Distributed Algorithms for Self–Disassembly in Modular Robots

by

Kyle W Gilpin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by .... [illegible] ....
Daniela Rus
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Distributed Algorithms for Self–Disassembly in Modular Robots

by

Kyle W Gilpin

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We developed a modular robotic system that behaves as programmable matter. Specifically, we designed, implemented, and tested a collection of robots that, starting from an amorphous arrangement, can be assembled into arbitrary shapes and then commanded to self–disassemble in an organized manner.

The 28 modules in the system were implemented as 1.77-inch autonomous cubes that were able to connect to and communicate with their immediate neighbors. Two cooperating microprocessors controlled the modules' magnetic connection mechanisms and infrared communication interfaces.

We developed algorithms for the distributed communication and control of the system which allowed the modules to perform localization and distribute shape information in an efficient manner. When assembled into a structure, the modules formed a system which could be virtually sculpted using a computer interface which we also designed. By employing the sculpting process, we were able to accurately control the final shape assumed by the structure. Unnecessary modules disconnected from the structure and fell away.

The results of close to 200 experiments showed the that the algorithms operated as expected and were able to successfully control the distributed system. We were able to quickly form one, two, and three dimensional structures.

Thesis Supervisor: Daniela Rus
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I owe a great deal of gratitude to Daniela Rus for her support throughout the last year. This project never would have come to fruition without the help of my colleagues in the Distributed Robotics Lab. In particular, I am indebted to Iuliu Vasilescu who provided the impetus for the algorithms that I developed as well as countless hours soldering and assembling the modules. Keith Kotay also deserves special recognition for the time he spent discussing and prototyping the early connector mechanisms. Finally, I would like to thank my parents, Linda and Bill, who both read this thesis and provided excellent feedback.

This thesis was made possible by support from NSF, Air Force MURI, Intel, and a generous donation of connectors by Magswitch Technology, Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Self–organization is the ability of distributed systems to form functional groups and structures in accordance with their environment and task. Self-organizing systems serve as tools which can be used to simulate the interaction of simple behaviors that, when aggregated, display surprisingly complicated phenomena. The systems are highly relevant and potential laden due to their similarities with numerous biological systems such as those of social insects and nervous systems. Consequently, artificial self–organizing systems can assist scientists in gaining a better understanding of the natural world and extracting information which can be utilized to better society.

This thesis looks at self–organizing systems which are able to self–disassemble. That is, they are able to transform from regular homogeneous structures into more intricate and interesting configurations by removing some system components. The basic idea behind the self–disassembling modules we developed can be thought of as sculpting a block of marble. A prearranged structure of individual modules will transform into a different structure by breaking apart in an orderly fashion. Much like a sculptor would remove the extra stone from a block of marble to reveal a statue, our self–disassembling system eliminates unnecessary modules to form a more interesting structure.

A collection of millions of modules, if each were small enough, could form a completely malleable building material that could solidify and then disassemble on command. The applications for such a material are numerous. For example, the material could be applied to help heal severely broken bones that would otherwise require the use of permanent steel plates or pins. Thousands of tiny modules could be packed around the break by a surgeon, or they could be injected into the body and drawn to the site of the break with a magnetic field. Once satisfied that enough modules surrounded the break, the modules could be instructed to solidify. The solid structure of modules would both immobilize the broken bone and provide a scaffolding on which new bone material could regenerate. The modules would also be able to sense the bone's regrowth and slowly disassemble in the places they were no longer needed. The modules could also provide feedback detailing how well the bone was mending. The disassembly process would continue until the two pieces of bone were securely re–attached. As the modules disassembled from the structure around the site of the break, they could be absorbed into the blood stream and passed out of the body in a harmless manner. While this level of sophistication is still far off, the system we developed serves as a means to study the algorithms the would be necessary for such a use.

In the particular system we developed, many small modules are initially arranged by hand to form a super–structure, for example, the 27–module cube shown in Figure 1-1(a). Each individual module is physically linked to up to six neighbors. After the super–structure is assembled, we model a desired final configuration of the system using a desktop computer. This model is transmitted to the structure where it is distributed. Once the super–structure receives information about the desired final configuration of the modules, the appropriate modules break their connections with their neighbors and disconnect from the structure. The unnecessary modules continue to separate until the desired final configuration, pictured in Figure 1-1(b), is achieved.

In the process of implementing our system, we faced numerous hardware design, control, and coordination challenges. When dealing with large self–organizing systems, there are many unsolved problems. We do not yet understand the most efficient way to communicate with all of the modules in a system, nor do we understand how to most effectively deal with malfunctioning units. Additional experimentation is

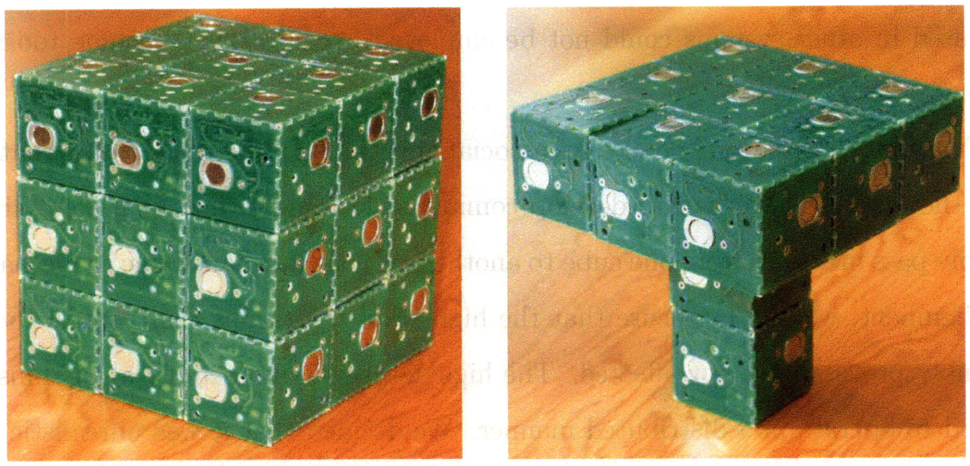


Figure 1-1: A self–disassembling system can transform from an initial uniform assembly of identical modules, (a), into a more interesting and functional assembly in (b).

needed to investigate the emerging properties of distributed, self–organizing systems. Researchers have yet to develop the communication and control algorithms necessary to successfully employ large distributed robotic systems for the study of natural phenomena. Additionally, studying self–organization presents challenges because it requires vast systems and robust algorithms capable of adaptive, task-driven distributed coordination and control. Methods of fabricating, programming, and debugging such systems do not exist.

Other groups have focused on self–assembling, self–reconfiguring, and self–replicating systems, but little research has been devoted to the specifics of self–disassembly. To study self–disassembling systems, we first had to design appropriate hardware that was as small as possible. In the design process, we imposed the additional constraint that the modules be completely autonomous—each module should have its own sensing, processing, actuation, and power supply capabilities. This choice differentiates our system from several other modular robotic systems that are also under development. Rarely do modules contain their own power supplies, and many are not able to perform any significant amount of processing. Because we wanted each module to contain its own power source, the standard mechanical and electromagnetic connectors used in other systems could not be employed because they consume too much power.

There were also several challenges associated with designing the software that controls the modules. We had to develop communication algorithms that were able to reliably pass messages from one cube to another. When these failed due to mechanical misalignment, we had to ensure that the high–level algorithms that control the disassembly process were not affected. The high–level self–disassembly algorithms also needed to operate in a distributed manner. We decided that it was unrealistic for a module to have any global information about the system. Individual modules should only know about the existence and state of their immediate neighbors. Even though the modules did not have information about the system as a whole, they needed to localize with respect to one another and be able to route shape distribution messages throughout the entire structure. Despite these hardware and software challenges, we developed a successful modular self–disassembling system that can serve as a platform for further study.

This thesis is divided into eight chapters. After this first chapter, the second addresses some related work. Chapter 3 then begins to discuss the specifics of the modules we developed by presenting their hardware platform. Following the discussion of the hardware, Chapter 4 explains the important aspects of the low–level software. This software interfaces with the hardware and allows the top–level self–disassembly algorithms to control the modules in a more abstract manner. Chapter 5 presents the basic self–disassembly algorithms, proves that they operate correctly, and analyzes their running times. To actually implement the algorithms on the hardware, we had to incorporate several refinements. These changes and the additional details explaining how the algorithms were implemented on the hardware are discussed in Chapter 6. Following the discussion of how the algorithms were implemented, Chapter 7 presents the results of close to 200 experiments that we performed to test the system's abilities and verify that the algorithms operated correctly. Finally, Chapter 8 draws several conclusions about the system and presents directions for future work.

Chapter 2

Related Work

Our work draws on prior and ongoing research in modular and distributed robotics. Specifically, we have built upon work in the fields of self–assembling and self–reconfiguring robotics to develop a new type of system that behaves as programmable matter. To date, only a limited amount of research has been devoted to self–disassembling systems. Most research has focused on self–assembly and self–reconfiguration in robotic systems. Typically, these systems construct and rearrange themselves in order to provide locomotion or to manipulate objects. The topic of this thesis, self–disassembly, can be viewed as a specialized case of self–reconfiguration, so the research of self–reconfiguring robotic systems is still relevant. In what follows, we summarize several modular systems which have already been developed.

Yoshida presents an intriguing self–reconfiguring system which uses shape memory alloy (SMA) springs to connect neighboring modules [16]. He states that he chose SMA based actuators because they maintain a favorable power to weight ratio when miniaturized. In fact, his smallest modules fit inside a 2 centimeter cube and weigh only 15 grams. Yoshida's system, which is currently confined to two dimensions, consists of square modules that include two male connectors, (on opposing vertices), and two female connectors, (on the opposite set of opposing vertices). The male connectors can swing through a 180 degree arc and bond with their female counterparts. This design allows one module to traverse around the exterior of a number of other connected modules. One drawback to Yoshida's system is the amount of power it consumes. While the system's connectors do not dissipate power in their static state, actuation requires 1 Amp. Consequently, the modules do not contain their own power supplies. Furthermore, all of the processing required for the system's motion planning is performed on a separate computer after which control commands are transmitted to the individual modules [16].

In a separate paper, Yoshida devotes additional space to describing a three-dimensional adaptation of his system [15]. Additionally, he presents a recursive method for describing the structures formed by modular robots. This recursive representation uses multiple layers of abstraction in order to separate the low-level details of the structure from the structure's general shape. For example, eight modules could be arranged to form a cube which is then viewed as a single, indivisible node by higher level descriptions of the system as a whole. Yoshida states that using such a recursive representation enables one to describe sizable, complicated shapes which would be impossible to characterize otherwise.

In [8], Rus describes a system of "crystal" modules which can expand and contract two of their dimensions by a factor of two. The crystal modules are composed of four movable faces arranged in a square. Two of these faces contain active connectors and two contain passive connectors. The active connectors can mate with passive connectors. By selectively latching and unlatching from their neighbors, a collection of crystals is able to arbitrarily modify its structure in two dimensions. In total, each crystal module has only three degrees of freedom: one to expand the faces and two to control the state of the latches on the active faces. Rus presents detailed algorithms to accompany the crystal hardware. These algorithms prove that composite crystal structures can assume arbitrary configurations and that any individual module can relocate anywhere in the structure.

A novel system for self–assembly and reconfiguration is presented by White [13] which uses fluid flow to bind individual modules together. In particular, the modules have gated inlets which, when opened, draw in fluid to exert a suction force on the neighboring modules. Unfortunately, the modules only operate in fluids more viscous than air, such as oil and water. Additionally, the modules have limited processing power and depend on an external pump to move the fluid and provide the suction force. According to the authors, the greatest advantage of their system is its ability to function correctly even if the modules are shrunk to the microscale [13], a property absent in most self–reconfiguring systems.

The CHOBIE robot developed by Koseki [6] is also unique in its mechanical design. The modules in the CHOBIE system, which are rectangular, are able to locomote by sliding in two planes relative to one another. A module cannot only slide horizontally across the top of another module, but vertically down a module's side as well. This ability allows one robot to climb up and over another. While the CHOBIE robots contain only basic processing power, they are self–contained and able to operate untethered. The system is still confined to two dimensions.

Mytilinaios presents an especially clever system of modular robotics in [7] which is able to self–replicate. The system is composed of cubes that have a rotational degree of freedom in line with the diagonal from one corner of the cubic modules to the corner farthest away. The system uses electromagnets to connect neighboring modules and two electrical contacts to share power and communication signals. Mytilinaios uses the system to investigate different initial configurations that, when given a supply for spare modules, can replicate themselves. As with other systems, the disadvantage of Mytilinaios' system is the apparent inability of the modules to perform high–level computation and the fact the the modules must be anchored to a special base which supplies the necessary power.

While most of the above research efforts focused on modular systems which are completely homogeneous, systems with multiple types of modules also exist. Yim presents the PolyBot system, which relies on two types of modules: segments and nodes [14]. The nodes are passive cubes that have connectors on all six faces. In contrast, the segments consist of only two connectors which are able to move relative to one another. Specifically, a rotational joint driven by a motor separates the two connectors. The PolyBot is a chain type reconfigurable modular robot. In other words, chains of modules connect in order to form structures such as loops, legs, or tendrils. Yim's system has the ability to achieve locomotion through any number of configurations. For example, a number of segment and node modules can form a loop that rolls across smooth terrain. If confronted with more challenging terrain, the robot can reconfigure itself as a multi-legged walking robot. The PolyBot is also unique because its connectors are not of a specific gender. Using a system of pins and latches, any connector face can mate with any other connector face.

Castano developed another chain–type modular robotic system called Conro that is detailed in [1]. The Conro robot is able to assume forms that resemble snakes and multi–jointed walkers. The Conro system is completely self–contained. Each module has a microprocessor, a power supply, sensing capabilities, communication interfaces, and actuation mechanisms that allow modules to both flex and join together. In fact, each individual module can be considered a robot by itself.

Finally, Kamimura et. al. have developed the M-TRAN modular robotic system [5]. The M-TRAN system is able to use numerous cooperating modules to, among other tasks, achieve a modular walking robot. The individual modules contain two degrees of freedom. In addition, they contain processing and battery power. The most interesting aspect of the M-TRAN system is the way in which it is controlled using a set of interconnected oscillators that are all out of phase with respect to one another. By optimizing the phase relationships between the oscillators, they can be used to drive the modules' motors in a coordinated fashion that leads to forward locomotion. The M-TRAN robot relies on evolutionary algorithms to perform this optimization process. The optimization process can occur in simulation before being implemented in hardware, or it can be performed in real–time to allow the robot to adapt to a changing environment [5].

The modular system presented in this thesis is different from all of the above systems for several reasons. First, the system presented here focuses on the process of self–disassembly. Unlike most existing systems which must find creative ways to compress or hide extra modules, the system presented here disposes of extra modules. Second, the system operates in three dimensions. The modules have no preferential direction and can be assembled into arbitrary structures. Third, the modules we developed contain significant amounts of processing and battery power. If necessary, this allows the modules to play a significant role in the process of shape planning and distribution. The modules' autonomous nature also enables any structure they compose to continue to perform sensing and computation even after the self–disassembly process is complete. Finally, this thesis explores several novel algorithms that communicate structural data about the system to a user interface and information about the structure's desired configuration back to the modules.

Chapter 3

Hardware

The self-disassembling system that we developed is composed of autonomous modules that cooperate to form more complex structures. A module, such as the one shown in Figure 3-1, is a cube that is identical to all other modules in the system. All modules contain the resources necessary to be totally self–sufficient: processing capabilities, actuation mechanisms, communication interfaces, and power supplies. This chapter will examine the above named features.

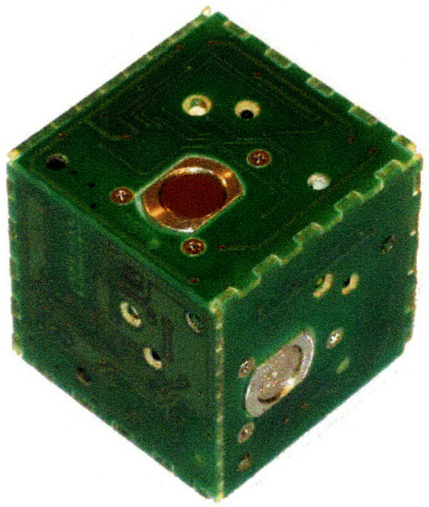


Figure 3-1: Each module in the system is a cube which measues 1.77 inches on each side and weighs 4.5oz. Each module is completely autonomous and can operate for several hours under its own power.

The modules are built from six distinct printed circuit boards that interlock to form a rigid structure. When completely assembled, each cubic module is 1.77 inches on a side and weighs 4.5 oz. As shown by an open module in Figure 3-2, all electronic components are surface mounted on the top side of the boards so that when assembled into cubes, all components reside on the inside. The only pieces of the system mounted externally are three steel plates that form half of the magnetic connection mechanism, presented in detail below.

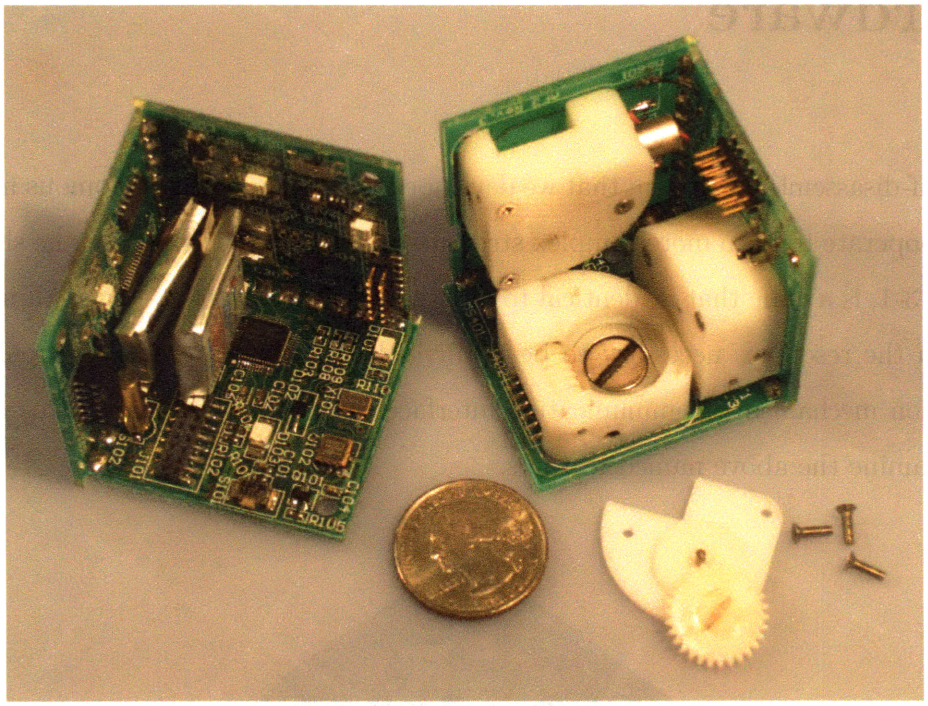


Figure 3-2: An open module shows all of its major components. Each contains two microprocessors, connection mechanisms, infrared emitters and detectors, an accelerometer, a tilt switch, and batteries. Each cube is totally self–sufficient.

This chapter details the major components of each module. To begin, Section 3.1 describes the processing capabilities of each module. It examines the major hardware resources of the two microprocessors in each module. Section 3.2 explains the magnetic connection mechanism that the cubes employ to latch together. Then, Section 3.3 describes the infrared communication system that the cubes use to interface with their neighbors. Section 3.4 characterizes the ability of the modules to detect their orientation with respect to gravity by using an accelerometer and a tilt switch. Finally, Section 3.5 outlines the power supply system.

3.1 Processors

Each module contains two microprocessors that perform different tasks. The primary microprocessor is a 32-bit ARM processor produced by Philips. It is responsible for all of the high–level disassembly algorithms. The second processor is an 8-bit programmable system on a chip (PSoC) that is manufactured by Cypress Microsystems. The PSoC handles the low–level functions that would otherwise occupy the ARM. The two systems communicate using the I^2C protocol [10].

3.1.1 Primary ARM Processor

The ARM processor executes all of the high–level algorithms that give the system the ability to self–disassemble. The processor, a Philips LPC2106, boasts a 32-bit wide data path, 128KB of reprogrammable flash memory for program storage, and 64KB of RAM for temporary data storage. We operate the processor at 11.0592MHz to minimize the ARM's power consumption, but the chip is capable of running at up to 60MHz [11].

A block diagram of the major hardware components of the ARM processor is displayed in Figure 3-3. Most of the functionality of the ARM chip is implemented in software and will be discussed in later chapters. Starting at the upper right portion of the diagram in Figure 3-3, the first hardware component worth noting is Timer 0. This timer interfaces with the accelerometer that is detailed in Section 3.4 and converts its two pulse width modified (PWM) output signals into acceleration estimates.

The next piece of hardware, Timer 1, operates as an interrupt source. It sends interrupt events to the software at fixed intervals. These interrupts serve two purposes. First, the interrupts, along with the PWM generator shown in the diagram, change the mode of the user LED so that it can display complex patterns that communicate the state of the module to an observer. Additionally, the interrupts prompt

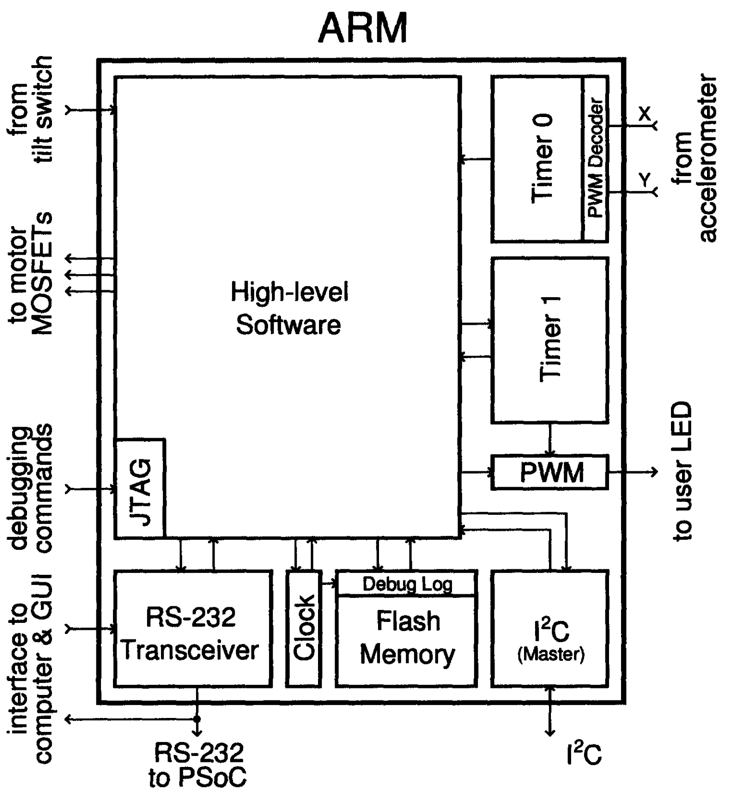


Figure 3-3: The ARM processor contains numerous hardware resources which are crucial to the correct operation of the system. These resources are all controlled by the software which executes on the ARM.

the RS-232 transceiver, shown in the bottom left of Figure 3-3, to automatically re-transmit any messages that it has queued. The RS-232 transceiver is used by the modules to communicate with their neighbors. Any message destined for a neighbor is transmitted on this interface to the PSoC. The PSoC then routes the message to the appropriate neighbor. Details of the message format are covered in Chapter 4.

The RS-232 transceiver also communicates with a desktop computer over what we term the up-link interface. The up-link interface allows one module to serve as a bridge between a desktop computer and the distributed system. By connecting the system to a desktop computer, the user can easily visualize and sculpt the system using a graphical user interface (GUI). Once the user has modeled the cubes into a final configuration, the configuration is transmitted by the desktop computer over the RS-232 interface back to the system.

In addition to the RS-232 transceiver, the ARM has two other communication interfaces. The first is the I^{2}C bus that it uses to interface with the PSoC. The ARM acts as the master of the bus and transmits requests to the PSoC. The PSoC then replies with the desired data. The final communication interface is the JTAG port, which can be used to debug the processor while software is being developed. The JTAG port allows the software developer to single–step through the source code, and, additionally, it also update the ARM's flash memory.

The last major piece of hardware on the ARM is the flash memory. One 8KB sector is reserved for use as a debugging log. If the ARM chip encounters errors during its normal course of execution, it logs them in this section of the memory. Then, even after the module has been power–cycled, the user can use the RS-232 interface to examine the contents of the log in an attempt to determine the source of any problems. All data stored in the log can be time–stamped with a resolution of one second using the clock which is illusrtated to the left of the debugging log in Figure 3-3 Time–stamping allows the user to reconstruct a composite sequence of events from event logs recorded in several different modules.

3.1.2 Secondary PSoC Processor

In addition to the ARM processor, each module in the system contains a Cypress CY8C29466 PSoC. Each PSoC has an 8-bit wide data path, 32KB of flash memory for program storage, and 2KB or RAM for data storage [9]. The PSoC is clocked by the same 11.0592MHz source as the ARM processor. The most significant benefit of using the PSoC chip is that it contains 32 reconfigurable hardware blocks that can serve many purposes. In particular, the CY8C29466 has 16 digital hardware blocks, eight of which are specifically designed for digital communication. It also contains 16 flexible analog blocks that lend themselves to processing and generating analog signals. Like the ARM, the PSoC can be reprogrammed without being removed from the circuit. Unfortunately, it cannot be debugged this way [9].

Figure 3-4 shows how we have configured the PSoC. The feature that is the basis for all of the PSoC's functionality is the I^{2}C slave interface shown in the upper right–hand corner of the diagram. It continuously waits for commands from the ARM. When it receives a command, it processes or generates the provided or requested data, and, if required, transmits a response to the ARM. The I^{2}C interface provides a means to control all of the PSoC's other peripherals.

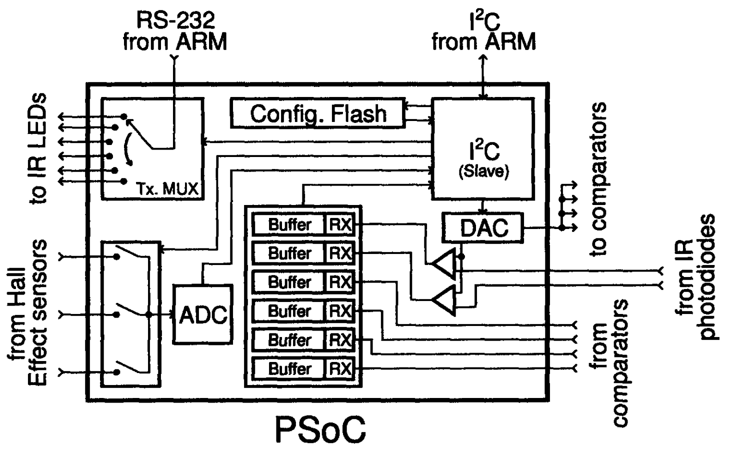


Figure 3-4: The PSoC processor implements much of the low–level functionality of the system. It is primarily responisble for buffering the data received from a module's six faces. The PSoC is controlled by the ARM processor using the I^{2}C interface.

Moving to the left across the top of Figure 3-4, the next important hardware resource is the configuration flash. This 64-byte sector of the PSoC's non–volatile flash memory is used by the module to store configuration parameters that must be retained when the power is switched off. This sector contains information such as the module's unique identification number and several threshold values, which ensure that the communication and orientation detection systems operate correctly.

Next, in the upper left–hand corner of the figure, two of the PSoC's digital blocks are used to implement a 1–input, 6–output multiplexer labeled, "Tx. MUX." This multiplexer, controlled by commands sent from the ARM processor over the I^2C interface, redirects the RS-232 data stream, (also generated by the ARM), to a specific IR LED on any of the six faces. By switching the output of the multiplexer, the ARM can send inter–module messages to a specific neighbor without also addressing all others.

