port.Write(command, 4);

      printf("written: %d\n", x);

      Sleep(100);

      if(port.BytesWaiting() > 0) {
      x = port.Read(response, port.BytesWaiting());
      response[x-1] = '\0';

      printf("%d returned: %s\n", ft_addr, response);
      if(strcmp(response, exp_response) == 0) {
      printf("SUCCESS!\n");
      return 0;
      } else {
      printf("FAIL! Expected %02X\n", led_val);
      getch();
      return 1;
      }
      } else {
      printf("no response received.\n");
      return -1;
      }
}

int send_led_change(CSerialPort &port, dev_board &tb, unsigned char l) {
      Sleep(SAFETY_SWITCH_TIME);
      ft245_write(ft_addr);
      Sleep(SAFETY_SWITCH_TIME);

      tb.led_val = l;
      unsigned char command[10];
      command[0] = 0x21;
      command[1] = 0x23;
      command[2] = 0x03;
      command[3] = 0x02;
      command[4] = tb.i2c_addr;
      command[5] = l;


      int x = port.Write(command, 6);
      printf("To %d LED change command (to %d) written: %d: ", ft_addr, tb.ft_addr, x);
      for(int ii = 0; ii < 6; ii++) printf("%02X ", command[ii]);
      printf("\n");

      Sleep(SAFETY_SWITCH_TIME);

      if(port.BytesWaiting() > 0) {
      printf("Send LED I2C FAILURE!\n");
      return -1;
      }

      return 0;
}
```

```
};



int ftcontrol() {
    char Version[100];
    unsigned long NumDevices = 0, LocID = 0, ChipID = 0;
    char SerialNumber[256], Description[256], ErrorMessage[256];
    FTID_STATUS dStatus;
    DWORD iNumDevs;
  char c;
  unsigned char mask = 0xFF;

  printf("Initializing FT245 I/O\n\n");

    FTID_GetDllVersion(Version, 100);

    printf("Dll Version = %s\n\n", Version);

    dStatus = FTID_GetNumDevices(&NumDevices);

    if((dStatus == FTID_SUCCESS) && NumDevices) {

         printf("Number of 232R devices = %ld\n\n", NumDevices);

         for(int i = 0; i < (int)NumDevices; i++) {

                printf("Device %d\n", i);

                dStatus = FTID_GetDeviceSerialNumber(i, SerialNumber, 256);
                if(dStatus == FTID_SUCCESS) {
                       printf("\tSerial Number: %s\n", SerialNumber);
                }

                dStatus = FTID_GetDeviceDescription(i, Description, 256);
                if(dStatus == FTID_SUCCESS) {
                       printf("\tDescription: %s\n", Description);
                }

                dStatus = FTID_GetDeviceLocationID(i, &LocID);
                if(dStatus == FTID_SUCCESS) {
                       printf("\tLocation ID: 0x%08lX\n", LocID);
                }

                dStatus = FTID_GetDeviceChipID(i, &ChipID);
                if(dStatus == FTID_SUCCESS) {
                       printf("\tChip ID: 0x%08lX\n", ChipID);
                }
                printf("\n");
         }
    }

    if(dStatus != FTID_SUCCESS) {
         FTID_GetErrorCodeString("EN", dStatus, ErrorMessage, 256);
         printf(ErrorMessage);
    }

    printf("Get Chip IDs using handle\n");

    FT_DEVICE_LIST_INFO_NODE *devInfo;
    //
    // create the device information list
    //
    ftStatus = FT_CreateDeviceInfoList(&iNumDevs);
    if (ftStatus == FT_OK) {
       printf("%d FTDI devices connected\n",iNumDevs);
    }

    //
    // allocate storage for list based on numDevs
    //
    devInfo = (FT_DEVICE_LIST_INFO_NODE*)malloc(sizeof(FT_DEVICE_LIST_INFO_NODE)*iNumDevs);

    //
    // get the device information list
    //
    ftStatus = FT_GetDeviceInfoList(devInfo, &iNumDevs);
    if (ftStatus == FT_OK) {
        for (int i = 0; i < iNumDevs; i++) {
               if(devInfo[i].Type == FT_DEVICE_232R) {
                       FT_Open(i, &ftHandle);
                       dStatus = FTID_GetChipIDFromHandle(ftHandle, &ChipID);
                       if(dStatus == FTID_SUCCESS) {
                               printf("\tChip ID: 0x%08lX\n", ChipID);
                       }

       printf("Setting Asynchronous Bit Bang Mode for I/O mask %X\n", mask);
       ftStatus = FT_SetBitMode(ftHandle, 0xFF, 1);
       if (ftStatus != FT_OK) {
       printf("Error in SetBitMode\n");
       }

       printf("Ready for FT245 communication\n");
               }
        }
    }
    return 0;
}