Below the RS-232 multiplexer, Figure 3-4 illustrates the PSoC's analog–to–digital conversion subsystem. It is composed of two of the PSoC's analog blocks. The first part of the system uses a demultiplexer to select between three analog inputs: one from each of the Hall Effect sensors used in the connection mechanism described in Section 3.2. The single analog output of the demultiplexer is then routed to the analog–to–digital converter (ADC) to be processed. The ADC provides an 8-bit digital output that ranges from 0 to 255 and which is routed back to the I^2C block.

Moving to the right along the bottom of Figure 3-4, the next component is the RS-232 receiver. This block consists of six identical subsystems that first receive and decode RS-232 data stream and then buffer the received data. Each RS-232 receiver occupies one of the PSoC's digital communication blocks. Because the receivers are implemented in hardware, they can all operate in parallel. This ability allows a module to simultaneously receive messages from each of its six neighbors. Because the modules in the system do not transmit their messages in a synchronized pattern, it is important that a single module be able to handle messages from all of its neighbors concurrently. The buffers, which sit behind the RS-232 receivers, each have a capacity of 65 bytes—large enough to hold even the longest inter–module messages.

Additionally, the RS-232 receivers automatically detect the start and stop indicators transmitted with each inter–module message. When a complete message is received, the corresponding buffer will not accept any additional characters until it is explicitly flushed. This restriction guarantees that when the ARM queries any of the receive buffers using the I^2C interface, the ARM always finds an empty buffer or a complete message.

Finally, the bottom right–hand corner of the PSoC block diagram in Figure 3-4 illustrates the digital–to–analog converter (DAC) and two of the comparators that are necessary for inter–module communication. In addition to the two comparators inside of the PSoC, there are four additional comparators in a separate integrated circuit. Together, the six comparators process the analog voltages returned from the six IR photodiodes on the module faces. The comparators' outputs are digital logic levels that can be processed by the RS-232 receivers. By modifying the analog voltage produced by the DAC, we can control the sensitivity of the receivers. Section 3.3 details the infrared communication interface.

3.2 Connection Mechanism

Individual modules bind to each other using switchable permanent magnet assemblies, hereafter referred to as Magswitches. These assemblies are produced by Magswitch Technology, Inc., and one is shown in Figure 3-5. Three faces of each cube contain Magswitches. Like all other components, they are mounted on the inside of the cubes and pass through similarly sized holes in the printed circuit boards. The other three cube faces are covered by steel plates. The steel is cold rolled A336/1008 that is 0.033 inches thick. When multiple cubes are assembled into a structure, Magswitches always attach to their neighboring cubes' steel plates, not other Magswitches. As a result, the modules can only attract one another. They do not repel but, instead, depend upon gravity or user intervention to clear unused modules from any final structure. A single Magswitch connected to a neighbor's steel plate can support over 4.5 lbs.—the combined weight of 17 other modules.

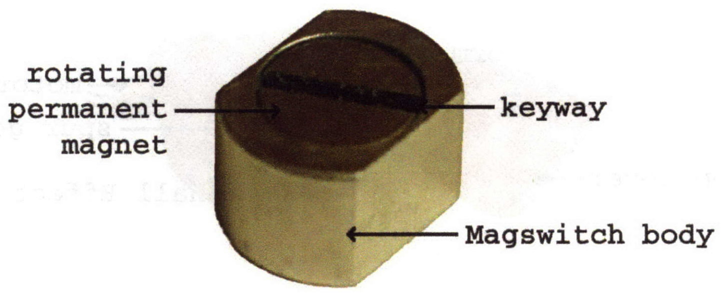


Figure 3-5: Each Magswitch consists of a two permanent magnets stacked on top of each other inside of metal housing. The bottom magnet is fixed while the top one contains a keyway and is free to rotate. As the top magnet is rotated through 180^o, the entire device switches from on to off or vice versa.

The Magswitch assemblies control a magnetic field by changing the relative orientation of two permanent disc magnets. The magnet with the keyway shown in the assembly in Figure 3-5 can rotate freely with respect to the fixed magnet that sits below it in the structure. Depending on their relative orientations, the Magswitch is either activated and attracts other ferromagnetic materials, or it is deactivated and releases it hold. The advantage of such a system is that power is only consumed while changing the state of the Magswitch. Once a Magswitch is on or off, it remains in that state indefinitely. This is invaluable for the battery life of the modules.

A miniature pager motor with an integrated gear box drives each Magswitch. These motors have stall torque of 0.28oz-in [12]. A 17-thread-per-inch worm gear is glued to the motor's output shaft. This worm gear turns a 30-tooth spur gear which has a key that matches the keyway of the Magswitch shown in Figure 3-5. The entire motor, worm gear, spur gear, and Magswitch assembly is illustrated in Figure 3-6. When driven with $4.1V$, the voltage of a fresh lithium-polymer battery, the motor requires approximately 1.3 seconds to switch a deactivated Magswitch on and back off again.

The motor driver circuit consists of a single MOSFET. As a result, the motor can only turn in one direction, but three additional MOSFETs, which would be needed to run the motor in both directions, are eliminated. The drawback to such

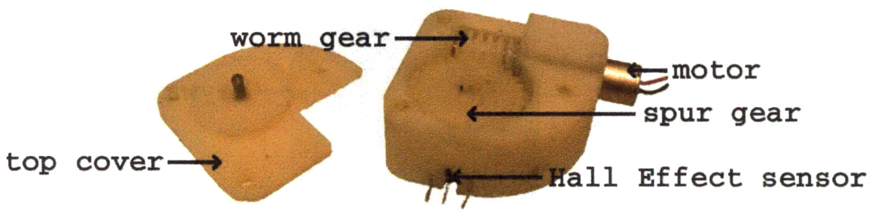


Figure 3-6: A worm gear attached to the output shaft of a minature DC motor turns a spur gear that mates with the keyway in the Magswitch. In the figure, the Magswitch is obscured by the spur gear, and the temporarily removed top cover of the entire assembly is shown on the left.

a configuration is that if the Magswitch is not completely deactivated, the process of rotating it to the off position may momentarily turn it on. Additionally, the Magswitches become difficult to activate if they are not placed in contact with the steel plate on a neighboring cube. Because without a steel plate to complete the magnetic circuit, the magnetic flux must flow through an air gap which has a high magnetic resistance. This magnetic resistance translates into mechanical resistance which the motor sometimes fails to overcome. As a result, there is no guarantee that a Magswitch can be deactivated if it is not in contact with another cube. This restriction, while inconvenient for testing, does not affect the capabilities of the system as a whole because the Magswitches only need to deactivate when they are already in contact with other cubes.

A Hall Effect sensor is used to detect the state of each Magswitch. The Hall Effect sensor is placed such that its axis of sensitivity is aligned with the magnetic field produced by the Magswitch. This is shown in Figure 3-7. The Hall Effect sensor produces an analog voltage that is proportional to the attractive force produced by the Magswitch. An analog to digital converter in the PSoC microprocessor converts this analog voltage to an 8-bit digital value that ranges between 0 and 255. Figure 3-8 shows how the converter's output approximates a sine wave as the Magswitch is rotated continuously. We chose to orient the Hall Effect sensor such that the peaks in the plot correspond to times at which the Magswitch is fully activated, and the minimums correspond to times when the Magswitch is completely off. Chapter 4 will discuss the algorithm that uses the Hall Effect sensor values to control the Magswitch's state.

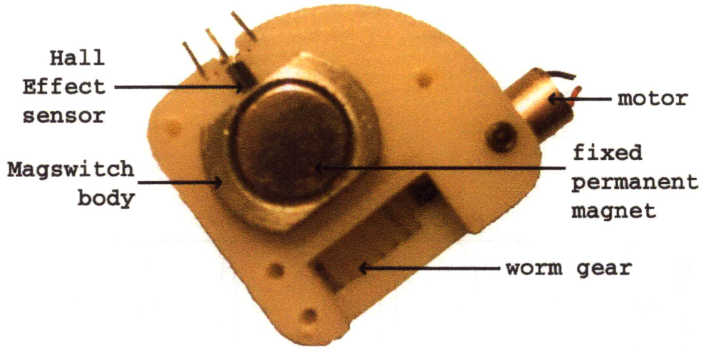


Figure 3-7: A Hall Effect sensor placed near the Magswitch is used to sense the amount of force it generates. We have chosen to align the Hall Effect sensor and Magswitch such that higher voltages at the output of the Hall Effect sensor correspond to stronger a attractive force.

3.3 Communication Interface

All communication between modules is performed using infrared light. Each of the six cube faces contains an infrared LED and an infrared sensitive photodiode. Together, these allow bidirectional communication between neighboring cubes at 9600 bits per second (bps). While higher bit rates were achievable, 9600bps proved adequate.

Infrared communication has several advantages over other alternatives such as direct electrical contacts. First, an infrared based system does not require that the faces of neighboring cubes be completely flush. In an assembly of many modules, this is a legitimate concern because imperfections in the manufacturing process produce cubes that are not perfectly square nor exactly the same size. Electrical contacts are also disadvantageous because they may short out on the steel plates that cover three of the six cube faces.

In order to simplify the design of the circuit boards which compose the faces of

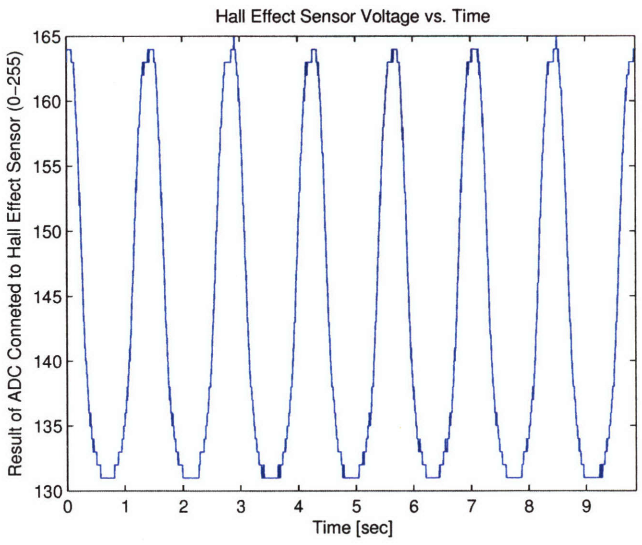


Figure 3-8: The output of the analog–to–digital converter connected to a Hall Effect sensor approximates a sine wave as the corresponding Magswitch is rotated continuously. The Hall Effect sensor and Magswitch are oriented such that peaks in the converter's output indicate that the Magswitch is exerting its maximum holding force. The minimums in the plot correspond to times at which the Magswitch was deactivated.

the modules, the infrared LED and photodiode were not always placed in the center of each face. This decision dictates that every module have only one valid orientation in a composite structure if the LEDs and photodiodes of neighboring cubes are to align. However, because any multi–module structure must be assembled by hand, this restriction does not affect the functionality of the system.

The infrared LED and photodiode have both a limited range and a limited field of view. Like all of the electrical components, the LED and photodiode are mounted on the inside faces of the modules, and they point down through holes in the circuit boards. In order to prevent these holes from further restricting the field of view of the emitters or receivers, they are countersunk on the back (bottom) side of the boards.

Figure 3-9 shows how the voltage at the output of the photodiode varies with distance between the photodiode and the infrared LED if they are laterally aligned. It illustrates that two cubes must be immediate neighbors in order to communicate. It is not possible for two cubes separated by the width of another cube to communicate. Figure 3-10 shows how the voltage varies with alignment offset between the emitter and detector if two modules have their faces touching. To convert these analog voltages to digital signals that can be used by a microprocessor, a voltage comparator is used. It forces any input voltage above a threshold to 3.3V and any voltage below that threshold to 0V. Empirically, a threshold voltage of 2.1V provided the best compromise between robustness to misalignment and noise rejection. Higher thresholds tended to produce noisy output, and lower thresholds reduced the system's robustness to misalignment. The chosen 2.1V threshold allows two modules to communicate if they were placed within approximately 0.25 inches of each other. The threshold voltage is stored in the PSoC's flash memory, and it is simple for the user to change using the ARM's RS-232 interface if desired.

3.4 Orientation Detector

Each module is able to detect its absolute three-dimensional orientation by using a two-axis accelerometer and a binary tilt switch that are connected to the ARM

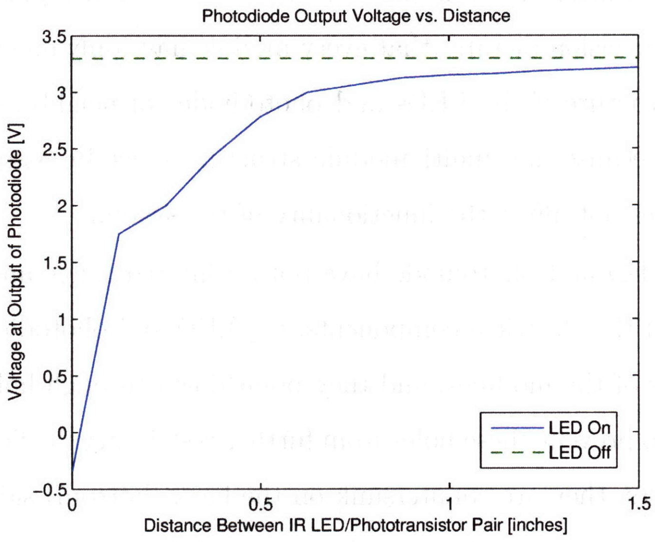


Figure 3-9: The voltage at the output of the photodiode varies as the IR LED is moved farther away. When the LED is off, the IR photodiode produces 3.3V. When the LED is turned on, the voltage produced by the photodiode drops by an amount inversely proportional to the distance.

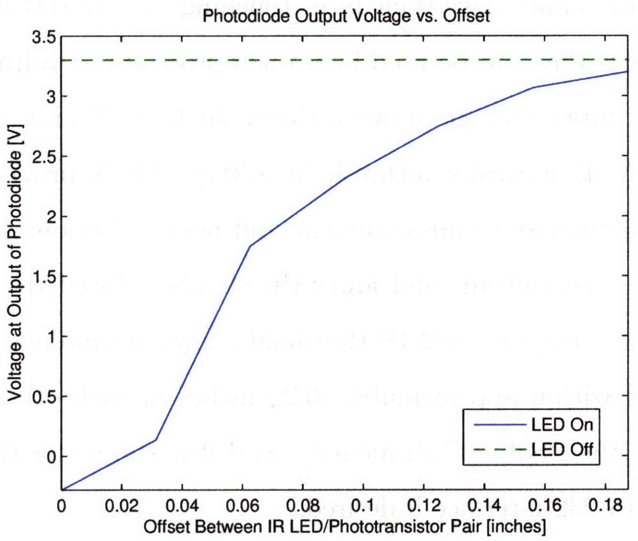


Figure 3-10: The voltage change at the output of the IR photodiode varies as the misalignment with the IR LED increases. In this test, two modules were placed face to face, and their alignment offset was gradually increased. When the LED is off, the IR photodiode produces 3.3V. When the LED is turned on, the voltage produced by the photodiode drops by an amount inversely proportional to the alignment offset.

microprocessor. The accelerometer returns two PWM signals that correspond to the acceleration that each axis is experiencing. The period of these signals is fixed, but the percentage of one period that the signal is on is proportional to acceleration. The ARM microprocessor measures the pulse width of the two signals to obtain an estimate of the cube's orientation.

The specific accelerometer, the MXD2004AL, produced by Memsic, Inc., was chosen because it is highly resistant to the large accelerations that can result from being dropped []. Given a structure of modules that is disassembling, many of the unused modules will detach and fall away from the structure. The accelerometer should be able to survive these falls.

A tilt switch is needed in addition to the accelerometer in order to detect a module's orientation when neither of the two accelerometer axes is experiencing acceleration due to gravity. There are two possible orientations which produce such a situation. The tilt switch differentiates between these two configurations by turning on or off. Like the accelerometer, the tilt switch was chosen for its durability. Instead of a typical mercury–filled glass cylinder, each tilt switch uses a small metal ball bearing encased in a metal cylinder. While the tilt switch only tells the microprocessor whether the module is oriented roughly up or down, this information, combined with the more precise data from the accelerometer, is enough to determine which side of a module is facing down.

3.5 Power Regulators

Each module is equipped with two rechargable lithium–polymer batteries connected in parallel. These batteries supply power to the module's electronics and motors. They provide 3.7V nominally and have a combined capacity of 340mAh. If the batteries are fully charged and the module is continuously transmitting messages on each face but not running its motors, the useable battery life is over six hours. The batteries drive the motors directly, but two voltage regulators provide power for the electronics. One produces 3.3V which is used by all of the components. The other regulator produces 1.8V which is only used by the core of the ARM microprocessor. The voltage regulators were chosen for their low–dropout voltages. The 3.3V regulator stops operating only when the batteries produce less than 3.5V [3]. Likewise, the 1.8V regulator drops out when the batteries can only supply 2V [4].

The modules can be recharged without removing the batteries. Each module contains an integrated circuit that manages the process. The electrical connection to recharge the batteries is provided through two of the metal faces that adorn the outside of the cubes. Large areas of solder mask are missing on the bottom (outside) of two of the printed circuit boards that form the faces of the cubes in order to electrically connect the steel plates to the circuit. To achieve a reliable connection, the plates are affixed with conductive epoxy. To recharge the batteries, the modules are set in a 28 inch long trough whose metal sides supply a potential difference of 5V. Because the trough is so long, it can recharge 15 modules simultaneously. Current to recharge each module's batteries flows from the sides of the trough, through the metal faces and conductive epoxy to the solder mask–free contacts on the back of the printed circuit boards. The integrated circuit responsible for managing the charging process automatically detects when a charging voltage is present. Therefore, starting or stopping the charging process is achieved by simply placing the modules in or removing the modules from the charging trough.

Chapter 4

Low–Level Software

To support the algorithms that allow our system of modules to disassemble, we have implemented a series of low–level functions that control the hardware in each module. These routines place an abstraction barrier between the localization, shape distribution, and disassembly algorithms and the complex hardware contained in each module. This separation facilitates the rapid implementation and modification of the high–level concepts which are responsible for the system's visible behavior. The high–level algorithms do not have to contend with the specifics of basic tasks such as exchanging messages or activating a Magswitch.

Once a module has the ability to transmit and receive messages, the low–level operation reduces to the simple process illustrated in Figure 4-1. After initializing, a module loops forever, simply receiving and transmitting messages to its neighbors. The interesting behavior responsible for the system's self–disassembly is governed by how the high–level algorithms, described in Chapters 5 and 6, respond to received messages.

Section 4.1 of this chapter explains how the ARM and PSoC processors communicate. Their cooperation is essential to all other functionality displayed by the modules. Both the ARM and PSoC are necessary to actuate the Magswitches, send messages, or receive messages from a module's neighbors. Section 4.2 presents the algorithm used by the ARM processor to control the Magswitches using Hall Effect sensors.

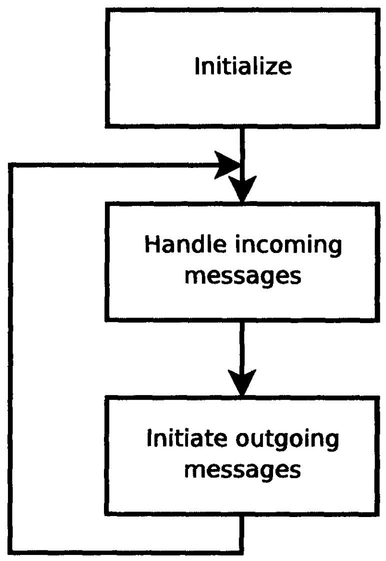


Figure 4-1: The message processing loop executing on each module is simple. First, modules initialize all their peripherals. Then, they loop infinitely, receiving and sending inter–module messages. How a module changes its internal state in response to received messages and what messages it transmits in return, dictate the system's high–level abilities.

Sections 4.3 and 4.4 explain the process a module must follow in order to send and receive messages, respectively. In addition to explaining how messages are transmitted, Section 4.3 details the basic structure of all inter–module messages. Finally, Section 4.5 examines several specific inter–module messages and the actions that their reception prompts.

4.1 I^{2}C Communication

The I^{2}C bus protocol is based on the concept of a master and a slave. The master transmits requests to the slave, and the slave replies [10]. We have designed our disassembling system so that the ARM microprocessor is the master and the PSoC is the slave. This relationship allows the ARM to be in control of the bus and request data from the PSoC when necessary. The ARM may initiate two types of data transfers: write and read. Writing and reading data cannot be combined in a single I^{2}C transaction. As a result, if the ARM wants specific data from the PSoC, it must utilize a write transaction to inform the PSoC what data it wants before it executes the read operation. While the I^{2}C specification allows for data rates of up to 3.4Mb/sec [10], we found that data rates above 20Kb/sec were unreliable given our choice of hardware. In large part, we feel this is due to the inability of the PSoC to simultaneously process the RS-232 data from the module's six neighbors and the I^{2}C commands from the ARM.

Each I^{2}C exchange, regardless of whether the ARM is transmitting or receiving, begins with the ARM sending one byte of data that contains the 7-bit I^{2}C address of the PSoC (64) and one bit that indicates whether the ARM wishes to read or write data. If the ARM is writing data, the second byte of the I^{2}C transmission is a command byte that indicates what type of data, if any, is to follow. Some commands transmitted by the ARM are followed by additional data bytes which further specify how the PSoC should react. If the ARM is receiving data from the PSoC, the PSoC should already know what type of data to provide, and every byte in the transaction after the first is sent by the PSoC to the ARM.

4.1.1 I^{2}C Commands

Table 4.1.1 provides a list of valid I^{2}C commands, the number of data bytes that are to follow, and an indication of whether they prepare the PSoC for a subsequent read transaction. For example, consider the first command listed in the Table 4.1.1, `MONITOR HALL SENSOR`. If the ARM wishes to read the current value of one of the Hall Effect sensors, it must first transmit a `MONITOR HALL SENSOR` command followed by one byte which specifies which Hall Effect sensor to monitor. After the I^{2}C write is complete, the ARM initiates a read operation to which the PSoC replies with one byte containing the most recent output of the ADC connected to the specified Hall Effect sensor.

The `STOP ADC` command is closely related to the `MONITOR HALL SENSOR` command. It is issued by the ARM whenever it realizes that it will not need to change the state of any Magswitches for a significant period of time. Because each Magswitch is actuated at most twice during any assembly/disassembly sequence, such times are

Table 4.1: The commands transmitted from the ARM to the PSoC over the I^{2}C bus may be one byte long, or they may be followed by several parameters as specified by the *Bytes to Follow* column. Some commands, as denoted by the *Read Next* column, also prepare the PSoC to respond to request to read data.

I^{2}C Command	*Bytes to Follow*	*Read Next*
`MONITOR HALL SENSOR`	1	Yes
`STOP ADC`	0	No
`SET TX CHANNEL`	1	No
`SET COMPARATOR`	1	No
`GET RX STATUS`	0	Yes
`READ RX BUFFER`	1	Yes
`SAVE CFG DATA`	64	No
`READ CFG DATA`	0	Yes

common. When the PSoC receives the `STOP ADC` command, it turns its ADC off to conserve power until another `MONITOR HALL SENSOR` command is received. There is no additional data sent along with the `STOP ADC` command.

Other commands operate in similar fashion to the two already discussed. The `SET TX CHANNEL` command is used to control the digital multiplexer which redirects the RS-232 data stream generated by the ARM to an IR LED on any of the six faces. The additional byte expected after the initial command specifies the face to which the stream should be redirected.

The `SET COMPARATOR`, `GET RX STATUS`, and `READ RX BUFFER` commands control the flow of inter–module messages received from a module's six neighbors. In particular, the `SET COMPARATOR` command modifies the output of the DAC, which drives the non–inverting inputs of the comparators used for processing the outputs of the six photodiodes. In this case, the additional byte transmitted by the ARM after the initial command specifies the output value of the converter. This, in turn, sets the threshold of the comparators. The `GET RX STATUS` command is used by the ARM to check which of the six RS-232 receive buffers have messages pending. The command requires no additional argument, but like the `MONITOR HALL SENSOR` command, readies the PSoC for a read operation in which it transmits a single byte which indicates which of the six receive buffers is full. After determining which of the six buffers contains a pending message using the `GET RX STATUS` command, the ARM typically issues several `READ RX BUFFER` commands to read the contents of every buffer that contains a pending message. The additional byte sent after the `READ RX BUFFER` command specifies which buffer the PSoC should prepare to transmit back to the ARM. A subsequent I^2C read retrieves that buffer contents.

Finally, the `SAVE CFG DATA` and `READ CFG DATA` commands provide a means for the ARM to store and retrieve configuration data from a single 64-byte page of the PSoC's flash memory. The `SAVE CFG DATA` command can be followed by up to 64 bytes of data to be stored. Likewise, the `READ CFG DATA` command prepares the the PSoC to return those 64 bytes during the next I^2C read operation.

4.2 Magswitch Control

This section describes how feedback from a Hall Effect sensor and the ability to only turn a Magswitch in one direction, it is possible to precisely control the state of each Magswitch. As shown in Figure 3-8, the voltage produced by the Hall Effect sensor approximates a sine wave as the Magswitch is rotated continuously. We have oriented the Magswitches and Hall Effect sensors such that when the voltage produced by the Hall Effect sensor is at a minimum, the Magswitch is deactivated. Likewise, when the voltage is maximized, the Magswitch exerts its maximum holding force.

Due to variations in the assembly process, not all of the Hall Effect sensors are placed at exactly the same distance from the side of the Magswitches. This results in a variance in the minimum and maximum voltage produced by each sensor. As illustrated by the red data points in Figure 4-2, the variance is significant when the Magswitches are activated but not in contact with a neighbor's steel plate.

4.2.1 Threshold–Based Magswitch Control

Initially, we attempted to control the state of the Magswitches using threshold–based algorithm. We assumed that if the reading from the Hall Effect sensor was above a certain value, the Magswitch was activated. If the value was below a different

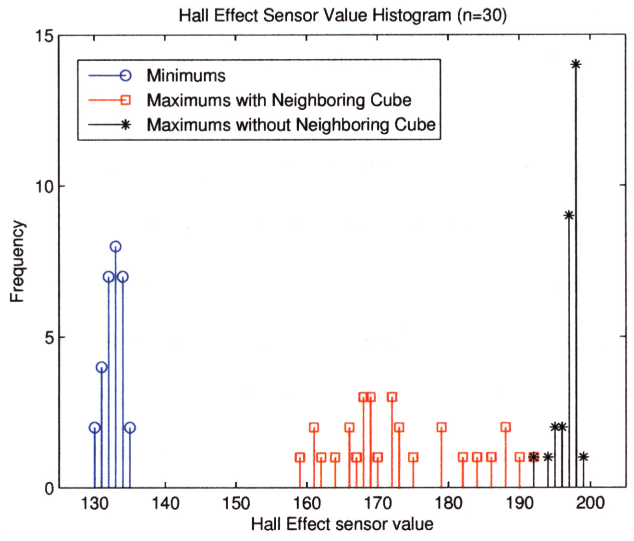


Figure 4-2: The variance seen among the minimum and maximum values of the Hall Effect sensors monitoring the Magswitches makes it difficult to determine when a specific Magswitch is activated or deactivated using simple connection and disconnection threshold. To overcome this problem, we looked at the derivative of the Hall Effect sensors value to determine when the Magswitches were on and off.

threshold, we assumed the Magswitch was deactivated. As soon as the appropriate threshold was crossed, the Magswitch control algorithm turned off the appropriate motor to stop the Magswitch's rotation.

The threshold–based algorithm presented several problems. First, it required that we individually calibrate every Magswitch of every module with an appropriate connection and disconnection threshold. These thresholds were stored in the PSoC's flash memory. Every time we reprogrammed a PSoC, the calibration information was lost.

The second problem stemmed from the fact that a Magswitch must be completely deactivated in order to release its neighboring module. We found that the force exerted by a Magswitch when nearly deactivated is very sensitive to how close the Magswitch was to the minimum. Even if the Magswitch was close, it still exerted enough force to lift another cube. In an attempt to ensure that the modules disconnected reliably, we were forced to set the disconnection threshold as low as possible. Unfortunately, the readings from the Magswitches varied slightly across experiments. Sometimes, the voltages produced by the Hall Effect sensors did not cross the thresholds necessary to stop further rotation. When attempting to disconnect, some Magswitches would rotate until a timeout was reached instead of ever stopping at point of minimum attractive force. This phenomenon was less of a problem when activating the Magswitches because even a partially activated Magswitch is strong enough to support several other modules. As a result, we could set the connection threshold farther from the true maximum value of the Hall Effect sensor.

4.2.2 Slope–Based Magswitch Control

We made the process of connecting and disconnecting the Magswitches more reliable by examining the slope of the values returned from the Hall Effect sensor to detect when the Magswitch's holding force was maximized or minimized. In particular, this happens when the Hall Effect sensor data has zero slope. A flowchart illustrating this approach when attempting to connect a Magswitch is presented in Figure 4-3. The algorithm assumes that the Magswitch has reached its maximum if the slope of the values returned by the Hall Effect sensor transitions from positive to zero or to a negative number.

The algorithm first checks to make sure that a specified time has not expired without connecting. If an error occurs, this check ensures that the ARM does not attempt to rotate the Magswitch indefinitely. If the time–out has not occurred, the algorithm reads a new value from the Hall Effect sensor and averages it with several of the most recent values in order to smooth any noise present in the data. If the new average is the same as the previous average, a counter is incremented. Then the value of the counter is checked. If it is large enough, it indicates that the readings from the sensor currently have zero slope. Next, the algorithm examines the previously computed slope of the Hall Effect sensor values. If it is negative, but the newest average is larger than the previous average, the algorithm updates its estimate of the slope to positive and begins another iteration of the loop by rechecking the timer. If the previous estimate of the slope is positive and the new slope is either zero

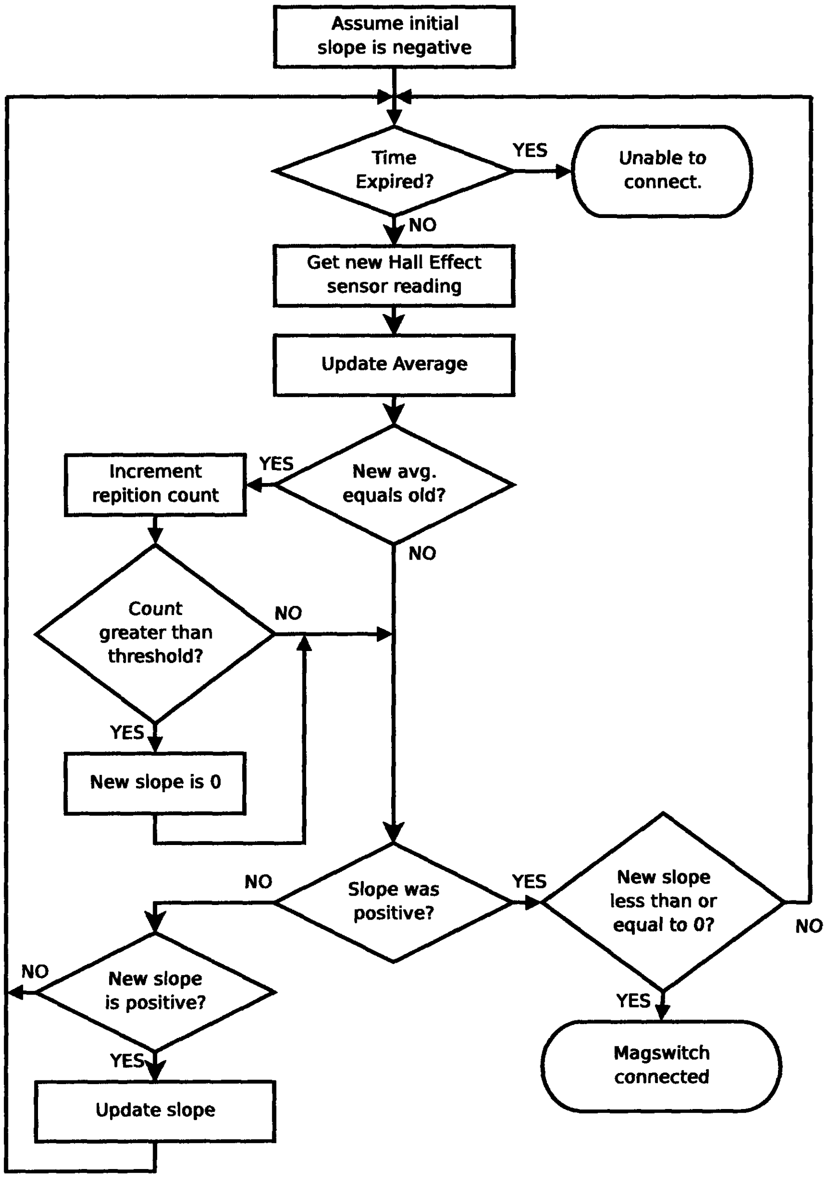


Figure 4-3: The slope–based algorithm used to activate a Magswitch stops the Magswitch's rotation when the slope of the Hall Effect sensor values transitions from positive to zero or a negative value.

or negative, the algorithm assumes that the Magswitch has achieved its maximum holding strength. Consequently, it deactivates the motor. A similar algorithm is implemented to deactivate the Magswitch.

To further ensure the system's reliability, every request to activate or deactivate a Magswitch results in the Magswitch switching both on and off, but not necessarily in that order. For example, when one of the high–level algorithms makes a request to activate a Magswitch, the Magswitch is first deactivated and then activated. Likewise, a request to deactivate a Magswitch results in the Magswitch first being turned on, after which it is turned off.

This strategy of turning a Magswitch on before turning it off, coupled with the slope–based algorithm above, produced better results than the threshold–based system described in Section 4.2.1. The most noticeable advantage to the slope–based system is its accuracy. When a Magswitch is under control of the slope–based algorithm, it always stopped very near the position that exerts the minimum or maximum force on its neighbor.

Table 4.2.2 compares how successful the two algorithms were by examining how often Magswitches controlled by the threshold based system, as opposed to those controlled by the slope–based algorithm, were still able to lift a neighboring cube after being deactivated. The results of the comparison favor the slope–based algorithm because it does not require any configuration, and a Magswitch under its control was never able to lift another module after being deactivated.

4.3 Inter–Module Message Transmission

This section explains how messages are transmitted from a module to one of its neighbors. As explained in Section 3.3, all inter–module communications utilize the IR LED and photodiode pairs that exist on each of the module's six faces. The process of transmitting a message involves several steps. First, the message body is constructed. Then, a checksum produced by a cyclic redundancy check (CRC) is appended to the message. Next, the ARM processor sends a `SET TX CHANNEL`

Table 4.2: The threshold–based and slope–based algorithms were compared while trying to disconnect a Magswitch from a neighboring cube. There were three possible outcomes, which are listed along the left side of the table. Successful outcomes corresponded to the Magswitch releasing its hold on the neighboring cube. Unsuccessful outcomes occurred when the Magswitch was still active enough to lift the neighboring cube. In some cases, time–outs occurred because the algorithm could not find what it believed to be the off position of the Magswitch. Given a correct threshold, the threshold–based algorithm was as successful as the slope–based one, but the threshold had to be chosen carefully. The slope–based algorithm excels because it does not require any configuration and always operates correctly.

	Threshold-based Alg.							*Slope-based Alg.*
Threshold	134	135	136	137	138	139	140	
Successful	0	15	15	5	11	1	0	30
Unsuccessful	0	0	0	10	4	14	15	0
Time–out	15	0	0	0	0	0	0	0

command over the I^2C interface directing the PSoC to activate the RS-232 transmission multiplexer so that the message is directed to the correct face. After the PSoC acknowledges that it received this command, the ARM sends the message over the RS-232 interface, through the PSoC, and to the correct IR LED. Finally, once all bits have been transmitted, the ARM once again uses an I^2C bus to deactivate the transmission multiplexer.

4.3.1 Message Format

Messages themselves are constructed from the standard set of printable ASCII characters. We make no attempt to compress the messages into a space–efficient format. While this lengthens the transmission time, it also makes it easy for anyone debugging the system to see exactly what is happening by simply scanning all of the messages that a module transmitts. The software running on the ARM builds the messages in a sequential manner starting at the beginning of a message and adding the appropriate information until the message is complete.

Each message begins with a unique start character—we chose the pound character, (`#`). Messages are also terminated by a unique character, the carriage return. Unique start and stop characters allow a stream of messages to be parsed accurately and efficiently. As explained in Section 3.3, the RS-232 receiver blocks in the PSoC do not begin to fill their associated receive buffers until a start character is detected. Likewise, they stop writing to their buffers when a stop character is detected. This ensures that any message the ARM processor reads from the the PSoC's receive buffers is complete.

4.3.2 Message Structure

After the start character, each message consists of a number of alphanumeric fields separated by ampersands. The ampersands never appear within a field, making it easy for the routines that process the received messages to identify any field by number. The structure of a typical message is shown in Figure 4-4.

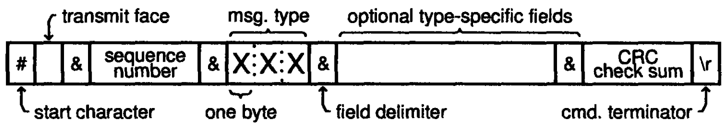


Figure 4-4: Messages are composed of a start character, some number of alphanumeric data fields separated by ampersands, a hexadecimal checksum, and a message terminator.

After the pound character, which starts each message, a single–digit number is added which specifies on which face the message is to be transmitted. This transmission face field exists solely for the purpose of debugging the system and allows anyone monitoring the messages transmitted by a module to correlate specific messages with each face. The transmission face field, like all data fields, is terminated by an ampersand. Following the transmission face field, each message contains a sequence number field which is a positive number of no more than four digits. As each new message is constructed by the software, the sequence number is incremented. As a result, messages that are transmitted a short time apart are guaranteed to carry unique sequence numbers. When a neighbor acknowledges a message, the original sequence number is echoed back to the sender to eliminate any ambiguity as to which message the neighbor is acknowledging.

After the sequence number field, each message contains a three letter type field. Different message types are used to accomplish different system tasks. Any module receiving a message will process different types of messages in different ways. Message types associated with low–level functionality are covered in Section 4.5. The message type field may be followed by several more type–specific fields, or it may be the last field before the CRC checksum. After the last message field and its terminating ampersand have been attached to the message, a 16-bit checksum is calculated. This checksum is represented in hexadecimal and appended to the end of the message. This ends the process of message construction.

4.3.3 Message Buffering

Instead of being transmitted immediately upon construction, a message is placed into a face–specific transmission buffer along with a number that specifies how many attempts should be made to transmit the message before timing out. If the destination module acknowledges the receipt of the message in the buffer before it has been transmitted the specified number of times, the buffer is cleared and the message is assumed to have been successfully transmitted. In practice, attempting to transmit a message 500 times usually proved more than sufficient.

Once a message is placed in the transmit buffer, the function that placed it there can forget about it. The function is not responsible for ensuring that the message is retransmitted the specified number of times or transmitted until an acknowledgment is received. Instead, retransmission happens as part of an interrupt routine that is triggered by the hardware–based Timer 1 at regular intervals. Each time that an interrupt occurs, the transmit buffer is checked for pending messages. If there is a pending message for any face, it is transmitted then. The interrupt routine transmits no more than one message on each face before returning control to the main program.

The software running on the ARM microprocessor treats acknowledge (`ACK`) and not acknowledge (`NCK`) messages differently than any other type of message. Instead of being placed into a buffer along with a retry count, `ACK` and `NCK` messages are transmitted in place of any other message that is pending in the buffer for a specific face, but they are only transmitted once. Normally, when a new message is placed into a transmit buffer, the message that was already present, regardless of whether it had been successfully transmitted or not, is overwritten. ACK and NCK messages do not override any messages already buffered for transmission. After the ACK or NCK message has been transmitted, the buffer returns to sending whatever message it had been transmitting previously.

This caveat in the message buffers ensures that ACK and NCK messages are not transmitted unnecessarily. Assuming that there are no problems with a received message, a single ACK message is transmitted in response. Because a module receiving an ACK or NCK message does not send an ACK message in response, the module transmitting the ACK or NCK message would always transmit the message the maximum number of times. This should be avoided because both the transmitting and receiving module must waste valuable processing time dealing with the extraneous messages, and the extra messages deplete the battery life of the modules. Finally, the fact that ACK and NCK messages do not overwrite any messages already in the transmission buffers improves two–way communication between modules. Messages can flow into a module through a specific face without significantly delaying any message already being transmitted on that face.

4.4 Inter–Module Message Reception

Receiving messages from neighboring cubes requires several steps. First, the ARM must use the I^2C bus to configure the digital to analog converter that drives the non–inverting inputs of the comparators. Once the DAC is configured, it will continue to supply a constant voltage. As a result, this step is only necessary when a module turns on. Assuming that the threshold voltage is reasonable, the ARM proceeds to query the PSoC receive buffers for the presence of any messages using the GET RX STATUS I^2C command. If some buffer contains a valid message, the ARM issues a READ RX BUFFER command to retrieve the message. After the PSoC transfers a message to the ARM, it automatically empties the buffer that had been holding the message. Doing so allows the buffer to once again begin filling with the next message that is received on the specified face.

Once the ARM has retrieved a message from the PSoC, it begins to process it. The first thing that it checks is the CRC checksum. It separates the checksum of the received message from the message body and recomputes the checksum of the body. If the computed checksum does not match the checksum that was received, the message is assumed faulty. Faulty messages prompt the software to send a not acknowledge, (`NCK`), message on the face on which the message was received. As explained Section 4.5.2, `NCK` messages cause the original transmitter to increase the number of times it attempts to transmit the message in question.

If the received message passes the initial CRC verification process, the software then identifies the message by its three–letter type. Depending on the type, the software parses the message for all of the associated data fields. In some cases there may be none. If the parsing routines are unable to extract a specific field, they interpret this as an error, abort the message processing, and send a `NCK` message to the original transmitter. If they successfully parse the message, they pass the contents of all of the message fields on to a type–specific handler. The data fields unique to each type of messages and the handlers for those low–level messages are described in the next section.

4.5 Basic Message Types

Modules use a number of messages to accomplish low–level tasks that do not directly affect the high–level behavior of the system. This section examines the format and implications of many of those inter–module messages.

4.5.1 Acknowledge (`ACK`) Messages

Acknowledge (`ACK`) messages are sent by modules in response to the reception of most valid messages. The structure of an `ACK` message is shown in Figure 4-5. It contains two message specific data fields. The first, the initial type field, specifies the three–

character type of the initial message that justified transmitting the ACK message. The second unique field is the initial sequence number field. Similar to the initial type field, it contains the sequence number of the initial message that was the impetus for the transmission of the ACK message.

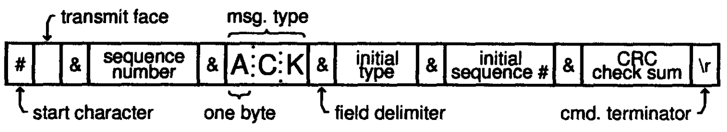


Figure 4-5: Acknowledge (ACK) messages contain two type specific fields. The first specifies the type of message in response to which the acknowledge message was generated. The second specifies the sequence identifier of that message. Together, these fields are used to correlate ACK messages to messages in a module's transmit buffer.

When a valid ACK message is received, two things happen. First, the initial message type and initial sequence number fields are compared against any message currently buffered for transmission on the receiving face. If the message types and sequence numbers match, the message in the buffer is assumed to have been successfully received and is not retransmitted in the future. The ARM also interprets a valid match as an indication that the module is able to successfully send and receive messages on the face that was the source of the ACK message. This is a valid assumption because the module's neighbor would not transmit an ACK message unless it has received a valid message first. Second, if the module from which the message originated was previously unknown, the ARM now notes that the module has a new neighbor.

4.5.2 Not Acknowledge (NCK) Messages

Not acknowledge (NCK) messages are similar to ACK messages except they are transmitted in response to errors encountered while processing a received message. Because they are transmitted in response to faulty messages, no information about the original message is included in the NCK. If such data were included, there would be no guarantee that it was correctly received in the first place. As a result, NCK messages contain no type specific fields.

When a module receives a NCK message, it compares the sequence number of the received message to the sequence number of the last NCK message to be received on the same face. If these sequence numbers differ, the ARM increments the number of times any message currently in the receiving face's transmission queue is assigned to be retransmitted. The ARM ensures that the sequence numbers differ before incrementing the retransmission count in order to avoid situations in which the transmission buffer never stops transmitting one specific message. Empirically, we found that incrementing the retransmission count by 5 was sufficient. We also found that the modules rarely transmit NCK messages. Usually, a module receives all of a message correctly or none of it.

4.5.3 Disconnect All Magswitches(DCA) Messages

Disconnect all Magswitches (DCA) messages are employed to quickly disconnect all modules in any structure that has been previously assembled. DCA messages lack any type specific fields. When a module receives a DCA message, it send an ACK message back to the transmitter. It forwards the DCA message on to every other face overriding any messages that were already in the transmission buffers. After it has done this, the module attempts to disconnect all three of its Magswitches if it knows that they are not already deactivated. Because of the aggressive manner in which they are forwarded, DCA messages will quickly propagate through any assembly of modules.

4.5.4 Disconnect Request (DRQ) Messages

Disconnect Request (DRQ) messages are transmitted by modules wishing to disconnect from the structure. In many cases, a module wishing to disconnect will be connected through the structure with not only one if its own Magswitches but also by one of its neighbors. In these cases, the DRQ message is transmitted to the neighbor. As a result, DRQ messages are only transmitted on faces that lack a Magswitch and should always be received on a face that does. If a module receives a DRQ message, it assumes that the receiving face is the one on which it should deactivate one of its Magswitches.

Therefore, DRQ messages do not contain any type specific fields. When a module receives a DRQ message, it transmits an ACK in reply and then attempts to deactivate the Magswitch on the receiving face.

4.5.5 Magswitch State (MSS) Messages

Magswitch state (MSS) messages indicate the state of the Magswitch on the face from which they are transmitted. To do so, they include one type specific "state" field as illustrated in Figure 4-6.

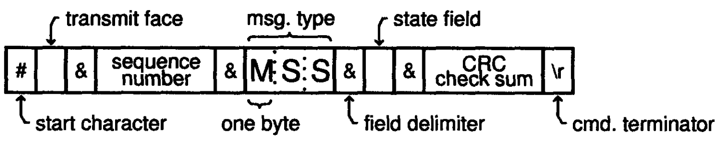


Figure 4-6: Magswitch state (MSS) messages contain one type specific field which indicates the state of the Magswitch on the face from which the message is transmitted. A MSS message is transmitted whenever the Magswitch on a face attempts to change state.

A MSS message is transmitted whenever a module attempts to change the state of a Magswitch. The message is transmitted regardless of whether that attempt is successful. The state field indicates the final state of the Magswitch, not whether the actuation attempt was successful. When a module receives a MSS message, it sends an ACK in reply. Then it updates its internal model of the transmitting neighbor to reflect the current state of the neighbor's Magswitch.

4.5.6 Reset (RST) Messages

Reset (RST) messages are used to reset the internal state of each module. This simple task requires no message specific fields. When a module receives an RST message, it acknowledges it by sending an ACK message to the transmitter and rebroadcasts the message on all other faces. An RST message sent to one module propagates through the entire assembly. After receiving an RST message, the module forgets any information it had amassed about its neighbors. Receiving an RST message is equivalent to manually resetting the module.

4.5.7 Real Time Clock (RTC) Messages

Real time clock (RTC) messages are used to reset the clock that is contained on the ARM microprocessor in each module. As illustrated in Figure 4-7, RTC messages contain numerous type specific fields. Fields exist for the current year, month,day of the year, day of the week, day of the month, hour, minute, and second.

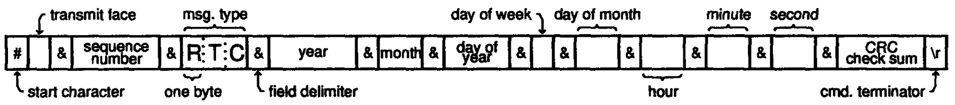


Figure 4-7: Real time clock (RTC) messages are used to synchronize the hardware clock of all modules in a system for debugging purposes. They contain many type specific fields which are used to transmit the current date and time.

When a module receives an RTC message, it updates the hardware clock in the ARM processor, sends an ACK message to the neighbor from whom the message originated, and rebroadcasts the message to all other neighbors. In this way, the RTC message will reach every module in an assembly of many, assuming there are no serious communication failures. In general, individual modules do not initiate the propagation of RTC messages. Instead, a user debugging the system transmits an RTC message to synchronize the clocks of each module so that different modules' debugging logs can be combined into a composite trace which includes a time–stamped list of all events that occurred in the system.

Chapter 5

Self–Disassembly Algorithms

The algorithms which give the system its unique ability to self–disassemble can be divided into the four phases illustrated in Figure 5-1. The first, neighbor discovery, which commences after the modules are reset, is discussed in Section 5.1. During this phase, modules detect any neighbors in close proximity and attempt to establish mechanical and communication links.

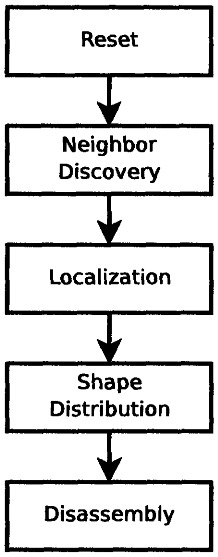


Figure 5-1: The entire self–disassembly process consists of four phases: neighbor discovery, localization, shape distribution, and disassembly.

During the localization phase, which follows neighbor discovery, modules discover their positions within the structure and transmit their positions back to the GUI running on the user's desktop computer. Localization is explained in Section 5.2. Once each module has transmitted its position to the GUI, the GUI can form a model of the system. Using this model, the desired final configuration of modules can be virtually sculpted. Using the GUI, each module can be included or excluded from the final shape. After this sculpting process is complete, the GUI generates a sequence of shape distribution messages that is sent to the modules during phase three. The algorithm used to generate this sequence of inclusion messages is presented in Section 5.3. Next, Section 5.4 addresses the shape distribution phase during which the modules propagate the inclusion messages generated by the GUI. The fourth, and final, phase is disassembly. During the disassembly phase, which is detailed in Section 5.5, extra modules disconnect from the system to reveal the shape the user sculpted using the GUI.

This chapter is primarily concerned with the theoretical aspects of the algorithms used in the self–disassembly processing. Their practical implementation will be addressed in Chapter 6. Additionally, Chapter 6 will address how the GUI is implemented.

5.1 Neighbor Discovery

Before a structure of modules can self–disassemble, the structure must first be constructed from a set of individual modules during the neighbor discovery phase. After being reset or being turned on for the first time, all modules begin in the neighbor discovery phase. During neighbor discovery, every module uses its IR LEDs and photodiodes to detect and connect to its neighbors.

The pseudocode for the neighbor discovery phase is provided in Listing 5.1. Every module begins by transmiting ping messages on all faces. If a module receives an acknowledgment on any face, it knows that a neighbor exists in that direction. If that face contains a Magswitch, the module activates it in order to bind to the neighbor. It is also possible that a module receives its neighbor's ping message before

Listing 5.1: The neighbor discovery algorithm broadcasts ping messages on each face until it receives an acknowledgement at which point it stops broadcasting ping messages on the receiving face and activates the face's Magswitch.

```
loop until localization message received
    for each face
        if no aknowledgment message has been received
            transmit a ping message
        elseif acknowledgment message newly received
            stop transmitting ping messages
            if face has Magswitch
                activate Magswitch
```

an acknowledge message. In this case, the module behaves identically–it attempts to activate the face's Magswitch, (if present). As discussed in Chapter 3, the range of the IR system is limited to approximately 0.25 inches. This ranage has prevented false detections in all observed cases. Once a module receives an acknowledgment that one of its ping messages has been received by a neighbor, it stops transmitting ping messages on that face because any further ping messages would be redundant. Because neighbor discovery occurs independently on each cube, it only requires $O(1)$ time for the phase to complete. At the end of the neighbor discovery phase, the modules have formed a solid structure, and they are ready for localization.

5.2 Localization

The localization phase ensures that each module discovers its absolute three–dimensional coordinates in the system. Once each module has that information, it can notify the GUI running on the desktop computer of its existence. Once all modules have localized and transmitted their positions to the GUI, the GUI has a complete model of the initial configuration of the system.

5.2.1 Localization Message Propagation

The localization process is initiated through the use of the GUI. When satisfied with the shape that has been assembled during the neighbor discovery phase, a connector is attached to one of the modules in the structure so that the computer and the module can communicate. This module is termed the root module. The GUI is instructed to send an initial localization message to the root. This message tells the root that it is located at coordinates $(0, 0, 0)$. This initial message is illustrated in Figure 5-2(a) by the single arrow with coordinates $(0, 0, 0)$.

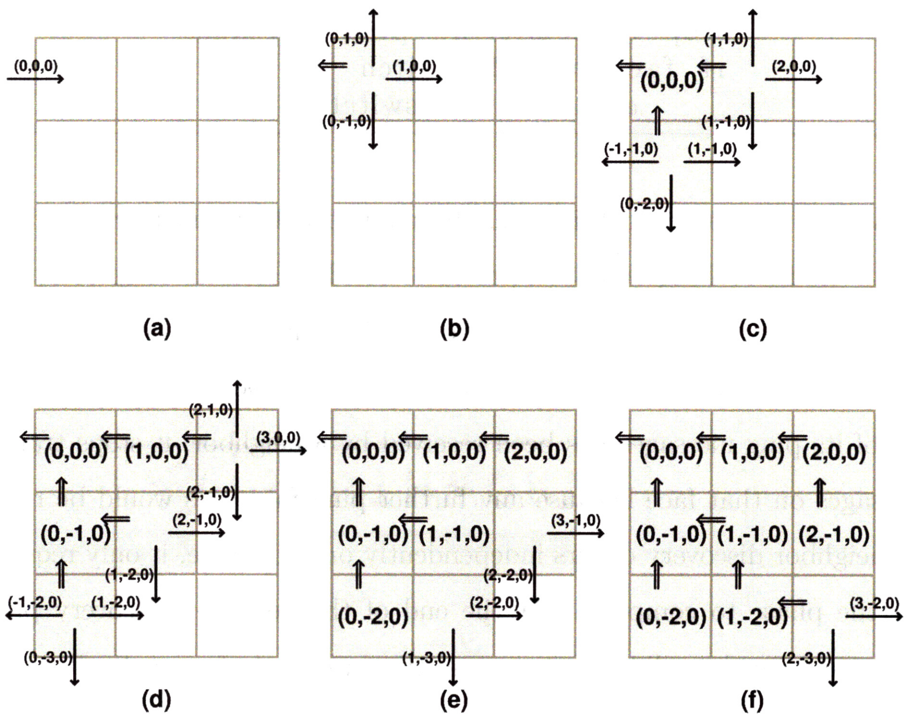


Figure 5-2: Localization messages, represented by the single width arrows, propagate from one module to the next and carry the location of the receiving module. Once a module is localized, it transmits a reflection message, represented by double arrow, to its parent and it eventually propagates back to the root module.

Once a module has received a localization message, it knows its position in the structure. Using its position, it can calculate the positions of all of its immediate neighbors by simply incrementing or decrementing the individual coordinates in the triplet that represents its location. For example, the module that lies to the right of the root is located at $(1, 0, 0)$; the module to the left: $(-1, 0, 0)$. Once the root has determined the coordinates of its neighbors, it sends an appropriate localization message to each. Each message contains the correct address of the neighbor to which it is transmitted. The messages sent by the root to its neighbors are shown in Figure 5-2(b) by the single arrows and their associated triplets.

This process continues unaltered until localization messages reach the farthest extents of the structure. Figures 5-2(c-f) show localization messages, (represented by single arrows), propagating through the remainder of the 3-by-3 structure. The only difference between how the root handles the localization message it receives from the GUI and how all other modules handle localization messages is that the other modules do not transmit a localization message back to the module from which they received the message. Such backward transmissions would be redundant because any module which transmits a localization message already knows its position. Additionally, no module, even if its receives another localization message, retransmits another set of localization messages to its neighbors.

5.2.2 Reflection Messages

Once a module is localized, it needs some way of informing the GUI that it exists. To do so, every module transmits a reflection message to the GUI after it is localized. Specifically, the modules only transmit the reflection messages on their parent faces. A module's parent face is defined as the first face on which it received a localization message. Figure 5-2 denotes each module's parent face using a double arrow. These pointers always point from a module to its parent. The figure illustrates the fact that a module's parent pointer always points toward the neighboring module from which it first received a localization message.