// The one and only application object

CWinApp theApp;

using namespace std;

void openport(CSerialPort &port) {
  try {
       port.Open(20, 9600, CSerialPort::NoParity, 8,
       CSerialPort::OneStopBit,
       CSerialPort::XonXoffFlowControl);
       //CSerialPort::XonXoffFlowControl);

       //char sBuf[] = { 64, 0 }; //"This should appear on the serial port";
       //port.Write(sBuf, strlen(sBuf));

  }
  catch (CSerialException* pEx)
  {
       cout << "Exception caught\n";
       TRACE(_T("Handle Exception, Message:%s\n"),
                      pEx->GetErrorMessage());
        pEx->Delete();
        exit(0);
  }
  //port.Set0WriteTimeout();
  //port.Set0ReadTimeout();
}

void closeport(CSerialPort &port) {
  port.Flush();

  port.Close();
}



void full_reset(CSerialPort &port) {
  char response[100];
  HANDLE hPort = port.Detach();
  int x;


  ft245_write(0x07); // Turn off the main system power
  Sleep(750);
  ft245_write(0x00); // Turn the main power back on
  Sleep(750);

  port.Attach(hPort);

  if(port.BytesWaiting() > 0) {
        //printf(".\n");
        //printf(".\n");
        x = port.Read(response, port.BytesWaiting());
        response[x-1] = '\0';

        printf("stripped %d characters from the serial stream: %s\n", x, response);
  }
}

int random_LED_test(dev_board db[3], CSerialPort &port) {
  unsigned int rboard = rand()%3;
  unsigned char randlval = rand()%0x7F;
  unsigned int rw = rand()%2;

  switch(rw) {
  case 0:
        return db[rboard].check_LED(port);
        break;
  case 1:
        db[rboard].write_LED(port, randlval);
        break;
  case 2:
        unsigned int tb = rand()%2 + 1;
        return db[rboard].send_led_change(port, db[(rboard + tb)%3], randlval);
        break;
  }

  return 0;
}

int test_sequence(CSerialPort &port) {
  dev_board db[3];
  while(1) {
        db[0].i2c_addr = 0x03;
        db[0].ft_addr = 0x00;
        db[0].write_LED(port, 0x08);
        //getch();
        db[1].i2c_addr = 0x02;
        db[1].ft_addr = 0x01;
        db[1].write_LED(port, 0x08);
        //getch();
        db[2].i2c_addr = 0x01;
        db[2].ft_addr = 0x02;
        db[2].write_LED(port, 0x08);

        for(int ii = 0; ii < 100; ii++) random_LED_test(db, port);
        full_reset(port);
  }

  //db[2].send_led_change(port, db[0], 0x33);

  return 0;
}

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
  ftcontrol();
  //ft245_write(0x00); // Turn the main power on, address board multiplexed at 00

  CSerialPort port;
  openport(port);
  full_reset(port);
    int nRetCode = 0;

  srand(time(0));

  COMMTIMEOUTS timeouts;
  port.GetTimeouts(timeouts);
  /*printf("%d\n", timeouts.ReadIntervalTimeout);
  printf("%d\n", timeouts.ReadTotalTimeoutConstant);
  printf("%d\n", timeouts.ReadTotalTimeoutMultiplier);
  printf("%d\n", timeouts.WriteTotalTimeoutConstant);
  printf("%d\n", timeouts.WriteTotalTimeoutMultiplier);*/

  timeouts.ReadIntervalTimeout = 1;
  timeouts.ReadTotalTimeoutConstant = 100;
  timeouts.ReadTotalTimeoutMultiplier = 1;

  port.SetTimeouts(timeouts);

/******************************************/
  string cmd, args;

    unsigned char lCommand[1] = {0},
         rCommand[1] = {0};

  printf("I think I have the serial port.\n");
  char command[100];
  char response[100];
  command[0] = 0x21;
  command[1] = 0x23;
  command[2] = 0x01;
  command[3] = 0x01;
  command[4] = 0xBB;
  command[5] = 0;
  test_sequence(port);

  closeport(port);

  printf("\nClosing FT245 Device Handle\n");
  FT_Close(ftHandle);

    return nRetCode;

}
```