In addition to initiating the transmission of a reflection message on its parent face, each module also forwards any reflection messages that it receives from its neighbors on to its parent face. Eventually, all reflection messages propagate back to the root module and from there to the GUI. Such a process, as will be shown in Section 5.2.3, guarantees that the GUI receives a reflection message from each module in the structure. In other words, starting from any module in Figure 5-2(f), one can trace a path back to the root module by following the modules' parent pointers.

Listing 5.2: The localization algorithm ensures that every module determines its position and that each informs the GUI in turn.

```
wait for localization message
when message with position L is received on face F
    Localized = TRUE
    Parent = F
    Location = L

    calculate locations of neighbors
    for each face , except Parent
        transmit localization message
    transmit reflection message to Parent
    ignore future localization messages
return
```

5.2.3 Correctness of the Localization Algorithm

Listing 5.2 illustrates with pseudocode the algorithm that each module uses to localize. To show that the localization algorithm operates correctly, we first show that each module receives a localization message. Then we show that these messages all contain the correct position information. Finally, we show that the algorithm terminates.

Theorem 1 *The localization algorithm ensures that each module determines its position in the structure in a finite amount of time.*

Proof: By assumption, we know that the root module receives a localization message because we use the GUI to send that message. Because the root retransmits the message to its neighbors, we know that all of the root's neighbors receive a localization message. The neighbors also retransmit the message. By induction, after k iterations, all modules which are at most k units away from the root along any contiguous path are localized.

Now we show that every localization message received by any module correctly identifies the module's position. (In reality, we only need to show that the first message is correct because the pseudocode ignores all localization messages after the first.) Again, by assumption, we know that the root is correctly localized because we use the GUI to tell the root module its position. Because of the symmetric and independent way in which the coordinates of each localization message are modified as the messages propagates, (incrementing the x–coordinate when a message is passed to a module's right and decrementing the x–coordinate when a message is passed to a module's left, etc.), no matter which path a localization message follows from the root to any other module, the final coordinates contained in the message when it reaches that module will be identical. Therefore, each localization message received by any module will contain the same set of coordinates and the module will localize correctly.

Finally, the localization algorithm terminates because each module only responds to the first localization message that it receives. Once all modules in the structure have received a localization message, no additional localization messages will be sent. ■

Listing 5.3 contains the pseudocode which forwards reflection messages to a module's parent. This pseudocode, combined with the localization pseudocode in Listing 5.2, demonstrates that all reflection messages eventually reach the root module.

Theorem 2 *All reflection messages reach the root module.*

Proof: After localization, some neighbor of the root must have a valid parent pointer that points to the root. (If this were not the case, localization messages could not have propagated to any other module in the system.) For the remainder of this proof, a valid parent pointer is one which points to a cube which already has a valid parent pointer. The root's neighbors, like all other modules, do not transmit localization messages to their neighbors until they have a valid parent. As a result, any localization message that another module receives originates from a module with a valid parent. By induction, all parent pointers must be valid, and they must eventually lead to the root. ■

Listing 5.3: Modules must forward reflection messages from their neighbors on to their parent to ensure that all reflection messages eventually reach the root module.

```
loop forever
    if reflection message received
        if Parent != NULL
            forward reflection message to Parent
```

5.2.4 Running Time of the Localization Algorithm

For the purposes of analyzing the running time of the localization algorithm, we place an upper bound on the amount of time required by a module to process any messages that it has received and produce outgoing messages in response. We denote this upper limit on a module's processing time t.

Theorem 3 *The running time of the localization algorithm is $O(nt)$.*

Proof: If there are n modules in a system, the running time of the localization algorithm is $O(nt)$ because the modules could form an n–unit chain, and each module could require time t to forward the localization message. Therefore, the time for the localization messages to reach the end of the chain is $O(nt)$. ■

Theorem 4 *The running time of the localization algorithm is $O(mt)$, where m is the length of the longest of the set of shortest paths from the root module to any other.*

Proof: While localization messages may propagate to a module through a path longer than k units in a shorter amount of time, localization messages will always propagate to the module along a path of length k in $O(kt)$ time. Likewise, localization messages will propagate to each module, i, in $O(k_i t)$ time if k_i is the length of the path from the root module to the i–th module. Therefore, if f is the module farthest away from the root, in particular, $k_f = m$ units away, all localization messages will be received in $O(mt)$ time. ■

Theorem 5 *The time required for all reflection messages to propagate back to the root module is $O(nt)$.*

Proof: Given n modules, the time required for all localization messages to return to the root is $O(nt)$ because all n modules could form a single chain. The reflection message from the module at the far end of the chain would have to pass through all $n-1$ other modules, each of which could require time t to process the message. ■

Theorem 6 *The bound on the time required for all reflection messages to return to the root cannot be reduced to $O(mt)$, where m is the length of the longest of the set of shortest paths from the root to any other module.*

Proof: We cannot claim an $O(mt)$ bound on the time it takes for reflection messages to return to the root because the chain of parent pointers may be longer than m, the longest of the set of shortest paths from the root module to any other. The chain of parent pointers may be longer than m because some chain of modules may process localization messages quickly, leading to a situation where a module that is k units away from the root receives its first localization message from a module that is $k+1$ units away from the root. In such a situation, the reflection message would need to travel through at least k other modules before reaching the root. ■

5.3 Inclusion Message Generation

After localization is complete, the GUI is utilized to select which modules are to be included in the final structure and which should detach. By default, every module assumes that it is not a part of the final shape. Only by receiving an inclusion message does a module learn that it is destined to be a part of the final configuration. Once the virtual sculpting process is complete, the GUI must generate a sequence of inclusion messages. The remainder of this section is devoted to explaining the algorithm used to do so. Before continuing, it is important to note that both the algorithm executed by the GUI and the shape distribution algorithm presented in Section 5.4 assume that the root module is a part of the final structure and that all modules in the final structure are connected. In other words, there is a contiguous path from the root module to every other module included in the final structure.

Listing 5.4: The message generation algorithm uses a BFS to find the shortest path from the root to all modules and then uses a DFS to generate the sequence of messages to be transmitted.

```
make graph w/ vertex for each module of final struct.
perform BFS to find shortest paths
perform DFS to generate message sequence
```

Message generation can be divided into the three steps that are seen in Listing 5.4. First, as explained in Section 5.3.1, the algorithm constructs a graph that contains information about the assembled structure of modules. Second, the algorithm performs a breadth first search (BFS) on the graph to find the shortest distance between the root module and all others. The BFS is detailed in Section 5.3.2. Finally, Section 5.3.3 illustrates how the algorithm uses a depth first search (DFS) to traverse the graph and produce a set of inclusion messages.

5.3.1 Constructing the Graph

The first step in generating the inclusion messages is to generate a graph, $G(v, e)$, in which every module of the final configuration is a vertex. Then, edges are added between vertices whose corresponding modules have touching faces. An example is shown in Figure 5-3. Part (a) of the figure shows the shaded modules that should be included in G. Figure 5-3(b) shows the first step in the construction of G: vertices have been added for every module that is a part of the final structure. Figure 5-3(c) shows G once complete: edges have been added between all nodes whose corresponding modules are neighbors.

After the final configuratin of the system is modeled, the time to construct G is $O(n)$, where n is the number of modules that will be a part of the final structure. For each of the n modules, the algorithm must insert up to six edges in G, one for each neighbor that is also included.

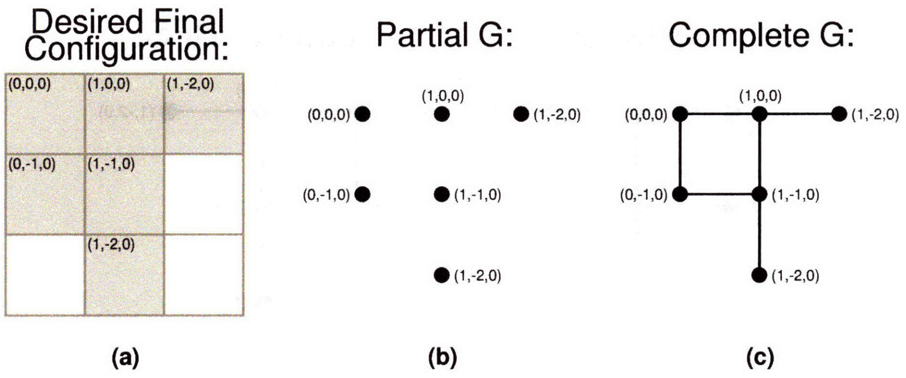


Figure 5-3: The desired final configuration of a structure of 9 modules is shown in (a) where shaded cubes represent modules that should be included in the final structure. As shown in (b), the first step in generating a sequence of inclusion messages is to construct a graph which contains a vertex for every module in the final structure. Edges, inserted in (c), represent the face that modules are neighbors.

5.3.2 Finding Shortest Paths

After the graph, G, of modules in the final structure is determined, the algorithm performs a BFS on G to determine the shortest path between the root vertex and all other vertices. A BFS produces the shortest paths because all paths have unit length [2]. The BFS modifies G so that it becomes a breadth–first tree. Any edge in G that is not part of a shortest path from the root module to any other module is eliminated. We want to find the shortest paths between the root and all other nodes because these paths are the sequence of modules that the inclusion messages should follow. If each inclusion message follows the shortest path between the root and its destination module, the shape distribution algorithm, discussed in Section 5.4, will be as efficient as possible.

Figure 5-4 shows how the initial G is transformed into a breadth–first tree by the BFS. The edge between the modules at positions $(0, -1, 0)$ and $(1, -1, 0)$ is eliminated in the breadth–first tree because moving from the root, $(0, 0, 0)$, to the module at $(1, 0, 0)$ and then to the module at $(1, -1, 0)$ provides an equally short path from the root to the module located at (1,-1,0) as following the path through $(0, -1, 0)$.

The typical BFS algorithm executes in $O(V+E)$ where V is the number of vertices

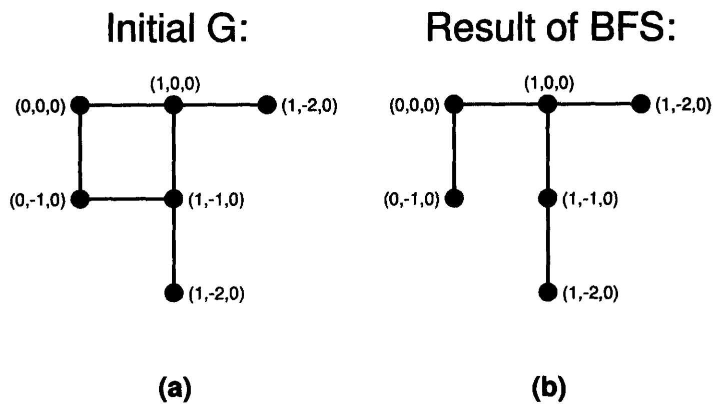


Figure 5-4: Performing a breadth–first search on G, the initial adjacency graph in (a) produces the breadth–first tree shown in (b). The search eliminates any edge which is not a part of the shortest path from the root module to any other.

and E is the number of edge in the graph [2]. In our system, the number of edges is never more than six times the number of vertices because each module has only six faces. Therefore, the running time of the BFS is $O(n)$, where n is the number of modules included in the final structure.

5.3.3 Generating the Inclusion Messages

Once the message generation algorithm has constructed a breadth–first tree, G, of all modules that are a part of the final structure, it performs a DFS on G starting at the root to determine the order in which the modules will be notified that they are a part of the final structure. Because G is already a tree, no additional edges are removed by the DFS. Instead, as the DFS progresses, it generates an inclusion message for each module as it is encountered for the first time. Table 5.3.3 shows one possible order in which the messages are generated from the breadth–first tree shown in Figure 5-4(b). The contents of these messages and the way they are distributed is addressed in Section 5.4.

The running time of the DFS algorithm when applied to an arbitrary graph is $\Theta(V + E)$ [2]. As discussed above, we can refine this bound to $\Theta(n)$, (where n is the number of modules in the final structure), because no vertex has more than six

Table 5.1: The inclusion message for a module is generated the first time a DFS encounters that module in the breadth–first tree. This ordering of inclusion messages was generated using the breadth first tree in Figure 5-4(b).

Order	*Module encountered*
1	$(0,0,0)$
2	$(1,0,0)$
3	$(2,0,0)$
4	$(1,-1,0)$
5	$(1,-2,0)$
6	$(0,-1,0)$

edges. The entire message generation process completes in $O(n)$ time because each step of the algorithm, (constructing G, performing the BFS, and walking down the tree using the DFS) requires $O(n)$ time.

5.4 Shape Distribution

This section is devoted to examining how the set of inclusion messages generated by the GUI is automatically distributed by the system of modules. As stated before, only by receiving an inclusion message does a module learn that it is destined to be a part of the final configuration. Also, recall that the final structure of modules is fully connected.

5.4.1 Inclusion Message Propagation

Each inclusion message carries two important pieces of information: a hop count and a branch direction. As the inclusion messages are distributed by the structure, a virtual chain of inclusion pointers is formed. The hop count field of each inclusion message dictates how far down this chain each message should travel. Once the message has reached the specified depth in the chain, it extends the chain by including the module specified by the branch direction. Listing 5.5 shows this algorithm implemented in pseudocode.

Listing 5.5: The shape distribution algorithm checks whether the hop count of the inclusion message is zerp. If it is the module assumes that it is a part of the structure. Otherwise the hop count is decremented and the message is forwarded along the inclusion chain

```
Included = FALSE
NextInIncChain = NULL

loop forever
    wait for inclusion message
    when msg. w/ H hops and branch dir. BD is rcvd.
        if H = 0
            Included = TRUE
            transmit reflection message to Parent
        elseif H = 1
            transmit inclusion message with ...
                     hops = 0 and ...
                     branch direction = BD ...
                     to NextInIncChain
            NextInChain = BD
        else
            transmit inclusion message with ...
                     hops = (H-1) and ...
                     branch direction = BD ...
                     to NextInIncChain
```

The algorithm operates as follows. It begins in lines 1-2 by assuming that the module is not included in the structure and that the module has no inclusion chain pointer. It then loops forever waiting for inclusion messages. When it receives an inclusion message, it chooses to perform one of three actions. The first option, listed on line 7 of Listing 5.5, occurs when the hop count of the received message is zero. This is an indication that the inclusion message was originally destined to include this module. As a result, the module now realizes that it is a part of the structure and transmits a reflection message back to the GUI. (Like the reflection messages transmitted in response to localization messages, it follows a chain of parent pointers to reach the root module.) The reflection message informs the GUI that the module was successfully notified of its status in the final structure.

The second case occurs when the hop count of the incoming message is one. This signals that one of the module's neighbors is the final destination of the message. The module determines which neighbor is the final destination of the message by examining the branch direction field of the inclusion message. The branch direction field indicates the face on which the neighbor that is supposed to be included is attached. The algorithm forwards a modified inclusion message whose hop count is zero to the neighbor specified by the branch direction field. In addition, as shown in line 15, the module updates its inclusion chain pointer to reflect where to forward the next inclusion message.

The third and final action is prompted by the receipt of an inclusion message in which the hop count is greater than or equal to two. In this scenario, the module should have already received at least two inclusion messages: one including the module itself, and another including one of its neighbors and assigning a valid direction to its inclusion chain pointer. When a module receives such a message, it decrements the message's hop count and forwards it along in the direction of the module's inclusion chain pointer.

Figure 5-5(a-d) illustrates the evolution of an inclusion message as it is forwarded from the root module, $(0, 0, 0)$, to the next module that should be included in the structure $(1, -2, 0)$. In the figure, the message is represented by the straight arrow.

One can observe the hop count decreasing as the message passes farther down the existing inclusion pointer chain, which is represented by the arced arrows. When the message reaches the neighbor of new module in Figure 5-5(c), it causes that module, $(1,-1,0)$, to update its inclusion chain pointer. The result is seen in Figure 5-5(d).

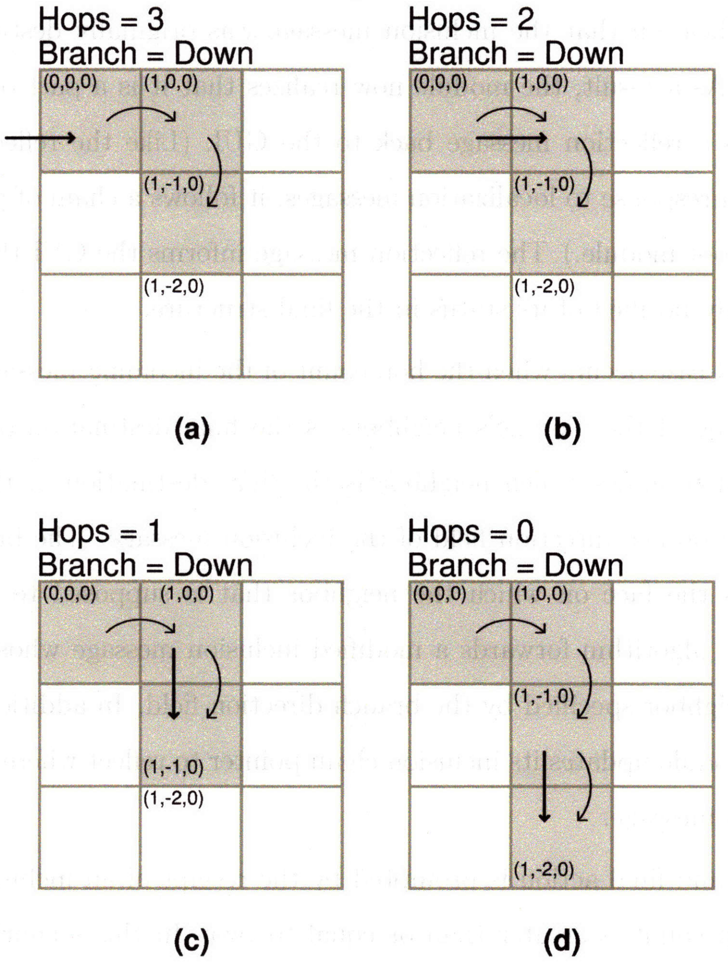


Figure 5-5: An inclusion message, represented by the straight arrow, progresses through a number of modules that, as denoted by their shading, have already been included in the structure. As the message propagates, it follows the arced inclusion chain pointers until it reaches the module at position (1,-1,0) shown in (c). At this point, the branch direction (down) directs the module to forward the message to its downward neighbor and update its own inclusion chain pointer.

The progression of several inclusion messages that are destined for different modules is pictured in Figure 5-6. In this structure, modules with addresses $(0,0,0)$,

$(1,0,0)$, $(1,-1,0)$, $(0,-1,0)$, $(2,0,0)$, and $(1,-2,0)$ are supposed to be included in the final structure. The six inclusion messages needed to notify the six modules of their destiny are injected into the system at the root module, $(0,0,0)$. From there, they propagate to their destination modules, which are shaded after they know that they are supposed to be a part of the final structure.

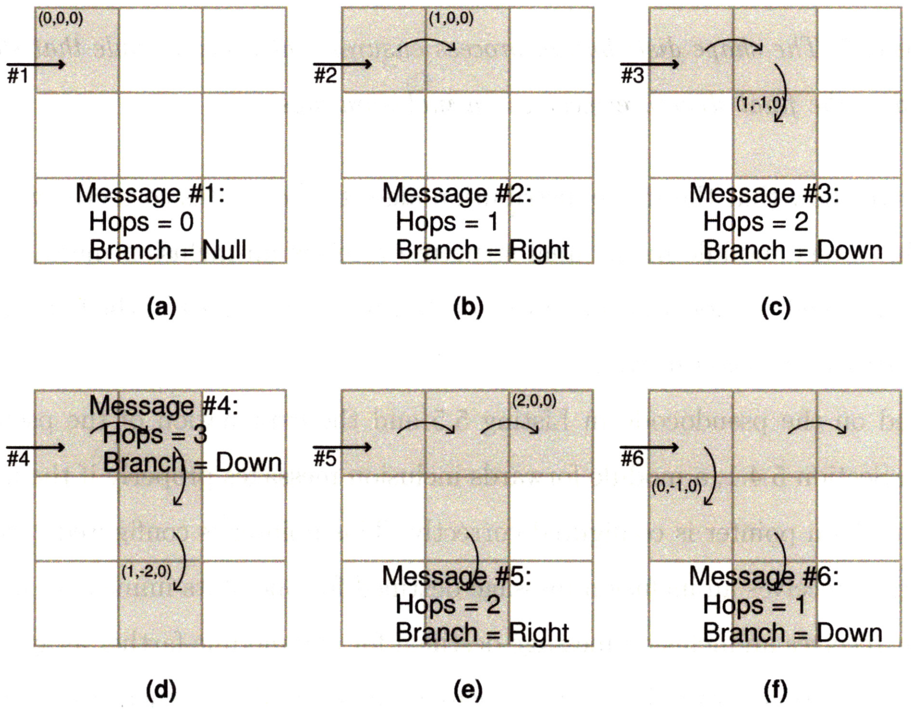


Figure 5-6: Six inclusion messages are transmitted by the GUI to the root module in order to include six modules in the final structure. The messages propagate by following the arced inclusion chain pointers until they reach their destinations. The branch direction and initial hop count for each message is indicated. Once a module knows that it is part of the final structure, it is shaded. Sometimes, as shown in (e), some inclusion chain pointers remain intact even after the active chain has shifted away.

5.4.2 Shape Distribution Correctness

Whether or not the shape distribution algorithm operates correctly depends on how the inclusion messages are generated by the GUI. As discussed in Section 5.3, the messages are generated by a DFS. The first time that the DFS algorithm encounters an included module, it generates that module's inclusion message. In the process, it keeps track of the newest module's depth in the tree and the branch direction from the previously encountered module which needs to be taken in order to arrive at the newest module. These two pieces of information are embedded in the inclusion message destined for the new module.

Theorem 7 *The shape distribution process ensures that every module that should be included in the final structure receives an inclusion message.*

Proof: To show that the shape distribution algorithm operates correctly, we need to show that every module in the final structure receives an inclusion message. Additionally, we need to show that modules not destined to be a part of the final structure do not receive inclusion messages.

Based on the pseudocode in Listing 5.5 and the explanation of the pseudocode given in Section 5.4.1, a module forwards inclusion messages properly if the module's inclusion chain pointer is configured correctly. This pointer is configured correctly if the module receives an inclusion message destined for one of its immediate neighbors before it receives an inclusion message destined for any module farther away from the root. In fact, this is exactly what happens because the DFS generates an inclusion message the *first* time that it encounters each module. The inclusion messages for modules past the current one are generated later. As a result, a module will always have a valid inclusion chain pointer before it needs to forward inclusion messages to modules other than its neighbors. This means that the inclusion messages are always forwarded correctly and that each module that should receive an inclusion message does.

Now that we know that every module in the final structure receives an inclusion message, it is easy to see that no other modules do. This is based on the fact that the message generation algorithm only generates one inclusion message for each module in the final structure. If all modules that are supposed to be part of the final structure receive inclusion messages, there are no additional inclusion messages that could be received by the other, soon to be discarded, modules. Therefore, the shape distribution algorithm operates correctly. ■

5.4.3 Running Time of the Shape Distribution Algorithm

As with the localization algorithm, we assume an upper bound, t, on the amount of time required by any module to process and produce a response to a received message. Also, assume that n modules are included in the final structure.

Theorem 8 *The running time of the shape distribution algorithm is $O(nt)$.*

Proof: Each of the n modules in the final structure requires a separate inclusion message to be transmitted from the GUI to the root module, where it will be distributed. Fortunately, the GUI does not have to wait for one inclusion message to reach its final destination before sending another. Instead, time t after sending the first inclusion message to the root, we know that the root has finished processing the message and can now accept another. After another t units of time, the root has passed the second inclusion message on to one of its neighbors and can accept a third message. The same constraints also apply to all other modules in the system, so the system can accept one new inclusion message every t units of time. Therefore, the system requires $O(nt)$ time to include all n modules in the final structure. ■

As explained in Section 5.2.4, the time required for all n reflection messages generated in response to the inclusion messages to return to the root module is also $O(nt)$.

5.5 Disassembly

Once all modules which need to be included in the final structure have received inclusion messages, the system is ready to disassemble. The disassembly process is initiated when the GUI is instructed to send a disassemble message.

Listing 5.6: The disassembly algorithm propagates any received disassembly message on all faces and only instructs the module to disconnect from the structure if it has not received an inclusion message.

```
wait for disassemble message
when message is received on face F
     transmit acknowledgment on face F
     for every face except F
          retransmit disassemble message
     if module not included in structure
          for each face with a Magswitch
               disconnect the Magswitch
          for all other faces
               transmit disconnect request to neighbor
```

5.5.1 Disassemble Message Propagation

Pseudocode for the disassembly algorithm is given in Listing 5.6. When a module receives a disassemble message, it transmits an acknowledgment to the transmitter and forwards the disassemble message to all of its other neighbors. Modules that are included in the final structure perform no further tasks after forwarding the disassemble message. In comparison, modules that are not a part of the final structure attempt to disconnect. First, they deactivate all three of their Magswitches. Second, for any face lacking a Magswitch that is attached to a neighbor, they send a disconnect request message to that neighbor. If the neighbor has not already deactivated the correct Magswitch, it does. Otherwise, the neighbor ignores the disconnect request message. In any case, the module is now free to fall away from the remaining structure.

5.5.2 Correctness of the Disassembly Algorithm

Theorem 9 *The disassembly algorithm only disconnects those modules which should not be a part of the final structure.*

Proof: We need to show that disassembly messages reach each module and that only the specified modules disconnect. Based on the correctness of the localization algorithm discussed in Section 5.2.3, we know that disassemble messages reach all modules because the disassemble messages are propagated identically to localization messages. From Section 5.5.1, we know that once each module receives a disassemble message, it only disconnects from the structure if the module has not received an inclusion message. Therefore, the disassembly algorithm operates correctly. ■

5.5.3 Running Time of the Disassembly Process

As before, we assume an upper bound, t, on the amount of time required by any module to process and produce a response to a received message.

Theorem 10 *The running time of the disassembly algorithm is $O(nt)$.*

Proof: Once a disassemble message reaches a module, the time required for the module to disconnect from its neighbors is $O(1)$. Therefore, if there are n modules in the structure, the time for the disassemble messages to propagate to all modules is $O(nt)$. This worst case occurs when the modules form a single chain. ■

As with the localization algorithm, we can provide a tighter bound if we are able to determine m, the length of the longest of the set of shortest paths from the root module to all other modules. In this case, the amount of time required for the disassemble message to reach all modules is $O(mt)$. The proof follows that given in Section 5.2

Chapter 6

Algorithm Implementation

This chapter describes a variety of issues related to implementing the algorithms of Chapter 5 in hardware. A diagram showing the software heirarchy is presented in Figure 6-1. It shows how the different pieces of software are layered from the lowest level hardware drivers, (shown at the bottom), to the high–level self–disassembly algorithms, (shown at the top). Except for some small pieces of assembly code in the PSoC, all other algorithms were implemented in C. The implementation of the low–level software was already addressed Chapter 4. This chapter will focus on how the high–level software running on the ARM chip was implemented.

This chapter begins, in Section 6.1, by discussing three modifications we made to the high–level algorithms which drastically improve the system's performance by reducing the number of lost messages. Then, similar to Section 4.5, Section 6.2 presents the format of the inter–module messages that are employed by the high–level self–disassembly algorithms. Finally, Section 6.3 discusses the graphical user interface (GUI) which is used to visualize and control the system.

6.1 Increasing System Reliability

While Chapter 5 proves that the self–disassembly algorithms work in theory, the algorithms, as described in that chapter, sometimes fail in practice. The primary cause of failure is lost messages, which can occur for two reasons. First, these algorithms

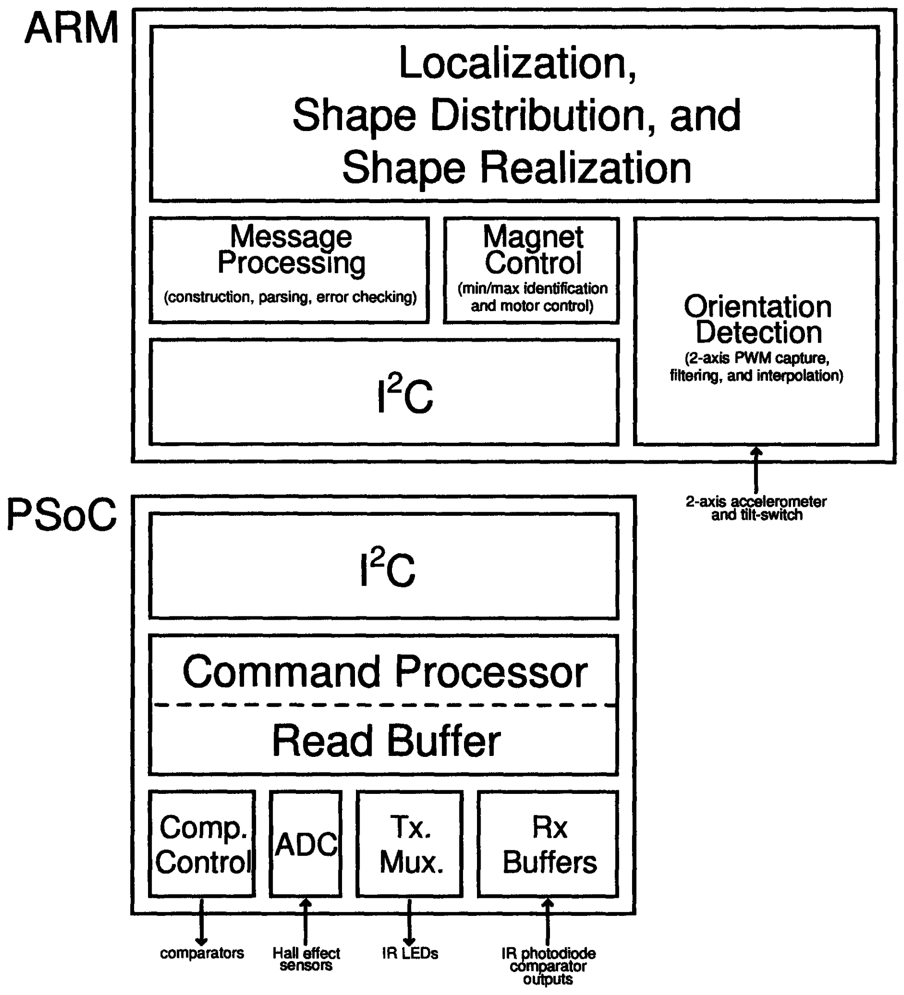

ARM
Localization, Shape Distribution, and Shape Realization
Message Processing
(construction, parsing, error checking)
Magnet Control
(min/max identification and motor control)
Orientation Detection
(2-axis PWM capture, filtering, and interpolation)
I2C
2-axis accelerometer and tilt-switch
PSoC
I2C
Command Processor
Read Buffer
Comp. Control
ADC
Tx. Mux.
Rx Buffers
comparators
Hall effect sensors
IR LEDs
IR photodiode comparator outputs

Figure 6-1: The software heirarchy is divided horizontally into a number of functional blocks. The blocks depicted lowest in the diagram correspond to the most basic functionality and provide abstractions of the hardware that can be used by the high–level algorithms running in the blocks near the top of the diagram.

sometimes fail to successfully transmit messages to the modules' neighbors. This type of failure is discussed in Section 6.1.1.

Second, neighboring modules are sometimes unable to communicate because they were poorly aligned. While we could attempt to carefully align the modules as we assembled them, imperfections in the manufacturing process make it difficult to ensure that every LED/phototransistor pair was able to communicate. For example, one module that is not perfectly square can affect the alignment of several neighboring modules. To account for misalignment, we implemented two modifications to the self–disassembly algorithms. Section 6.1.2 details how we made the localization process robust to misalignment. Then, Section 6.1.3 explains how we modified the way inclusion messages are distributed by appending additional information to all reflection messages. This resulted in a more reliable shape distribution phase.

6.1.1 Synchronization

We found that if a module was attempting to send several messages to a neighbor in quick succession, that some of those messages were lost. We traced this problem back to the way in which the transmission buffers were implemented, as discussed in Section 4.3.3. When one of the high–level routines transmits a message on a specific face, the new message overrides any message that was already buffered for transmission on that face. Therefore, if the message that was already pending in the buffer has not yet been received by the neighboring module, it is lost. As a result, if a module is busy communicating with many of its neighbors, some of those neighbors, in their hurry to transmit messages to the module, may drop some messages.

To rectify this problem, we synchronized the message passing process. When a module receives any high–level message that has a specific destination, (e.g., a shape distribution or reflection message), it does not send an acknowledgment immediately. Instead, it first checks to see if the transmission buffer for the face on which the message should be retransmitted is empty. If the buffer is empty, the algorithm handling the message transmits an acknowledgment back to the sender and then places the new message in the transmission buffer of the appropriate face. If the needed transmission buffer is not empty, the module never transmits an acknowledgment back to the sender. As a result, the sender continues transmitting the message until the needed transmission buffer on the receiving module is available. As a result of this synchronization, messages that have specific faces over which they need to be forwarded, never override any message already being transmitted on that face. In a common scenario, this prevents one inclusion message from overriding another.

All of the lower–level messages presented in Section 4.5, (except acknowledge and not acknowledge messages), in addition to ping, localization, and disassemble messages automatically override any message already in a face's transmission buffer because they do not have specific destinations. This is acceptable because these messages fall into one of two categories. They either modify system–wide properties of the structure, or else they are not used while system–critical messages, (like inclusion and reflection messages), are being propagated. Table 6.1.1 shows into which of the two categories each type of messages falls.

Table 6.1: All messages that do not need to be transmitted on a specific face fall into two categories. Either they modify system–wide properties that invalidate the data contained in any message that must be retransmitted on a certain face, or they do not propagate through the system at the same time as messages that need to be retransmitted on a certain face.

Message Type	*System–wide Mod.*	*Not Prop. Concurrently*
Disconnect All (`DCA`)	Yes	No
Disconnect Request (`DRQ`)	No	Yes
Magswitch State (`MSS`)	No	Yes
Reset (`RST`)	Yes	No
Real Time Clock (`RTC`)	Yes	No
Ping (`PNG`)	No	Yes
Localization (`LOC`)	No	Yes
Disassemble (`DAS`)	Yes	Yes

It is acceptable for messages which modify system–wide properties to override destination–specific messages because they make irrelevant any information contained in a destination specific message. For example, a reset message can safely override any inclusion or reflection message because when the system is reset, it forgets any information that it has amassed through the passing of inclusion and reflection messages. Likewise, we never need to consider some messages overriding destination specific messages because those messages are not presented in the system during the same time periods. For example, a module finishes passing all localization messages before it begins to pass reflection messages. As another example, disassemble messages are only injected into the system after all inclusion and reflection messages have been sent. In practice, the synchronization system described in this section effectively eliminates scenarios in which messages are lost because they are overwritten while in a transmission buffer.

6.1.2 Localization Modification

In the localization algorithm presented in Section 5.2, the modules assume that the first face over which they receive a localization message is their parent face. This is problematic because the ability to receive inter–module messages on a specific face does not correlate with the ability to successfully send messages on the face. Using the algorithms presented in Chapter 5, modules had parent faces on which they were unable to transmit, (or the parent was unable to receive). Some modules were able to localize, but they were unable to tell the GUI of their existence because they could not transmit reflection messages, (which follow a sequence of parent pointers), back to the GUI. This type of error affects more than just the modules that are unable to successfully transmit information to their parents. It also affects any module whose chain of parent pointers passes through another module that cannot communicate with its parent.

To rectify the problem, we modified how modules localize. Instead of blindly accepting the first neighbor that transmits a localization message as its parent, each module first checks whether two–way communication with that neighbor is possible. This check does not require any additional messages; if two–way communication is possible, the module should have received an acknowledgment to one of the ping messages it sent during the neighbor discovery phase. If the module never received an acknowledgment of any of its pings, it assumes that two–way communication with the neighbor is impossible and waits for another neighbor, with whom two–way communication is possible, to transmit a localization message.

This modification has two effects on the system. First, some modules, even if they can receive localization messages, may not localize. This does not affect the system's functionality because those modules would be unable to inform the GUI of their presence even if they did localize. If the GUI does not know about a module's existence, it does not matter if the modules are localized or not because they would be uncontrollable anyway.

Second, the modification to the localization algorithm affects its running time. It is no longer reasonable to assume that localization messages can flow along the shortest path from the root module to any other. Even if the structure is not a single chain, some localization messages may have to travel through all n modules in the structure before reaching the final module. We can no longer claim that, if m is the length of the longest of the set of shortest paths from the root module to any other, the running time of the algorithm is $O(mt)$. Instead, the running time of the localization algorithm is $O(nt)$, where t is the maximum time required by any module to process a message and generate a response.

6.1.3 Shape Distribution Modification

The localization process is not the only aspect of the high–level self–disassembly algorithms which can be affected by communication failures. If a module cannot transmit or receive on specific faces, the shape distribution process may be impeded. The original message generation algorithm presented in Section 5.3 assumes that modules can reliably communicate with all of their neighbors. As a result, it routes inclusion messages through the faces which lie along the shortest path from the root to any module included in the final system configuration. Unfortunately, these paths may sometimes cross faces which are unable to communicate with their neighbors.

To operate successfully, the message generation algorithm needs to know which faces can successfully communicate and which cannot. To transmit this information to the GUI, we modified the reflection messages. In addition to the transmitter's location and user identifcation (UID), the reflection messages were modified to include a field that indicates with which neighboring modules the sender can successfully communicate. The GUI uses this information when constructing the graph, G, of all modules which are supposed to be included in the final structure. If a module that is supposed to be in the final structure cannot communicate with a neighbor that is also a part of the final structure, no edge is inserted between them.

Accounting for missing communication links makes the system more reliable and does not increase the theoretical running time of the shape distribution algorithm because one inclusion message can still be transmitted from the GUI to the root every t time units, the maximum amount of time required by any module to process a message and generate a response. It is the case that some inclusion messages will have to travel longer than optimal paths in the structure to reach their destinations. This implies that some messages will travel through more modules than they did in the optimal scenario. Therefore, the message load on some modules will increase, and t may increase in turn.

6.2 Message Specification

This section details the specifics of the inter–module messages used by the algorithms presented in Chapter 5. All messages follow the basic format presented in Figure 4-4. They have a start character, data fields separated by ampersands, a checksum, and a termination character. For an explanation of all other inter–module messages, one should consult Section 4.5.

6.2.1 Ping (`PNG`) Messages

Ping (`PNG`) messages, such as the one illustrated in Figure 6-2, are transmitted on all faces of a module after it is reset in an attempt to detect neighboring modules. If a module receives a `PNG` message, it assumes that it has a neighbor on the face that received the message and transmits an `ACK` in response. Additionally, if the module has a Magswitch on any face that receives a `PNG` message, it activates that Magswitch. Because `PNG` messages are simply used to detect a module's neighbors, they do not contain any type specific fields.

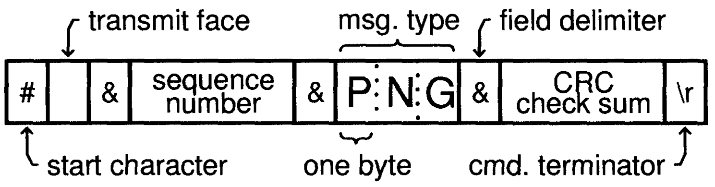


Figure 6-2: Ping (PNG) messages contain no type specific fields because they are only used to detect the presence of a module's neighbors. If a module receives a PNG message, it replies with an ACK.

6.2.2 Localization (LOC) Messages

The first localization (LOC) message is transmitted by the GUI to localize the root module. From there, localization messages radiate outward. Each localization message uses three type specific fields to hold the x, y, and z coordinates of the receiving module. When a module receives an LOC message, and if it has a two–way communication channel with the transmitter, it transmits an ACK message in return, followed by a reflection message. The module is now localized, and its parent is the module which transmitted the localization message. Now that the module is localized, it transmits LOC messages to all of its neighbors. For additional details of the localization algorithm, consult Sections 5.2 and 6.1.2.

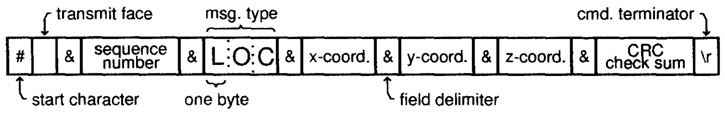


Figure 6-3: Localization (LOC) messages contain three type specific fields which specify the x, y, and z coordinates of the module which is destined to receive the message.

6.2.3 Reflection (REF) Messages

Reflection (REF) messages are transmitted in response to a module localizing or receiving an inclusion message. REF messages notify the GUI that the transmitter exists or that it has received an inclusion message. There is no algorithmic motivation for transmitting REF messages in response to the receipt of an inclusion message, but they allow the GUI to display which modules have received inclusion messages and which have not. REF messages are always transmitted on a module's parent face. This ensures that every REF message eventually propagates back to the root module. From there, it is transmitted to the GUI.

REF messages have five type specific fields that can been seen in Figure 6-4. The first three fields contain integer values and specify the transmitter's absolute location in the structure. The next field, which is also integer valued, contains the transmitting module's unique identification, (UID). The UID, while not used by any of the high–level algorithms, is helpful when debugging the system. The last type specific field is a list of all neighboring modules with which the transmitting module can communicate. This list is up to six digits long—one digit for each potential face. The order of the digits is unimportant.

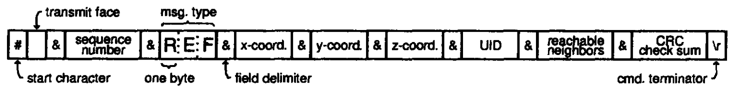


Figure 6-4: Reflection (REF) messages are used to inform the GUI of the structure's configuration. Each REF message contains five type specific fields. The first three type specific fields contain the x, y, and z coordinates of the transmitter, respectively. The fourth field contains the module's unique identification (UID). The last field is a list of all neighboring modules with which the transmitter can communicate.

6.2.4 Inclusion (INC) Messages

Inclusion (INC) messages are used by the shape distribution algorithm to include individual modules in the final structure. They are generated by the GUI and propagate down a virtual chain of inclusion pointers in the final structure. Each additional INC message transmitted down the chain extends the chain in some direction and includes the module which forms the new link of the chain in the final structure. When a module receives an INC message, it either determines that it is now a part of the final structure, or, if it already knows this, it forwards the message along to one of its neighbors. INC messages include two type specific fields. The first, the hop count, specifies how far down the inclusion pointer chain the message should travel before branching off. Each time an INC message is forwarded, its hop count is decremented by one. The second type specific field is the message's branch direction which specifies in which direction the inclusion pointer chain should be extended when the message has reached the specified depth in the chain.

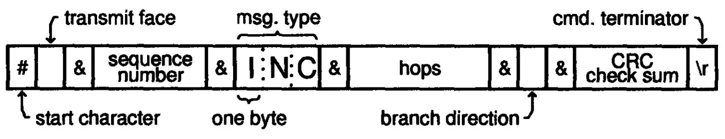


Figure 6-5: Inclusion (INC) messages contain two type specific fields. The first, the hop count, or hops, specifies how many modules down the inclusion pointer chain the message should travel before branching off. The second field, the branch direction, determines in which direction the inclusion should branch off the inclusion pointer chain once it has reached the specified depth.

6.2.5 Disassemble (DAS) Messages

A disassemble (DAS) message, such as the one shown in Figure 6-6, is transmitted by the GUI to the root module to start the disassembly process. Each module to receive a DAS message propagates the message on all faces. If a module is to be a part of the final structure because it has previously received an inclusion message, it performs no other actions after retransmitting the DAS message. In comparison, when a module that is not part of the final structure receives a DAS message, it disconnects from the structure. The simple task of informing all modules that they should begin the disassembly process does not require that DAS messages contain any type specific fields.

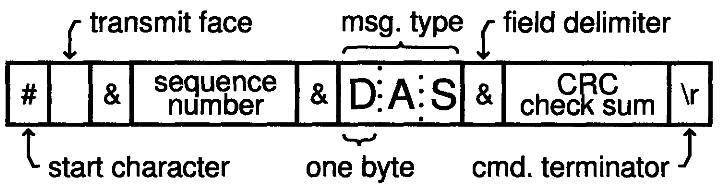
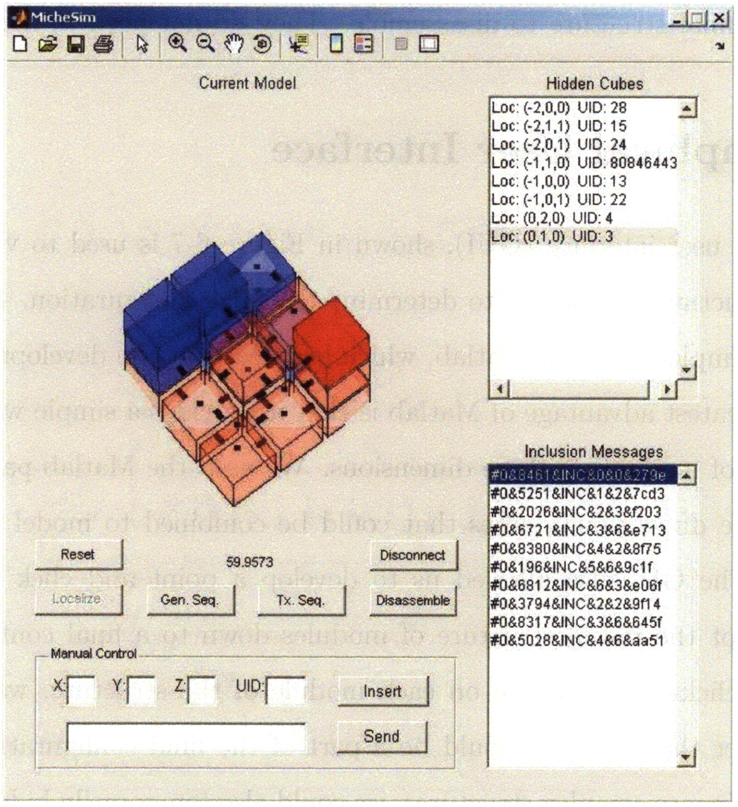


Figure 6-6: Disassemble (DAS) messages are used to tell the modules which are not a part of the final structure to disassemble. They do not need to include any type specific fields.

6.3 Graphical User Interface

The graphical user interface (GUI), shown in Figure 6-7 is used to virtually sculpt the initial structure of modules to determine the final configuration. For simplicity, the GUI was implemented in Matlab, which ensured that the development time was short. The greatest advantage of Matlab is that it provides a simple way to visualize the structure of modules in three dimensions. We used the Matlab `patch` command to create three dimensional cubes that could be combined to model more complex structures. The GUI also allowed us to develop a point–and–click interface with which to sculpt the initial structure of modules down to a final configuration. By simply right–clicking the mouse on each module of the structure, we were able to choose whether that module should be a part of the final configuration. To aid in the creation of more complex structures, we could also temporarily hide modules near that outside of the structure in order to sculpt modules that were otherwise obscured.

As mentioned in Chapter 5, the GUI generates the sequence of inclusion messages that is sent to the structure of modules over the serial port. The GUI code responsible for transmitting messages over the serial port is similar to the transmission buffers on the modules. The code, once supplied with a message to transmit, uses a timer to automatically retransmit the message until an acknolwedgment is receieved or a time–out occurs. These retransmissions occur without the intervention of the main program. As per the synchronization proceedure presented in Section 6.1.1, the GUI waits to transmit any inclusion message until the previously transmitted message has been acknowledged by the root module. This ensures that that the root module is



Hidden Cubes

```
Loc: (-2,0,0)  UID: 28
Loc: (-2,1,1)  UID: 15
Loc: (-2,0,1)  UID: 24
Loc: (-1,1,0)  UID: 80846443
Loc: (-1,0,0)  UID: 13
Loc: (-1,0,1)  UID: 22
Loc: (0,2,0)  UID: 4
Loc: (0,1,0)  UID: 3
```

Inclusion Messages

```
#0&8461&INC&0&0&279e
#0&5251&INC&1&2&7cd3
#0&2026&INC&2&3&f203
#0&6721&INC&3&6&e713
#0&8380&INC&4&2&8f75
#0&196&INC&5&6&9c1f
#0&6812&INC&6&3&e06f
#0&3794&INC&2&2&9f14
#0&8317&INC&3&6&645f
#0&5028&INC&4&6&aa51
```

Figure 6-7: A graphical user interface (GUI) was used to virtually sculpt the initial configuration of modules into a more interesting configuration. Here, modules that will be included in the final structure are shown in red, and those that will not be included are shown in blue. The list box in the lower right displays the sequence of messages that will be transmitted to the root module (the darkest red cube) for distrubtion in the structure.

not overrun with more messages than it can successfully handle.

Matlab also automatically processes messages received from the root module. We attached a callback function to the serial port so that every time a message termination character is received, Matlab calls a routine which parses and handles the new message. By using this callback, the GUI does not have to poll the serial port continuously. We found that Matlab does not deal with large volumes of serial data well. In particular, Matlab garbles the reception of or completely misses a large precentage of the messages from the root module. In effect, this symptom retards the rate at which Matlab can process messages. The problem is exacerbated when Matlab is also transmitting messages to the root. Because the root module retransmits each message many times, the fact that Matlab misses some of the messages does not threaten the GUI's effectivness. It does, as discussed in Section 7.4, seem to affect the execution speed of the shape distribution algorithm.

Chapter 7

Experimental Results

To test the algorithms of Chapter 5 and the modifications that we proposed in Chapter 6, we conducted 191 experiments. These experiments involved measuring the success rate and running time of the algorithms. To measure the running time of each algorithm, we used a timer built into the GUI which starts counting in fractions of a second after the GUI is told to begin the localization or shape distribution process. After either of these processes begins, pressing the an assigned key on the desktop computer's keyboard causes the GUI to record and display a split time. Pressing the GUI's localization or shape distribution buttons again halts the associated process and the timer. The total elapsed time is then displayed separately from the split time. The ability to record a split time is crucial in measuring the time required for all modules to receive localization and inclusion messages. In general, all modules receive these messages at least several seconds before their associated reflection messages propagated back to the GUI. The split time allows us to quantify the difference between the two times.

Following the same structure as Chapter 5, the experimental results associated with the neighbor discovery and localization phases are discussed in Sections 7.1 and 7.2, respectively. We show that the observed behavior of the algorithms correlates accurately with the behavior and running time predicted by the analysis in Chapter 5. Then, Section 7.3 demonstrates that the message generation algorithm associated with the GUI operates correctly. Next, the result of implementing the shape distribution algorithm on the hardware is presented in Section 7.4. The experiments presented in this section show that the observed running time of the shape distribution algorithm does not match the predicted running time. This section discusses why this may be the case. Finally, Section 7.5 presents a separate set of experiments that show the disassembly process operating correctly.

7.1 Neighbor Discovery Results

We tested the neighbor discovery process with modules that had anywhere from one to six neighbors. The neighbor discovery process worked correctly in all cases. When two modules were brought into close proximity with each other and aligned, they each detected the other's presence and latched together. Sometimes, if a module was poorly aligned with a neighbor, it failed to detect it. Shifting either of the modules slightly tended to fix this problem. Even if the modules were not adjusted, they did not compromise the system's overall reliability. Fortunately, it was never the case that a module was misaligned with all of its neighbors, so every module always had at least one mechanical and communication link to the rest of the structure. In all of the 191 experiments that we conducted, every single module successfully connected to the structure and prepared for localization. That means the neighbor discovery process operated correctly over 1,200 times.

7.2 Localization Results

In each of the 191 experiments that we performed, we tested the localization algorithm. We could observe the algorithm's progress in two ways. First, we were able to monitor the algorithm's progress by watching the user LED on each module. After a module localized, it began to flash its LED. Also, as specified by the localization algorithm, the modules transmit reflection messages after they are localized. When these messages propagate back to the desktop computer connected to the root module, the modules appear in the GUI. We were able to measure the amount of time required for the user LEDs on all modules to begin flashing and the amount of time required for all modules to appear in the GUI. We measured these times for linear, square, and cubic structures of different size. We were not able to measure the amount of time required for all user LEDs to begin flashing in a cubic structure because the LEDs of the modules in the middle of the structure were obscured by other modules. In these cases, we were still able to record how long it took before all modules appeared in the GUI.

The first experiment we performed measured the amount of time required for a line of modules to localize. We recorded the localization times for chains of modules that were one, four, seven, and nine units long when the root module was at the end of the chain. In the case of one module, we performed twenty experiments. Because a single module localizes so quickly, it was impossible to accurately resolve the amount of time required for the module's LED to begin blinking. We were able to accurately record how quickly a single module appeared in the GUI. We performed 16, 15, and 15 experiments for the 4-, 7-, and 9-module cases, respectively. Figure 7-1 illustrates the mean and standard deviation of the time required for all user LEDs to begin blinking. Likewise, Figure 7-2 presents the mean and standard deviation of the time required for all modules to be displayed in the GUI. Both figures show a linear relationship between the number of modules in the chain and the time required for all to localize and transmit their positions back to the GUI.

The next experiment that we carried out used 1-by-1, 2-by-2, 3-by-3, 4-by-4, and 5-by-5 square assemblies of modules. In this set of experiments, the root module was always chosen to be a corner module. We performed 20, 16, 17, 18, and 6 trials for the 1-, 4-, 9-, 16-, and 25-module squares, respectively. The time required for all modules to receive the localization messages and activate their LEDs is shown in Figure 7-3. The time required for all reflection messages to return to the GUI, where they appear as modules, is shown in Figure 7-4. In both figures, we have plotted the average times as circles, and the whiskers represent the standard deviations of the different experiments. Both plots, especially the second, demonstrate a linear relationship between the number of modules and the localization time.

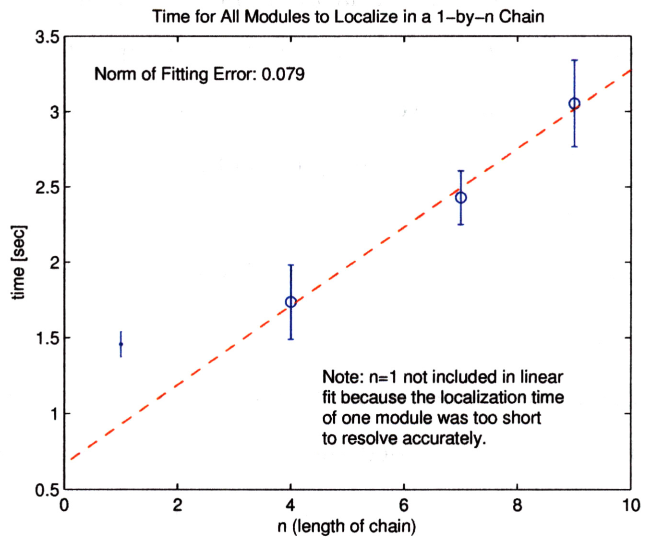


Figure 7-1: The time required for a chain of modules to localize is linear in the length of the chain. The circles represent the average time required for all modules to localize. For each different experiment, the whiskers span two standard deviations. When fitting the line to the data, the one–module case was ignored because it was too difficult to resolve the time required to localize one module.

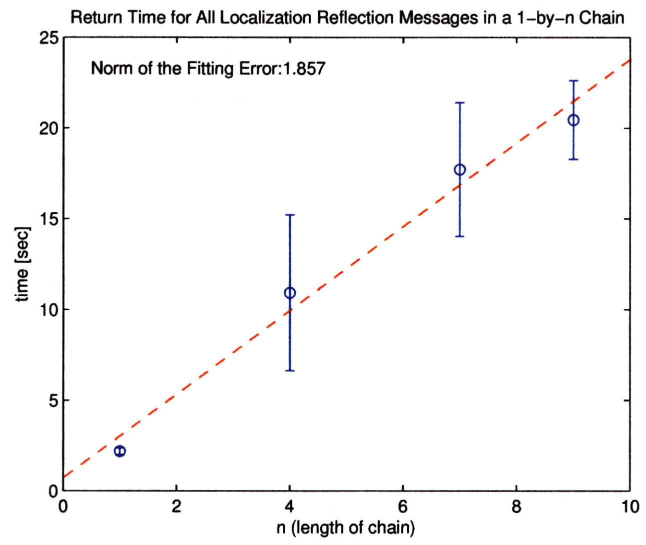


Figure 7-2: The time required for all reflection messages transmitted by a chain of modules to reach the GUI is linear in the number of modules in the chain. The circles in the plot represent the average time required for all reflection messages to propagate back to the GUI. The whiskers associated with each data point span two standard deviations.

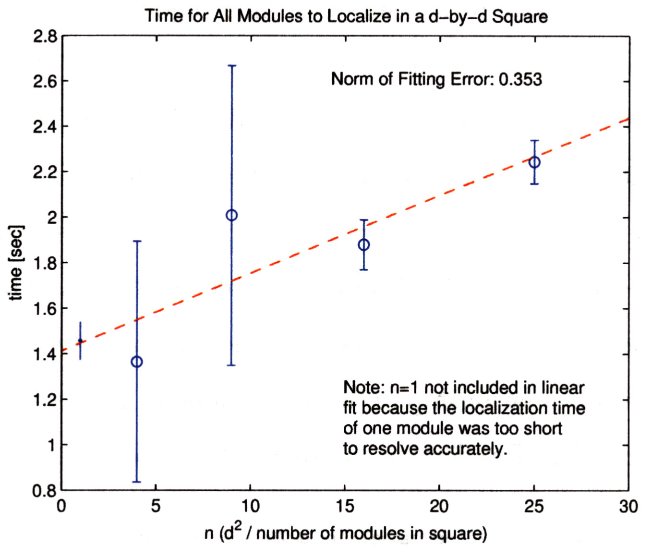


Figure 7-3: There is a linear relationship between the number of modules in a square structure and the amount of time required for all modules to receive a localization message. The circles represent the average time for each different experiment. The associated whiskers reach one standard deviation in each direction.

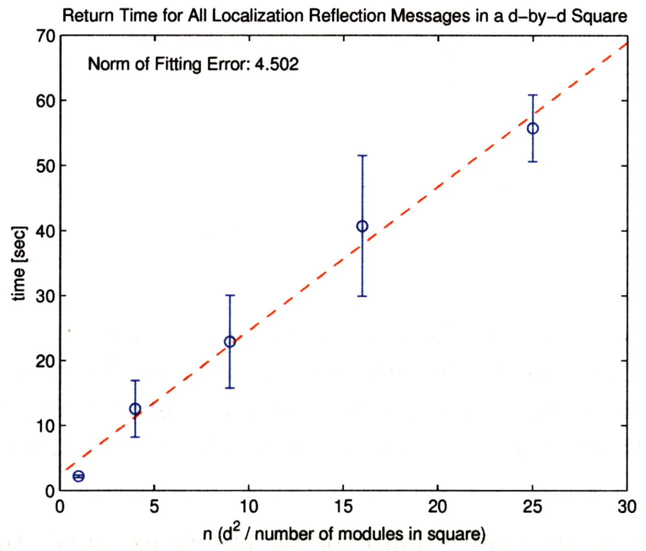


Figure 7-4: The time required for all reflection messages transmitted by a square structure of modules to reach the GUI is linear in the total number of modules in the square. The circles in the plot represent the average time required for all reflection messages to propagate back to the GUI. The whiskers associated with each data point span two standard deviations.

Although we were unable to measure the time required for all modules in a cubic structure to receive localization messages, we did measure the amount of time required for all of the modules to appear in the GUI. As before, the root module was chosen to be a corner of the structure. The number of different experiments we could run was limited by the number of cubes available, (27), but we did conduct 20, 16, and 6 experiments for 1–, 8–, and 27–module cubic structures. The results of these experiments are shown in Figure 7-5. As with the one– and two–dimensional cases, the figure shows a linear relationship between the number of modules in the structure and the time required for all reflection messages to propagate back to the GUI.

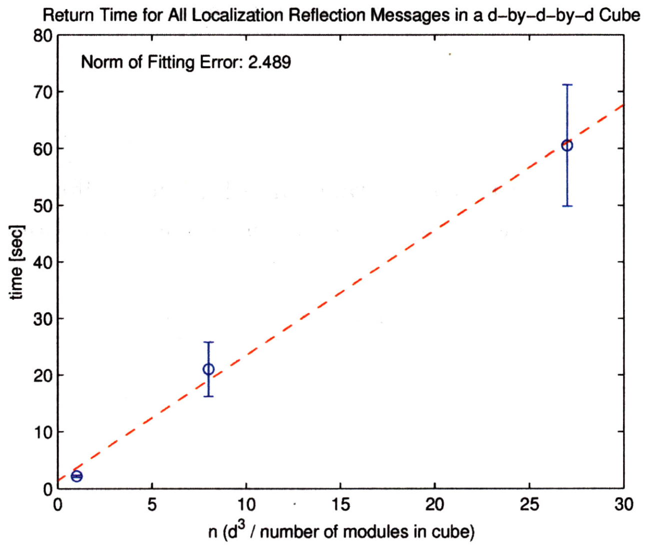


Figure 7-5: In a cubic structure, the amount of time required for all reflection messages to propagate back to the GUI is linearly related to the total number of modules. The circles represent the average time required for each different sized cubic structure. The associated whiskers reach one standard deviation in each direction.

Considering all of the localization experiments we executed, the success rate of the localization algorithm was excellent. In the 191 experiments, we only encountered 2 occasions where a single reflection message did not propagate back to the GUI. Considering that the equivalent of over 1,500 cubes were used in the experiments, this failure rate is less than one percent. Unfortunately, it is difficult to locate the source of the errors. The reflection messages typically travel through several modules, making it difficult to track a specific message. It is also possible that, due to a malfunctioning module, the missing messages were never transmitted.

Ignoring these two failures, the experiments were conclusive and indicate a linear relationship between the number of modules in a structure and the time required for all of them to receive a localization message. This supports the initial $O(nt)$ bound that we proposed in Section 5.2.4, where t is the maximum time required by any module to process a message. The experiments do not support the tighter $O(mt)$ that we also proposed in that section. (Recall, m is the longest of the set of shortest paths from the root module to any other, and it scales linearly with the side length of any square or cube.) This discrepancy can be explained by the fact that the modules were running the modified localization algorithm presented in Section 6.1.2. This is the algorithm that checks whether the module can successfully transmit messages to whichever neighbor it chooses as its parent. As shown in Section 6.1.2, the theoretical running time of this algorithm is $O(nt)$ and agrees with our results.

The experiments also found a strong linear relationship between the number of modules in a structure and the time required for all reflection messages to return to the GUI. This agrees with the $O(nt)$ bound we proposed in Section 5.2 for the receipt of all reflection messages. In the case of the reflection messages, there was never any guarantee that they would return any faster than $O(nt)$.

7.3 Inclusion Message Generation Results

The message generation algorithm worked flawlessly. In each of the 191 experiments, we used the GUI to include all the modules of the initial configuration in the final structure. In all cases, the GUI successfully generated the shortest possible path for all messages while taking into account the constraints imposed by pairs of neighboring modules that were unable to communicate with one another. In the five cases where not all modules in a structure received the inclusion messages destined for them, the source of the failure was narrowed to a bad communication channel or a faulty cube, never an incorrect message path.

The time required for the GUI to generate sequences of messages is quantified in Figure 7-6. The plot displays the time required to generate message sequences for chains of modules in which the root was placed at one end and all modules are included in the final structure. The figure demonstrates a linear relationship between the length of the chain and the time required to generate the sequence. This matches the theoretical bound of $O(n)$ presented in Section 5.3.3.

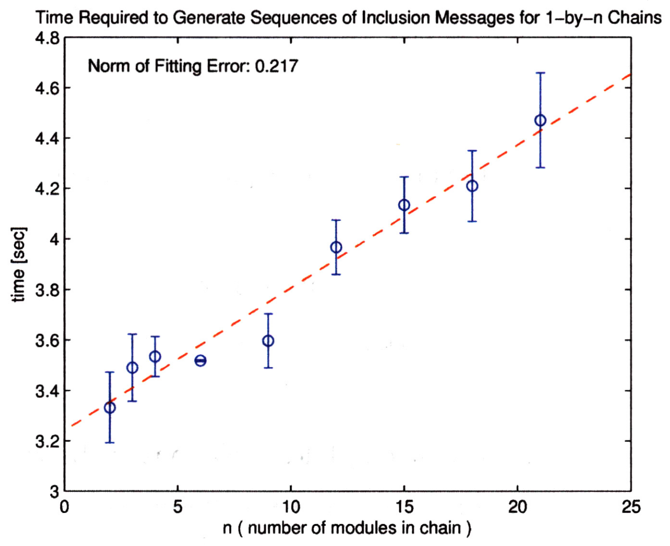


Figure 7-6: The time required by the GUI to generate sequences of messages is linear in the number of messages that need to be generated. In particular, the plot considers the time required to generate sequence of messages to include every module in an n-unit chain. The circular data points represent the average time required for each n, and the whiskers span a total of two standard deviations.

7.4 Shape Distribution Results

For each of the 191 experiments that we performed, we included every module that was a part of the initial structure in the final configuration. While not interesting from the perspective of disassembly, including every module in the final structure provided the most stringent test of the system. As mentioned in Section 5.4, modules assume they are not a part of the final structure unless they receive an inclusion message. Therefore, including every module in the final structure required that the maximum number of inclusion messages be distributed by the structure. For each experiment, we attempted to measure both the time required for all modules to receive their inclusion messages and the time for the associated reflection messages to return to the GUI. We were able to measure the amount of time required for all modules to receive their localization messages by watching the user LEDs on the modules and using the split function of the timer in the GUI. When a module determines that it is a part of the structure, it changes its LED from flashing to solid.

The specific experiments we used to test the shape distribution algorithm were identical to the experiments used to test the localization algorithm. We began by measuring the time required to send inclusion messages to all modules in an n–unit chain when the root module was placed at one end of the chain. We repeated this experiment 20 times for 1 cube; 16 times for a chain of 4 cubes; and 15 times for chains of both 7 and 9 cubes. We plotted the average time for all inclusion messages to reach their destinations in Figure 7-7. The plot shows a quadratic relationship between the number of modules in the chain and the time required for inclusion messages to reach them all. We also measured, and plotted in Figure 7-8, the time required for all reflection messages to return to the GUI. Typically, the last reflection message returned to the GUI shortly after the last module received its inclusion message, so the time required for all inclusion messages to propagate back the the GUI is also quadratic in n, the total number of modules in the system.

After finishing all experiments with chains of modules, we proceeded to perform the same experiment with squares of modules. We already had the results from the case of single cube. As with the tests of the localization algorithm, we performed 16 trials with 2-by-2–module squares; 17 trials with 3-by-3 squares; 18 trials with 4-by-4 squares; and 6 trials with 5-by-5 squares. In each test, we chose to place the root module in the corner of the square. The average time required for all modules to receive their inclusion messages, (and the associated set of error bars), is shown in Figure 7-9. Figure 7-10 shows the closely related time required for all reflection messages to return to the GUI. Figure 7-10 does not include a data point for the case of a 25 module square because in the six trials, there was never an outcome in

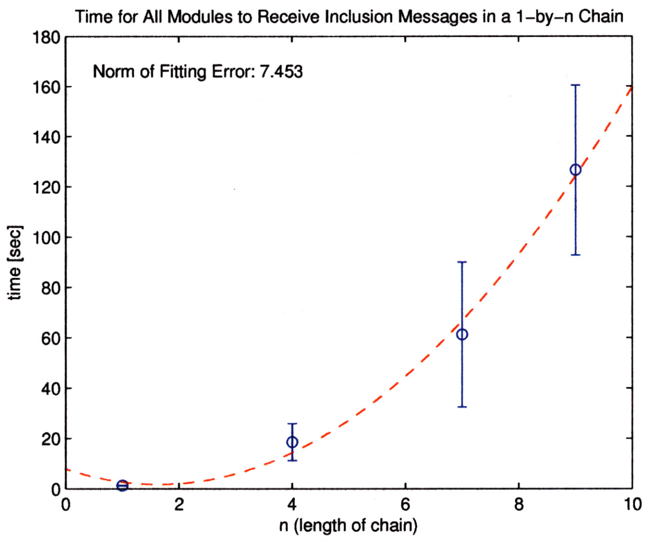


Figure 7-7: Time required for all modules in an n–unit chain to receive inclusion messages varies as n^2. In the plot, the circles represent the average time required for all modules to receive their inclusion messages, and the error bars cover a total of two standard deviations.

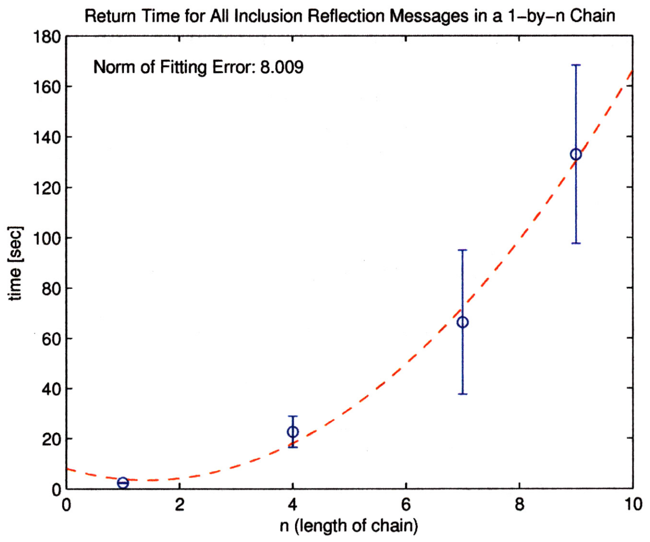


Figure 7-8: Given a chain of modules, the time required for the GUI to receive all of the reflection messages that are sent during the shape distribution phase is quadratic in the length of the chain. The circular data points are average times and the whiskers span a total of two standard deviations.

which all 25 reflection messages returned to the GUI. One or two messages always went missing. Both figures show a strong quadratic dependence between the number of modules in the square and the time required for the shape distribution phase to complete.

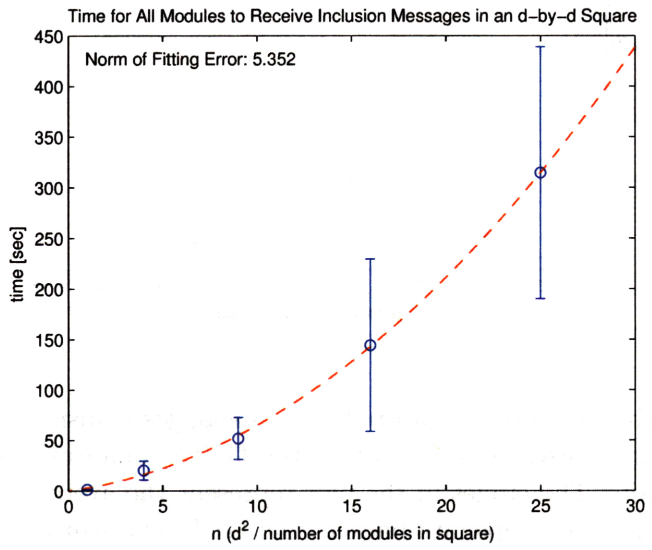


Figure 7-9: There is a quadratic relationship between the number of modules in a square structure and the amount of time required for all modules to receive an inclusion message. The circles represent the average time for each different experiment. The associated whiskers reach one standard deviation in each direction.

We also experimented with modules arranged to form a cube. We performed 16 experiments with an 8–module cube and 6 with a 27–unit cube. The 20 trials with a single module were also included in this set of experiments because a single module is also a cube. As before, we placed the root module at the corner of the structure. Figure 7-11 shows the time required for inclusion messages to reach each module in the structure. It is not surprising that the three data points can be fit perfectly by a quadratic function because a parabola has three degrees of freedom. We did not include a plot of the time required for all reflection messages to return to the GUI because there was never an experiment with the 27–module cube in which all reflection messages returned. In the six trials, one or two were always lost.

Over the course of all 191 experiments, we encountered 5 cases where some num-

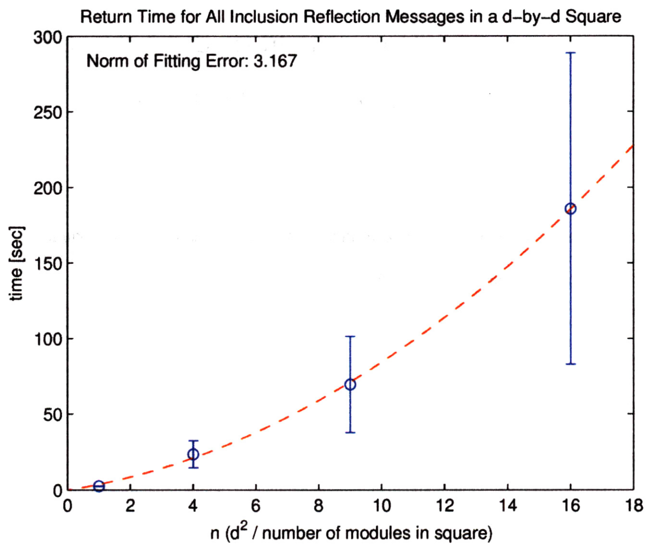


Figure 7-10: The time required for all reflection messages transmitted by a square of modules during shape distribution to reach the GUI is quadratic in the number of modules in the chain. The circles in the plot represent the average time required for all reflection messages to propagate back to the GUI. The whiskers associated with each data point span two standard deviations.

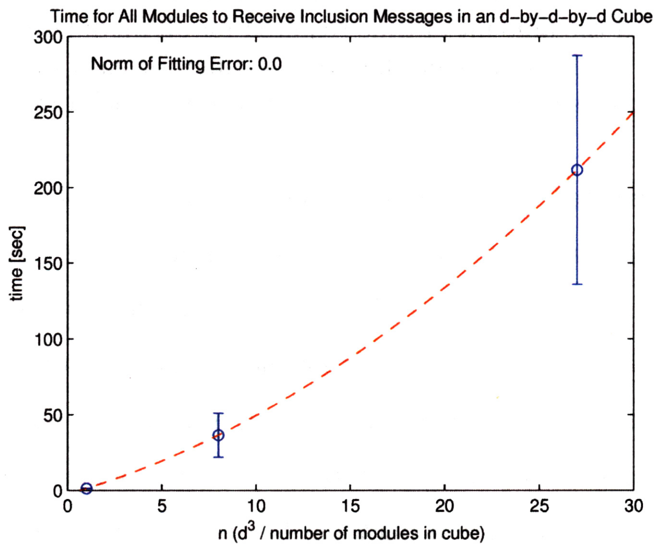


Figure 7-11: The three average times required to transmit inclusion messages to all modules in a cube can be fit perfectly by a quadratic function. The average times, shown by the circles, are bounded by whiskers which extend one standard deviation in each direction.

ber of inclusion messages were not received. This accounted for a total of 17 messages that were missed. In cases where several inclusion messages were not received during one experiment, the cause could generally be traced back to one poorly aligned IR LED/photodiode pair that had possibly shifted since the modules decided it was a valid message path. Because of the way inclusion messages are distributed, a malfunctioning communication interface can affect all modules which depend on that interface being a part of the inclusion pointer chain that delivers their inclusion messages. Even though 17 messages were lost, this number is still less than 2 percent of the more than 1500 inclusion message that had to be sent over the course of all experiments.

There were also 26 reflection messages that did not propagate back to the GUI. This is also less than 2 percent of the total number of reflection messages sent during the shape distribution phase for all experiments. One explanation for the slightly higher number of reflection messages that did not return to the GUI during the shape distribution phase in comparison to the localization phase is that during the shape distribution phase, the reflection messages must contend with the inclusion messages which are also propagating through the system. The total number of messages places a high load on the system, and it is possible that some messages reached their timeouts before being acknowledged by a neighboring module. This explanation is supported by the fact that most reflection messages were lost in larger square and cubic structures in which the modules near the root are handling a proportionally greater number messages than in smaller structures.

7.4.1 Running Time Comparison

Section 5.4.3 theorized that the running time of the shape distribution algorithm would be $O(nt)$, where n is the number of modules in the system, and t is the time required for the slowest module to process a message. All experiments indicate a quadratic, $O(n^2)$, running time. This quadratic relationship may be explained by the way Matlab behaves when bombarded with too much serial data. As stated in Section 6.3, Matlab, when faced with a large amount of serial data, begins to drop or corrupt many of the messages received from the root module. In turn, this slowdown affects t because the synchronization process described in Section 6.1.1 couples how quickly the modules can exchange messages with how quickly Matlab can process them.

Because the rate at which Matlab can receive and process messages seems to be inversely proportional to how busy it is transmitting messages, larger structures, because they require more inclusion messages to be transmitted, may limit how quickly Matlab can receive messages from the root. In short, this means that t is directly correlated to the size of the structure, n. Therefore, the running time of the shape distribution algorithm is actually $O(n^2)$. This dependence of t on n also explains why the time to receive all reflection messages during the shape distribution phase is quadratic in n: the last module to be included has to wait until after it has received its inclusion message before transmitting a reflection message. Therefore, if the time required by the modules to receive all inclusion messages is $O(n^2)$, the time for all reflection messages to propagate back to the GUI must also be $O(n^2)$.

7.5 Disassembly Results

The disassembly process itself was not specifically tested after each of the 191 experiments. It would have required too much time to allow the modules to fall apart and then reassemble them by hand. Additionally, the self–disassembly and reassembly process would have required activating each Magswitch twice. Over time, it would have amounted to a significant drain on the batteries of each cube and would have required more frequent recharging. Instead of sending a disassemble message after each experiment, we sent a reset message. Reset messages, like disassemble messages, are propagated by broadcast and quickly reach all modules in a structure. In each of the 191 recorded experiments, the reset messages successfully reached all modules. This allows us to conclude that disassembly messages would also have reached all modules.

For a more specific test of the system's ability to disassemble, we conducted an experiment which used fifteen modules initially arranged in a 3-by-5 rectangle. We followed the standard process of neighbor discovery, localization, message generation, and message distribution followed by actual disassembly. The final structure that we modeled in the GUI resembled a basic humanoid robot and is shown in Figure 7-12. To transform the initial 3-by-5 rectangle into the desired configuration, 5 of the initial 15 modules had to separate from the structure. We tested this process sixteen times during which one individual Magswitch failed twice. Once the Magswitch was replaced, all trials were completely successful. The average time for all required for modules to transmit their positions back to the GUI was 30.79 seconds. The average time required for the GUI to generate a sequence of inclusion messages was 3.58 seconds. The time required for all modules to use reflection messages to inform the GUI that they had received their inclusion messages was 54.58 seconds. Once the five modules not a part of the final structure received disassemble messages, they began disconnecting immediately. This allowed us to conclude that the system was truly able to self--disassemble.

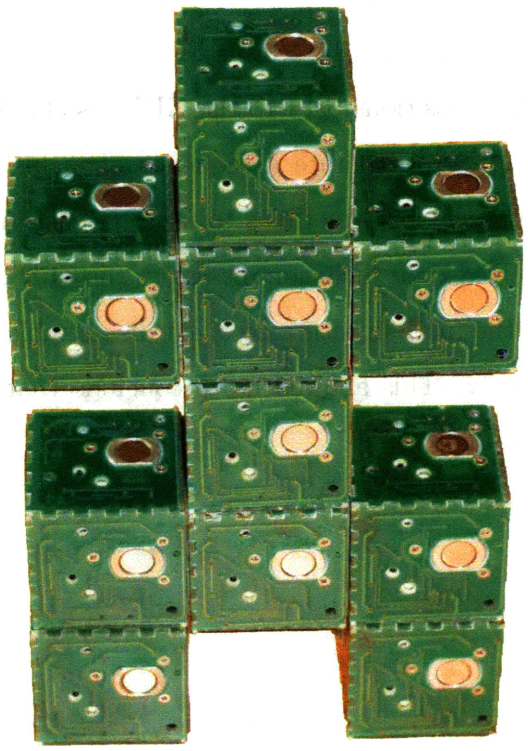


Figure 7-12: The final robot–like shape we "self–disassembled" using an initial 3-by-5 rectangle of modules. From start to finish, the self–disassembly process required a total of approximately 90 seconds excluding any time spent modeling the desired final shape.

Chapter 8

Conclusions

In the process of designing, building, programming, and testing our self–disassembling system, we encountered many problems. We developed solutions for some of the problems, but we learned from them all. If we were able to develop solutions for the problems we encountered, we attempted to incorporate the changes into the system. In some cases this was impossible. In what follows, we present the most important lessons that we learned while developing our self–disassembling system. Where appropriate, we attempt to render advice for future revisions of the system as well as more general advice about distributed and modular systems as a whole. Section 8.1 presents several considerations that should be kept in mind when designing the hardware for a distributed or modular system. Then, Section 8.2 addresses some of the lessons we learned during the software development process.

8.1 Hardware Lessons Learned

The most important thing learned in the hardware development process was the value of easy manufacture. If we were only producing one module, the complexity of its assembly would be irrelevant. Since we built 28, it was important to make the assembly process as simple as possible. While we made a concerted effort to simplify the hardware assembly process, assembling a single module still required approximately ten major steps and two days of time. It required several weeks and the work of several individuals to assemble and test all 28 modules that we produced. If we were to produce any additional modules, we would consider using a commercial fabricator. Unfortunately, the mechanical and electrical systems of the modules are so tightly integrated that it would be difficult to communicate the intricacies of the assembly process to anyone not highly focused on the project. Future revisions to the project should take this into account and attempt to further simplify the assembly process.

We also came to appreciate the ability to individually test each subsystem of each module. This was possible for some aspects of the system, for example the Magswitches, but not all. In the manufacturing process, we encountered many commercially produced printed circuit boards that contained broken, non–conducting vias. These vias, until they were repaired, rendered the modules completely inoperable. Unfortunately, the broken vias were difficult to locate because we did not have the ability to isolate the different electronic subsystems. As with the assembly process, the different subsystems were too tightly integrated. Eventually, the experience we gained debugging the early modules accelerated the debugging process in modules that were assembled later. By planning for debugging the hardware during the design phase, we could have made the entire process more efficient.

8.2 Software Lessons Learned

During the software development process, one of the first lessons we learned was that every attempt should be made to ensure that communication systems are as robust as possible. The inter–module messages contained a start character, checksum, and stop character because without these, communication was unreliable. Even with these features, Matlab still failed to handle serial communication well. Future versions of the system should replace Matlab with a GUI that can better handle vast amounts of serial data. Another possibility may be to implement a low–level program which parses the data from the desktop computer's serial port before passing it along to Matlab to be be displayed in the GUI.

The self–disassembly algorithms we implemented need to be made more robust to communication failures. In the current system, if two modules are misaligned, they are unable to pass message to each other. The message generation algorithm attempts to detect these broken communication links and route messages around them, but it makes no attempt to account for communication links that are partially reliable. If the message generation algorithm mistakenly thinks that a broken link is viable, a potentially large number of modules may not receive their shape distribution messages. Improvements to the system should focus on re-routing messages on–the–fly to avoid unreliable communication links. This should be done by the modules, not the GUI, in a distributed manner. If these algorithms to reroute messages around broken links are implemented, they may also be able to relieve the GUI of all its path planning duties.

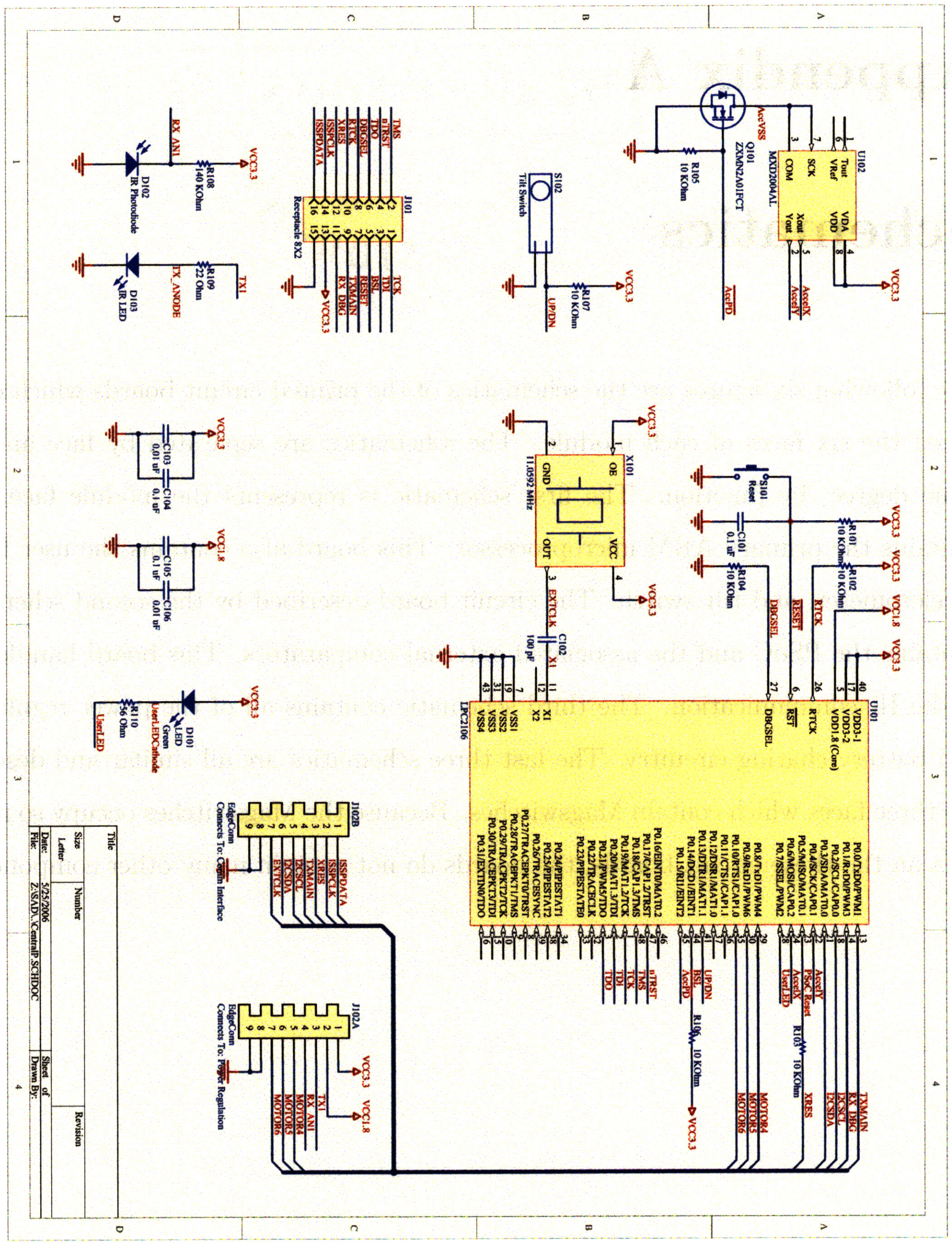
The final lesson we learned is the importance of, (and difficulty of obtaining), accurate information about any distributed system's state. It is nearly impossible to debug distributed systems efficiently. Because each module executes asynchronously, it is difficult to understand the state that every module is in at any particular time. In our case, the messages transmitted by the modules were indicative of their status, but they only provided a high–level impression of their internal state. While the development tools used for the ARM processor allowed us to debug the code executing on one module and gain detailed information about that module's state, we could only do this for one module at a time. Because the interactions between modules are what give a distributed system its unique abilities, the ability to only debug one module at a time is a hinderance to the implementation of more interesting and complex algorithms. In our case, the algorithms were relatively simple, but even three or four modules produced too much data to process and understand. At best, it was possible to observe and understand the messages transmitted by two neighboring modules. Even so, it was not possible to accurately order both sets of messages by transmission time. We were only able approximate the time that a message was sent by one cube with respect to the time that a message was sent by the other.

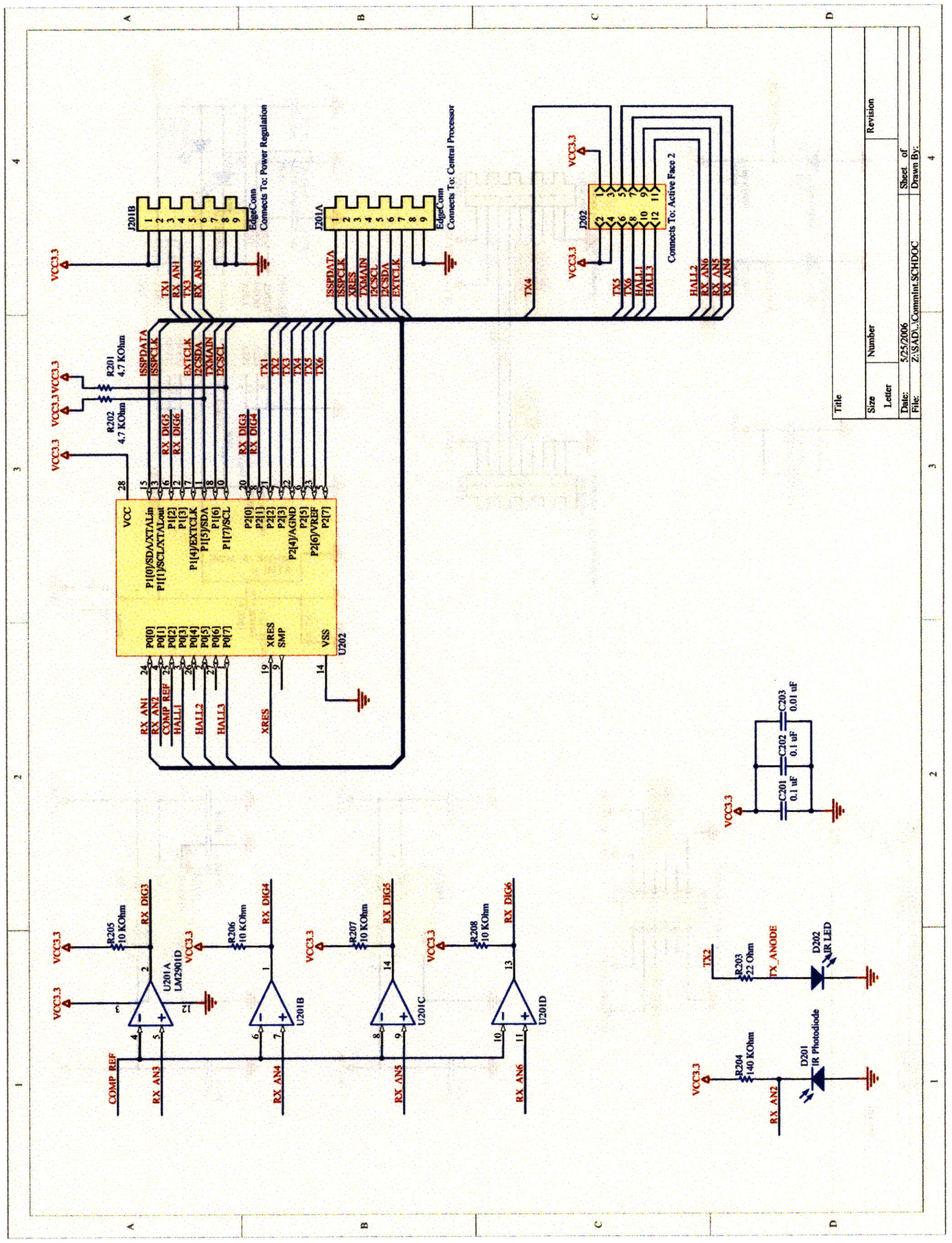
In particular, individual distributed systems should be designed with special care paid to how they can be analyzed and debugged. In general, more research should be devoted to developing generic tools that can be used to effectively analyze, debug, and control large distributed systems. If we are ever to reach a point in time when we can successfully deploy distributed systems containing millions of units, we will first need to develop interfaces that can analyze, program, and control these systems in an efficient manner.

Appendix A

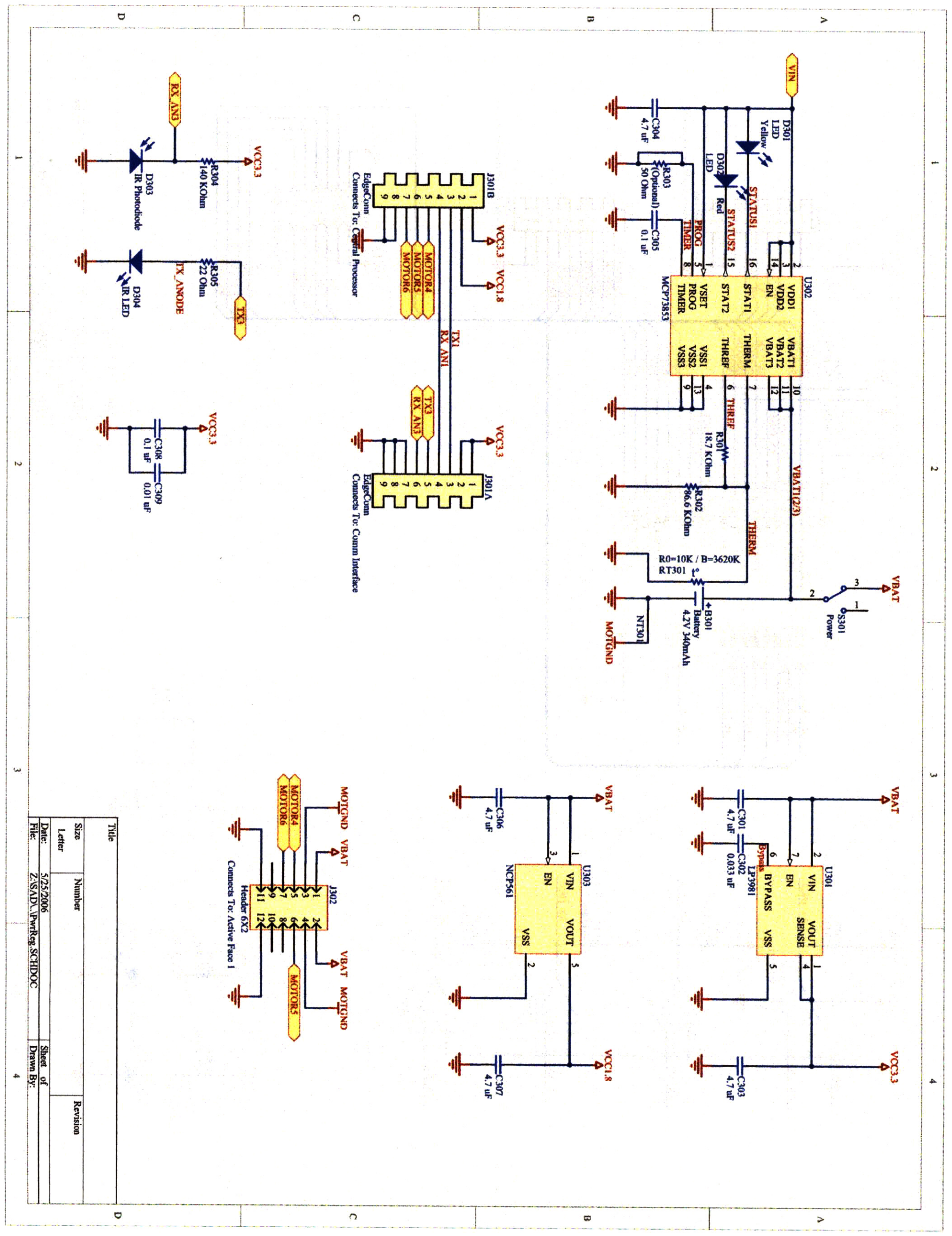
Schematics

The following six figures are the schematics of the printed circuit boards which composed the six faces of each module. The schematics are separated by face and, to some degree, by function. The first schematic is represents the module face that contains the primary ARM microprocessor. This board also contains the user LED, accelerometer, and tilt switch. The circuit board described by the second schematic contains the PSoC and the associated external comparators. This board handles all of the IR communication. The third schematic contains all of the power regulation and battery charing circuitry. The last three schematics are all similar and describe the three faces which contain Magswitches. Because the Magswitches occupy so much area on the these circuit boards, the boards do not contain many other components.

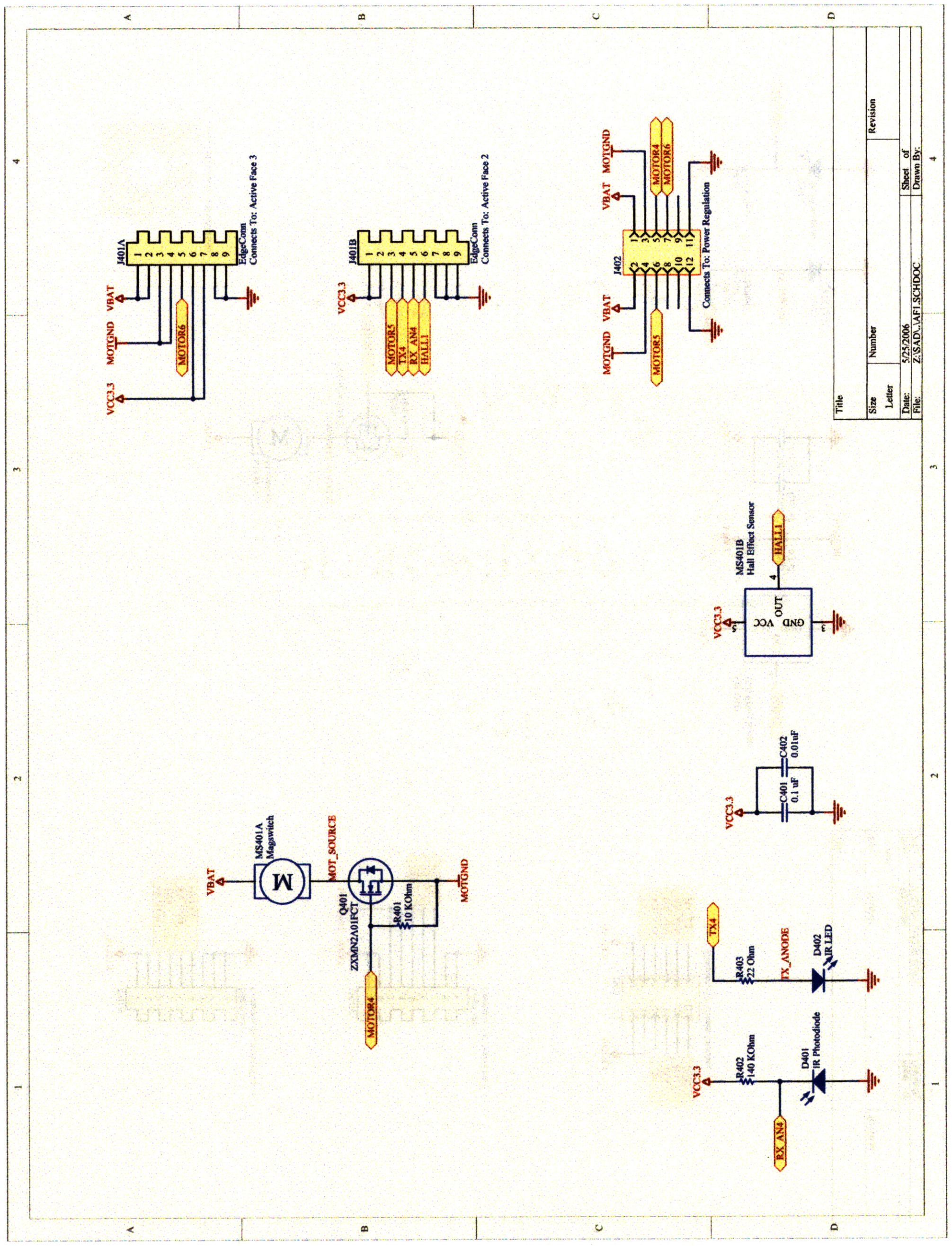




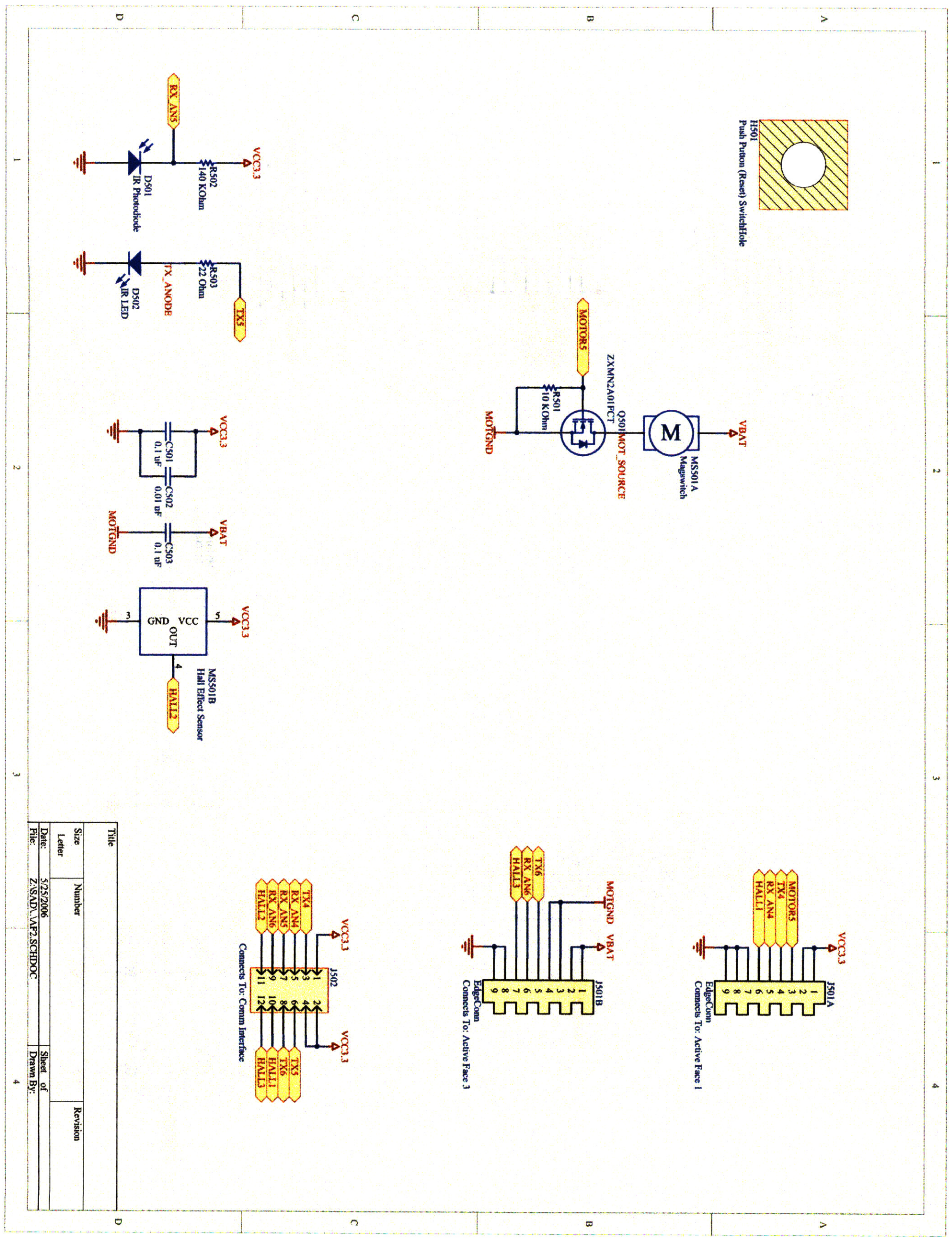
U202
J201B
EdgeConn
Connects To: Power Regulation
J201A
EdgeConn
Connects To: Central Processor
J202
Connects To: Active Face 2
U201A
LM2901D
U201B
U201C
U201D
R201 4.7 KOhm
R202 4.7 KOhm
R203 22 Ohm
R204 140 KOhm
R205 10 KOhm
R206 10 KOhm
R207 10 KOhm
R208 10 KOhm
C201 0.1 uF
C202 0.1 uF
C203 0.01 uF
D201 IR Photodiode
D202 IR LED
VCC3.3
COMP REF
RX AN1
RX AN2
RX AN3
RX AN4
RX AN5
RX AN6
RX DIG3
RX DIG4
RX DIG5
RX DIG6
TX_ANODE
TX1
TX2
TX3
TX4
TX5
TX6
HALL1
HALL2
HALL3
XRES
ISSPDATA
ISSPCLK
EXTCLK
I2CSDA
I2CSCL
TXMAIN
Title
Size
Letter
Number
Revision
Date: 5/25/2006
File: Z:\SAD\..\CommInt.SCHDOC
Sheet of
Drawn By:

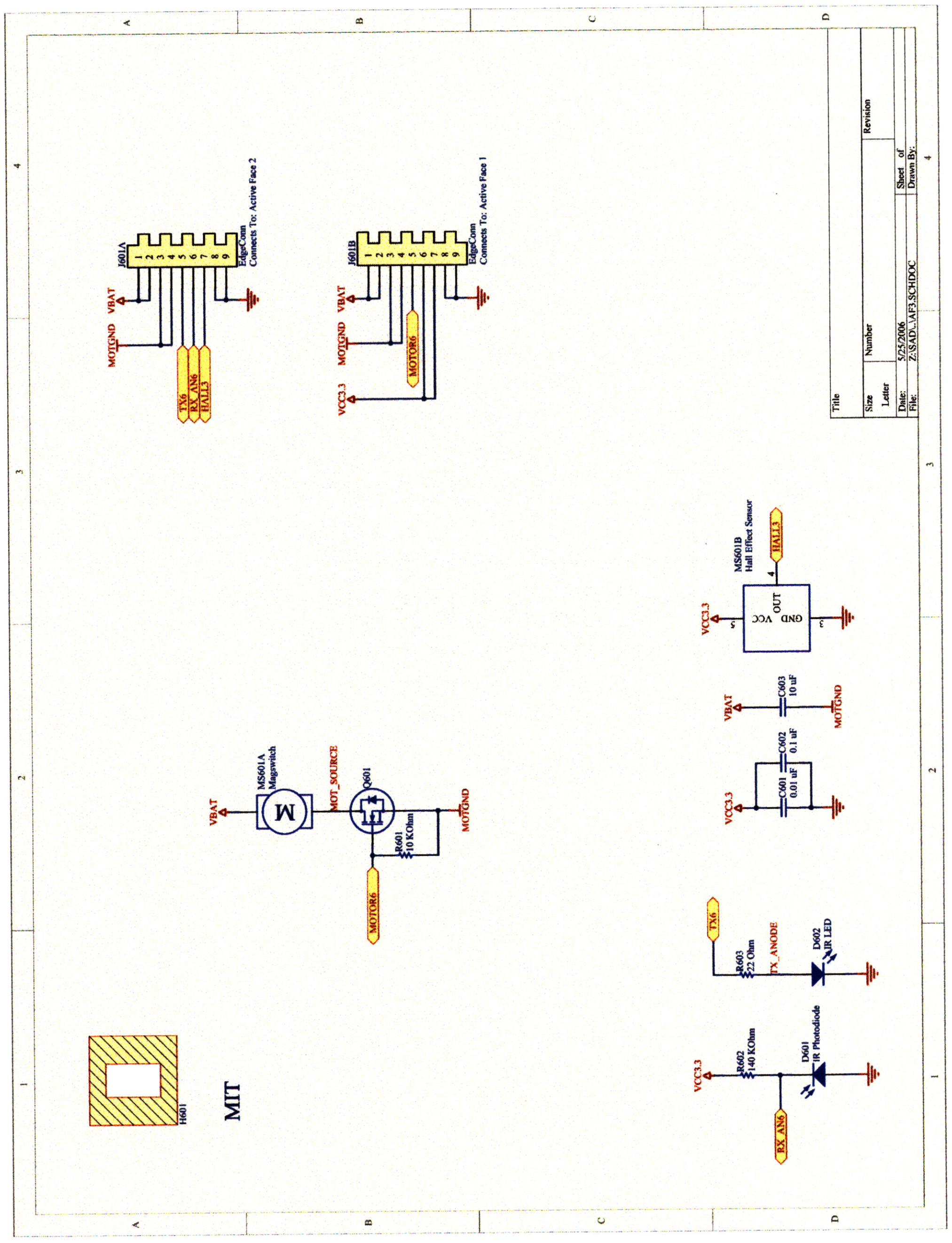


VIN
D301 LED Yellow
STATUS1
D302 LED Red
STATUS2
C304 4.7 uF
R303 (Optional) 50 Ohm
C305 0.1 uF
PROG
TIMER
U302
VDD1 VDD2 EN STAT1 STAT2 VSET PROG TIMER
VBAT1 VBAT2 VBAT3 THERM THREF VSS1 VSS2 VSS3
MCP73853
THREF
R301 18.7 KOhm
R302 86.6 KOhm
VBAT(2/3)
THERM
R0=10K / B=3620K
RT301 t°
B301 Battery 4.2V 340mAh
NT301
MOTGND
S301 Power
VBAT
J301B EdgeConn Connects To: Central Processor
VCC3.3 VCC1.8
MOTOR4 MOTOR5 MOTOR6
TX1 RX_AN1
TX3 RX_AN3
J301A EdgeConn Connects To: Comm Interface
RX_AN3
VCC3.3
R304 140 KOhm
D303 IR Photodiode
R305 22 Ohm
TX3
TX_ANODE
D304 IR LED
C308 0.1 uF
C309 0.01 uF
U301
VIN EN BYPASS VOUT SENSE VSS
LP3981
Bypass
C301 4.7 uF
C302 0.033 uF
C303 4.7 uF
VCC3.3
U303
VIN EN VOUT VSS
NCP561
C306 4.7 uF
C307 4.7 uF
VCC1.8
J302 Header 6X2 Connects To: Active Face 1
MOTGND VBAT
MOTOR4 MOTOR6 MOTOR5
Title
Size Letter
Number
Revision
Date: 5/25/2006
File: Z:\SAD\..\PwrReg.SCHDOC
Sheet of
Drawn By:



J401A
EdgeConn
Connects To: Active Face 3
VCC3.3
MOTGND
VBAT
MOTOR6
J401B
EdgeConn
Connects To: Active Face 2
MOTOR5
TX4
RX_AN4
HALL1
J402
Connects To: Power Regulation
MOTOR4
MS401A
Magswitch
MOT_SOURCE
Q401
ZXMN2A01FCT
R401
10 KOhm
C401
0.1 uF
C402
0.01uF
MS401B
Hall Effect Sensor
GND
VCC
OUT
R403
22 Ohm
TX_ANODE
D402
IR LED
R402
140 KOhm
D401
IR Photodiode
Title
Size
Letter
Number
Revision
Date:
5/25/2006
Sheet of
File:
Z:\SAD\..\AF1.SCHDOC
Drawn By:


H501
Push Button (Reset) SwitchHole
VBAT
M
MS501A
Magswitch
Q501
ZXMN2A01FCT
MOT_SOURCE
MOTOR5
R501
10 KOhm
MOTGND
VCC3.3
R502
140 KOhm
RX_AN5
D501
IR Photodiode
R503
22 Ohm
TX5
TX_ANODE
D502
IR LED
C501
0.1 uF
C502
0.01 uF
C503
0.1 uF
GND
VCC
OUT
MS501B
Hall Effect Sensor
HALL2
J501A
EdgeConn
Connects To: Active Face 1
MOTOR5
TX4
RX_AN4
HALL1
J501B
Connects To: Active Face 3
TX6
RX_AN6
HALL3
J502
Connects To: Comm Interface
TX4
RX_AN4
RX_AN5
RX_AN6
HALL2
TX5
TX6
HALL1
HALL3
Title
Size
Letter
Number
Revision
Date: 5/25/2006
Sheet of
File: Z:\SAD\...\AF2.SCHDOC
Drawn By:



MIT
H601
VCC3.3
R602
140 KOhm
D601
IR Photodiode
RX_AN6
TX6
R603
22 Ohm
TX_ANODE
D602
IR LED
VBAT
MS601A
Magswitch
M
MOT_SOURCE
Q601
R601
10 KOhm
MOTGND
MOTOR6
C601
0.01 uF
C602
0.1 uF
C603
10 uF
MS601B
Hall Effect Sensor
VCC
GND
OUT
HALL3
J601A
EdgeConn
Connects To: Active Face 2
J601B
EdgeConn
Connects To: Active Face 1
Title
Size
Letter
Number
Revision
Date:
5/25/2006
Sheet of
File:
Z:\SAD\..\AF3.SCHDOC
Drawn By:
A
B
C
D
1
2
3
4

Appendix B

ARM Source Code

This appendix countains all of the source code which controls the Philips ARM microprocessor. For more details, consult Chapters 4 and 6.

B.1 main.c

This source file initializes the system and then loops forever calling the message handling and transmission routines.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include "miche.h"

#define _main_
#include "main.h"
#undef _main_

#include "hal/rtc.h"
#include "hal/uart.h"
#include "hal/i2c.h"
#include "hal/magswitch.h"
#include "debug.h"
#include "hal/orientation.h"
#include "usrint.h"
#include "crc.h"
#include "hal/psoc.h"
#include "msgs/handlemsg.h"
#include "localize.h"
#include "msgs/parsemsg.h"
#include "msgs/msgsched.h"
#include "log.h"

struct cfgData settings;


/******************************************************************************/
int main(void) {
  struct time t,u,v;
  unsigned int ctime;
  char msg[65];
  int temp;

  MAMCR = 2;                                    //Enable full memory acceleration
  initSystem();

  while (1) {
    handleIncomingMsgs();
    handleOutgoingMsgs();
    handleBackgroundTasks();

  }
}


/******************************************************************************/
void initSystem(void)
{
  initI2C();
  initUART(9600);
  initRTC();
  initAccelerometer();
  initTiltSwitch();
  initTMR1();
  initLED();

  generateCRCTable();

  setTMR1MR1Period( 20000 );

  __ARMLIB_enableIRQ();

  initMagswitches();                            // must happen after interrupts
                                                // are enabled because the end
                                                // of routine re-enables them

  IOCLR  = PSOC_RESET;                          // PSoC reset is active high
  IODIR |= PSOC_RESET;                          // make PSoC reset an output
  resetPSoC();                                  // reset the PSoC
  pause(5000);                                  // give PSoC time to initialize
  readSettingsFromPSoC();                       // read settings
  setComparatorThreshold(settings.comparatorThreshold);
                                                // setup comparators for RX'ing

  logEvent( 0, "%s", "System Reset");

}


/******************************************************************************/
void handleBackgroundTasks() {
  if( (ledOnForIRT ) && ( isTimeExpired() ) )
    blankLED();
}
```

B.2 handlemsg.c

This code implements most of the high–level disassembly algorithms.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include "miche.h"
#include "msgs.h"
#include "hal/rtc.h"

#define _handlemsg_
#include "handlemsg.h"
#undef _handlemsg_


#include "hal/psoc.h"
#include "hal/magswitch.h"
#include "log.h"
#include "usrint.h"

#include "main.h"

#include "demo.h"

#include "debug.h"

#define _logerrors_

struct cube nbrs[7] = {
  {{0, 0, 0}, FALSE, 0, FALSE, FALSE, FALSE},
  {{0, 0, 0}, FALSE, 0, FALSE, FALSE, FALSE},
  {{0, 0, 0}, FALSE, 0, FALSE, FALSE, FALSE},
  {{0, 0, 0}, FALSE, 0, FALSE, FALSE, FALSE},
  {{0, 0, 0}, FALSE, 0, FALSE, FALSE, FALSE},
  {{0, 0, 0}, FALSE, 0, FALSE, FALSE, FALSE},
  {{0, 0, 0}, FALSE, 0, FALSE, FALSE, FALSE}
};

unsigned char ledOnForIRT = FALSE;

static unsigned char incChainPtr = 0;
static int state = ASSEMBLING;
static unsigned int totalRxCount = 0;

/******************************************************************************/
void handleIncomingMsgs() {
  unsigned char rx_status;
  char raw_msg[MAX_MSG_LENGTH+ 1];

  int rcvface;

  unsigned int seq_id;                        // for all msg types
  unsigned int acked_type, acked_seq_id;      // for ACK msgs
  int discon_face;                            // for DRQ msgs
  unsigned int hops, bd;                      // for INC msgs
  struct point dest;                          // for LOC msgs
  unsigned char mss;                          // for MSS msgs
  struct point src;                           // for REF msgs
  unsigned int ref_type, uid;
  char rnbrs[7];
  struct time t;                              // for RTC msgs
```

```
getRxBufferStatus( &rx_status );             // get summary of which faces have incoming msgs pending

for( rcvface = 0; rcvface <= 6; rcvface++ ) { // for each face including the uplink:

  /* we first deal with the uplink interface separately */

  if( rcvface == 0 ) {                        // for the uplink,
    readUplinkQueue( raw_msg );               // read uplink message buffer

    if( raw_msg[0] == 0 )                     // if no message received,
      continue;                               // continue to to check face 1
    nbrs[ME].rxAble = TRUE;                   // otherwise, we can rcv uplink
    nbrs[ME].txAble = TRUE;                   // and assume we can transmit
    if( strcmp( raw_msg, "#*" ) == 0 )        // if user entered #*CR,
      startDebugInterface();                  // start the debugging interface
  }

  /* now, we deal with msgs from neighboring cubes */

  else if( rx_status & (0x01 << (rcvface - 1) ) )
                                              // for all faces besides uplink that have msg. pending,
    readPSoCRxQueue( rcvface, raw_msg );      // read msg buffer on that face
  else                                        // if no message pending,
    continue;                                 // then continue to next face

  /* if we've made it this far, new msg. exists in raw_msg */

  if( rcvface != 0 )                          // if new msg not from uplink,
    totalRxCount++;                           // inc. total # msgs rcv'd

  if( !checkCRC( raw_msg ) )                  // if CRC is unsuccessful,
  {
    txNCKMsg( rcvface );                      // send NCK msg on rcving face
    continue;                                 // and continue to next face
  }

  /* if we make it here, new msg with valid CRC exists in raw_msg */

  nbrs[rcvface].rxAble = TRUE;                // indicate we can rcv msgs from that neighbor

  if( !getMsgSeqID( raw_msg, &seq_id ) )      // if can't extract seq_id,
  {
    txNCKMsg( rcvface );                      // send NCK msg on rcving face,
    continue;                                 // and continue to next face
  }

  /* if we make it here, we were able to extract Seq. ID of the new msg */

  switch( getMsgType( raw_msg ) )             // process msg. based on type
  {
    case ACK_MSG:                             // ACKnowledge message
      acked_type = getMsgAckedType( raw_msg );// get the type of msg ACK is responding to

      if( !getMsgAckedSeqID( raw_msg, &acked_seq_id ) )
      {                                       // if can't extract ACKed Seq. ID,
        txNCKMsg( rcvface );                  // send NCK msg on rcving face,
        continue;                             // and continue to next face
      }

      /* if we make it to here, ACK msg was parsed correclty */
  if( msgq_handleACKMsg( rcvface, acked_type, acked_seq_id ) )
                                        // if msg queue system successfully matches ACK msg rcvd on
                                        // a face to msg pending in tx queue for that face,
    nbrs[rcvface].txAble = TRUE;        // indicate that we can successfully tx on that face

  handleACKMsg( rcvface, acked_type, acked_seq_id );
                                        // process ACK msg
  continue;                             // move on to next face

case DAS_MSG:                           // DisASsemble message
  handleDASMsg( rcvface, seq_id );      // nothing else to parse, process DAS msg
  continue;                             // move on to next face

case DCA_MSG:                           // DisConnect All message
  handleDCAMsg( rcvface, seq_id );      // nothing else to parse, process the msg
  continue;                             // move on to next face

case DRQ_MSG:                           // Disconnect ReQuest message

  handleDRQMsg( rcvface, seq_id );      // nothing else to parse, process the DRQ message
  continue;                             // move on to next face

case INC_MSG:                           // INClusion message
  if( !getMsgHops( raw_msg, &hops ) )   // if unable to extract number of hops,
  {
    txNCKMsg( rcvface );                // transmit NCK msg,
    continue;                           // and move on to next face
  }

  if( !getMsgBranchDirection( raw_msg, &bd ) )
  {                                     // if unable to extract branch direction,
    txNCKMsg( rcvface );                // transmit NCK msg,
    continue;                           // and move on to next face
  }

  /* if we make it to here, INC msg was parsed correclty */

  handleINCMsg( rcvface, seq_id, hops, bd );
                                        // process the INC msg
  continue;                             // move on to next face

case IRT_MSG:                           // InfraRed Test message
  handleIRTMsg( rcvface, seq_id );      // nothing else to parse, process IRT msg
  break;

case LOC_MSG:                           // LOCalization message
  if( !getMsgDest( raw_msg, &dest ) )   // if unable to extract destination,
  {
    txNCKMsg( rcvface );                // transmit a NCK msg
    continue;                           // and move on to next face
  }

  /* if we make it to here, LOC msg was parsed correclty */

  handleLOCMsg( rcvface, seq_id, &dest ); // handle the LOC msg
  continue;                             // move on to next face

case MSS_MSG:                           // MagSwitch State message

  if( !getMsgMSStatus( raw_msg, &mss ) )  // if unable to extract magswitch state,
  {
    txNCKMsg( rcvface );                 // transmit NCK msg,
    continue;                            // and move on to next face
  }

  /* if we make it to here, INC msg was parsed correclty */

  handleMSSMsg( rcvface, seq_id, mss );  // process MSS message
  continue;                              // move on to next face

case NCK_MSG:                            // Not aCKnowledge message
  handleNCKMsg( rcvface, seq_id );       // nothing else to parse, process NCK msg
  continue;                              // move on to next face

case PNG_MSG:                            // PiNG message
  handlePNGMsg( rcvface, seq_id );       // nothing else to parse, process NCK msg
  continue;                              // move on to next face

case REF_MSG:                            // REFlection message

  if( !getMsgRefType( raw_msg, &ref_type ) )
  {                                      // if unable to extract ref type
    txNCKMsg( rcvface );                 // transmit NCK msg
    continue;                            // and move on to next face
  }

  if( !getMsgSrc( raw_msg, &src ) )      // if unable to extract src,
  {
    txNCKMsg( rcvface );                 // transmit NCK msg,
    continue;                            // and move on to next face
  }

  if( !getMsgUID( raw_msg, &uid ) )      // if unable to extract uid,
  {
    txNCKMsg( rcvface );                 // transmit NCK msg
    continue;                            // and move on to next face
  }

  if( !getMsgReachableNbrs( raw_msg, rnbrs ) )
  {                                      // if can't get reachable nbrs,
    txNCKMsg( rcvface );                 // transmit NCK msg
    continue;                            // and move on to next face
  }

  /* if we make it to here, REF msg was parsed correclty */

  handleREFMsg( rcvface, ref_type, seq_id, &src, uid, rnbrs );
                                         // process the REF msg
  continue;                              // move on to next face

case RST_MSG:                            // ReSeT message
  handleRSTMsg( rcvface, seq_id );       // handle RST message
  continue;                              // and move on to next face

case RTC_MSG:                            // Real Time Clock message

  if( !getMsgTime( raw_msg, &t ) )       // if unable to extract time,
  {
    txNCKMsg( rcvface );                 // transmit NCK msg
    continue;                            // and move on to next face
  }
```

```
        /* if we make it to here, RTC msg was parsed correclty */

        handleRTCMsg( rcvface, seq_id, &t );    // process the RTC msg
        continue;                               // move on to next face
    }
  }
}


/*****************************************************************************/
void handleOutgoingMsgs() {
  int txface;

  if( (state == DISCARDED) || (state == SOLIDIFIED) ||
      (state == LOCALIZED) || (state == SCULPTED) )
    return;

  for( txface = 1; txface <= 6; txface++ )      // for each neighbor,
  {
    if( (!nbrs[txface].pop) || (!nbrs[txface].txAble) )
                                                // if we don't have a neighbor,
                                                // or if we have so far been
                                                // unsuccessful txing to it,
      if( msgq_queueEmpty( txface ) )           // if the tx queue is empty,
        txPNGMsg( txface );                     // ping our neighbor
  }
}


/*****************************************************************************/
static void handleACKMsg
  ( int rcvface, unsigned int acked_type, unsigned int acked_seq_id )
{
  if( ( (acked_type == PNG_MSG) || (acked_type == LOC_MSG) ) &&
    ( nbrs[rcvface].pop == FALSE ) )              // if ACK in response to PNG
  {                                               // or LOC msg,
    if( (state == SOLIDIFIED) || (state == DISCARDED) )
                                                  // but if already solidified
                                                  // or discarded,
      return;                                     // return without action

                                                  // otherwise,
    nbrs[rcvface].pop = TRUE;                     // remember our new neighbor

    if( (rcvface == 4) || (rcvface == 5) || (rcvface == 6) )
    {                                             // if nghbr touches active face,
      if( setMagswitchState( rcvface, MSS_ON ) )  // if able to connect magswitch,
      {
        nbrs[rcvface].con = TRUE;                 // indicate it
        txMSSMsg( MSS_ON, rcvface );              // and let neighbor know
      }
      else                                        // otherwise, unable to connect,
      {
        nbrs[rcvface].con = FALSE;                // indicate it
        txMSSMsg( MSS_OFF, rcvface );             // and let neighbor know
      }
    }
  }
}
```

```
/*******************************************************************************/
static void handleDASMsg( int rcvface, unsigned int seq_id )
{
  int txface;

  txACKMsg( DAS_MSG, seq_id, rcvface );        // transmit an ACK msg

  if( state == LOCALIZED )                     // if already localized,
    state = DISCARDED;                         // we haven't rcvd INC msg, so
                                               // we're now 'discarded'
  else if( state == SCULPTED )                 // if we did rcv INC msg,
    state = SOLIDIFIED;                        // we're now 'solidified'
  else                                         // otherwise,
    return;                                    // return

  for( txface = 1; txface <= 6; txface++ )     // for each face,
  {
    if( txface != rcvface )                    // if face is not rcving face, &
      txDASMsg( txface );                      // fwd DAS msg
  }

  if( state == DISCARDED )                     // if not part of final struct.,
    disconFromStructure();                     // disconnect from the structure
}


/*******************************************************************************/
static void handleDCAMsg( int rcvface, unsigned int seq_id )
{
  /*
  handleDCAMsg is called whenever a "DisConnect All" message is received.  It
  sends an ACK message on the face on which the DCA message was received and
  forwards the DCA message to every other face.  Then it calls
  disconFromStructure() to turn off all of the cube's MagSwitches.
  */

  int txface;
  char raw_msg[MAX_MSG_LENGTH+ 1];


  txACKMsg( DCA_MSG, seq_id, rcvface );        // tx ACK msg

  state = DISCARDED;                           // update our state

  for( txface = 1; txface <= 6; txface++ )     // for faces 1-6,
  {
    if( txface != rcvface )                    // if face isn't receiving face,
      txDCAMsg( txface );                      // forward the DCA message
  }

  disconFromStructure();                       // disconnect from the structure

  /* the following code is new as of 5/17/06 */

  measureSeconds(6);                           // pause for 2 seconds
  while( !isTimeExpired() );

  for( txface = 0; txface <= 6; txface++ )     // clean the PSoC RX buffers
    readPSoCRxQueue( txface, raw_msg );
}
```

```
/*****************************************************************************/
static void handleDRQMsg ( int rcvface, unsigned int seq_id )
{
  static int prevSeqID[3] = {10000, 10000, 10000};
                                            // init. seq id's are invalid

  txACKMsg( DRQ_MSG, seq_id, rcvface );        // send an ACK message

  if( (rcvface != 4) && (rcvface != 5) && (rcvface != 6) )
                                            // if face lacks magsswitch,
    return;                                 // return without doing anything

  if( prevSeqID[rcvface - 4] == seq_id )       // if msg is duplicate of prev,
    return;                                 // ignore it and return

  prevSeqID[rcvface - 4] = seq_id;             // save new msg for next time

  if( setMagswitchState( rcvface, MSS_OFF ) )  // if able to discon. magswitch,
    txMSSMsg( MSS_OFF, rcvface );             // tell nghbr rqst was honored
  else                                      // else, unable to disconnect:
    txMSSMsg( MSS_ON, rcvface );              // tell nghbr rqst unsuccessful
}


/*****************************************************************************/
static void handleINCMsg
  ( int rcvface, unsigned int seq_id, unsigned int hops, unsigned int bd )
{
  int txface, msg_type;
  char *queued_msg;

  if( (state == LOCALIZED) || (state == SCULPTED) )
                                            // if we were localized,
    state = SCULPTED;                       // we've now been 'sculpted'
  else                                      // otherwise,
  {
    txACKMsg( INC_MSG, seq_id, rcvface );      // tx ACK msg to source of INC
    return;                                 // return w/out changing state
  }

  /* the following is only executed if we received an INC msg while localized */

  /* first, we will clear the TX queues of any pending msg that is not a INC or
     REF msg--doing so should produce improvements in system responsiveness
     because we will not waste time transmitting LOC messages to non-existent
     neighbors when we could be forwarding INC messages */

  for( txface = 1; txface <= 6; txface++ )     // for each face,
  {
    if( msgq_getQueueContents( &queued_msg, txface ) )
    {                                       // if there exists queued msg,
      msg_type = getMsgType( queued_msg );     // get its type
      if( (msg_type != REF_MSG) && (msg_type != INC_MSG) )
                                            // if its not a REF nor INC msg,
        msgq_purgeQueue( txface );            // purge the queue to stop its
    }                                       // transmission
  }

  /* the following code either includes this cube in the structure or forwards
```

```
    the INC message to one of our neighbors */

  if( (hops == 0) && (bd == 0) )                // if we are final dest of msg,
  {
    solidLED( 75 );                             // illuminate our User LED,
    txACKMsg( INC_MSG, seq_id, rcvface );       // tx ACK msg to source of INC,
    txREFMsg( INC_MSG, settings.uid, nbrs[ME].p );
                                                // and tx REF msg to our parent
  }

  else if( (hops == 1) && msgq_queueEmpty( bd ) )
  {                                             // if dest is direct nghbr,
                                                // and that tx queue is empty,
    incChainPtr = bd;                           // update our inc. chain ptr
    if( fwdINCMsg( 0, 0, incChainPtr ) )        // if able to forward INC msg,
      txACKMsg( INC_MSG, seq_id, rcvface );     // tx ACK msg to source of INC
  }

  else if( (hops >= 2) && (incChainPtr != 0) && msgq_queueEmpty( incChainPtr ) )
  {                                             // if dest not direct nghbr,
                                                // our inc.chain ptr is valid, &
                                                // tx queue on inc. chain ptr
                                                // face is empty,
    if( fwdINCMsg( hops - 1, bd, incChainPtr ) )// if able to forward INC msg,
      txACKMsg( INC_MSG, seq_id, rcvface );     // tx ACK msg to source of INC
  }

  else if( incChainPtr == 0 )                   // if inc. chain ptr invalid,
    logEvent( INVALID_INC_MSG, "no incChainPtr for face %d msg", rcvface);
                                                // log an error
}


/*****************************************************************************/
static void handleIRTMsg( int rcvface, unsigned int seq_id )
{
  /*
  handleIRTMsg is called whenever a "InfraRed Test" message is received.  IRT
  messages are used to test the ability of each cube to transmit and receive IR
  messags on each face.  The routine sends an ACK message on the face on which
  the IRT message was received and forwards the IRT message to every other face.
  In addition, the the routine illuminates the cube's User LED for one second so
  that it is possible to tell that the IRT message was received.
  */

  int txface;

  txACKMsg( IRT_MSG, seq_id, rcvface );         // tx an ACK msg

  if( (state == ASSEMBLING) || (state == IRTEST) )
    state = IRTEST;                             // switch to 'IR test' state
  else                                          // for any other state,
    return;                                     // return without action

  solidLED(128);                                // turn the LED on at half power
  ledOnForIRT = TRUE;                           // signal why User LED is on
  measureSeconds(1);                            // start the countdown timer

  for( txface = 1; txface <= 6; txface++ ) {    // for every face:
    if( (txface != rcvface) )                   // if face isn't receiving face,
      txIRTMsg( txface );                       // forward the IRT message
  }
}


/******************************************************************************/
static void handleLOCMsg( int rcvface, unsigned int seq_id, struct point *dest )
{
  int txface;

  txACKMsg( LOC_MSG, seq_id, rcvface );         // tx an ACK msg

  /* first, if LOC came from unknown neighbor, connect */

  if( (!nbrs[rcvface].pop ) )
  {                                              // if prev unaware of neighbor,
    nbrs[rcvface].pop = TRUE;                    // remember that we have nghbr
    if( (rcvface == 4) || (rcvface == 5) || (rcvface == 6) )
    {                                           // if nghbr touches active face,
      if( setMagswitchState( rcvface, MSS_ON ) )// if able to connect magswitch,
      {
        nbrs[rcvface].con = TRUE;               // indicate it
        txMSSMsg( MSS_ON, rcvface );            // and let neighbor know
      }
      else {                                    // otherwise, unable to connect,
        nbrs[rcvface].con = FALSE;              // indicate it
        txMSSMsg( MSS_OFF, rcvface );           // and let neighbor know
      }
    }
  }

  /* now, deal with specific repercusions of the LOC msg */

  if( (state == LOCALIZED) || (state == SCULPTED) ||
      (state == SOLIDIFIED) || ( state == DISCARDED) )
  {                                             // if we not in a state in which
                                                // LOC msgs are relevant,
    if( addrcmp( &nbrs[ME].addr, dest ) == FALSE )
                                                // and if LOC msgs doesn't match
                                                // our assumed position,
      logEvent( LOC_DISCREPENCY, 0 );           // log an error
    return;                                     // return regardless
  }

  else if( (state == ASSEMBLING) || (state == IRTEST) )
  {                                             // if we are not localized,
    if( nbrs[rcvface].txAble )                  // and if we can successfully
                                                // transmit msgs on the face on
                                                // which the LOC msg was rcv'd
    {
      state = LOCALIZED;                        // we're now localized,

      nbrs[ME].addr = *dest;                    // our new addr is dest of msg
      nbrs[ME].pop = TRUE;                      // our address is now populated
      nbrs[ME].p = rcvface;                     // parent is rcving face
      nbrs[nbrs[ME].p].pop = TRUE;              // parent must be populated

      genNeighborAddresses();                   // generate addrs of neighbors

      for( txface = 0; txface <= 6; txface++ )  // scan through all faces
      {
        if( txface == nbrs[ME].p )              // when hit parent/rcving face,
          txREFMsg( LOC_MSG, settings.uid, txface );
                                          // transmit REF msg,
        else if( txface != 0 )            // for other faces xcept uplink,
          txLOCMsg( txface );             // forward LOC msg
      }
      blinkLED(382400, 20);               // indicate we're localized
    }
  }
}


/******************************************************************************/
static void handleMSSMsg( int rcvface, unsigned int seq_id, unsigned int mss )
{
  txACKMsg( MSS_MSG, seq_id, rcvface );         // and tx an ACK msg
  nbrs[rcvface].con = mss;                      // update connection status
}


/******************************************************************************/
static void handleNCKMsg( int rcvface, unsigned int seq_id )
{
  static unsigned int prev_nck_seq_id[7];

  if ( seq_id != prev_nck_seq_id[rcvface] )     // if new SeqID is not same as
                                                // that of last NCK on rcvface,
    msgq_incAttempts( rcvface, 5 );             // inc number of re-tx atttempts

  /*
  Checking to see that newest NCK is different from previous prevents deadlock
  in which two cubes send nothing but NCK to each other.  This should have only
  been an issue in the old system in which (A/N)CK msgs were treated like any
  other msg.  In the new system, (A/N)CK are only transmitted once for each
  faulty rcvd msg.  As a result, rcving a NCK would never cause a NCK to be
  retransmitted.  Keeping the shouldn't harm anything and helps prevent one
  message from being transmitted forever.
  */

  prev_nck_seq_id[rcvface] = seq_id;            // update
}


/******************************************************************************/
static void handlePNGMsg( int rcvface, unsigned int seq_id )
{
  txACKMsg( PNG_MSG, seq_id, rcvface );         // send ACK immediately

  if( nbrs[rcvface].pop )                       // if aleady aware of neighbor
    return;                                     // simply return

  /* the following is only executed if new neighbor was previously unknown */

  nbrs[rcvface].pop = TRUE;                     // remember that we have nghbr

  if( (rcvface == 4) || (rcvface == 5) || (rcvface == 6) )
  {                                             // if nghbr touches active face,
    if( setMagswitchState( rcvface, MSS_ON ) )  // if able to connect magswitch,
    {
      nbrs[rcvface].con = TRUE;                 // indicate it
      txMSSMsg( MSS_ON, rcvface );              // and let neighbor know
    }
    else {                                      // otherwise, unable to connect,
      nbrs[rcvface].con = FALSE;                // indicate it
      txMSSMsg( MSS_OFF, rcvface );             // and let neighbor know
    }
  }
}


/*****************************************************************************/
static void handleREFMsg
  ( int rcvface, unsigned int type, unsigned int seq_id,
                           struct point *src_p, unsigned int uid, char *rnbrs  )
{
  struct point queued_src;
  unsigned int queued_type, queued_uid;
  char *queued_msg;

  if( msgq_queueEmpty( nbrs[ME].p ) )           // if tx queue on parent face is
  {                                             // empty,
    txACKMsg( REF_MSG, seq_id, rcvface );       // tx ACK to sender of REF msg.,
    fwdREFMsg( type, src_p, uid, rnbrs, nbrs[ME].p );
                                                // and forward REF msg to parent
    return;
  }

  /* below code ACKs rcvd REF msg if that msg matches msg in parent queue */

  if( msgq_getQueueContents( &queued_msg, nbrs[ME].p ) )
  {                                             // if able to get msg queued for
                                                // tx to parent,
    getMsgRefType( queued_msg, &queued_type );  // get type of queued msg,
    getMsgSrc( queued_msg, &queued_src );       // get src. of  queued msg, and
    getMsgUID( queued_msg, &queued_uid );       // get UID of source cube

    if( addrcmp( src_p, &queued_src ) &&        // if src's match,
             (uid == queued_uid) &&             // if uid's match, and
             (type == queued_type) )            // if type's match,
      txACKMsg( REF_MSG, seq_id, rcvface );     // tx an ACK on rcving face
  }
}


/*****************************************************************************/
static void handleRSTMsg( int rcvface, unsigned int seq_id )
{
  int txface;
  char raw_msg[MAX_MSG_LENGTH+ 1];
  unsigned int totalTxCount;

  totalTxCount = msgq_getTotalTxCount();        // log total # msgs sent/rcvd
  logEvent( 0, "Tx Total: %d,  Rx Total: %d", totalTxCount, totalRxCount );
  msgq_clearTotalTxCount();                     // clear the totals
  totalRxCount = 0;

  nbrs[ME].addr.x = 0;                          // forget our location
  nbrs[ME].addr.y = 0;
  nbrs[ME].addr.z = 0;
  nbrs[ME].pop = FALSE;                         // forget that we're populated
  nbrs[ME].p = 0;                               // forget our parent

  for( txface = 1; txface <= 6; txface++ )
  {
    if( txface != rcvface )
      txRSTMsg( txface );
    else
      msgq_purgeQueue( txface );

    nbrs[txface].addr.x = 0;                  // forget neighbor's location
    nbrs[txface].addr.y = 0;
    nbrs[txface].addr.z = 0;
    nbrs[txface].pop = FALSE;                 // forget that neighbor exists
    nbrs[txface].con = FALSE;                 // this is a lie--doesn't matter
    nbrs[txface].p = 0;                       // forget our neighbor's parent
    nbrs[txface].txAble = FALSE;              // forget transmit ability
    nbrs[txface].rxAble = FALSE;              // forget receive ability
  }

  state = ASSEMBLING;

  blankLED();

  measureSeconds(6);                          // pause for 2 seconds
  while( !isTimeExpired() );

  for( txface = 0; txface <= 6; txface++ )    // clean the PSoC RX buffers
    readPSoCRxQueue( txface, raw_msg );


}


/******************************************************************************/
static void handleRTCMsg( int rcvface, unsigned int seq_id, struct time* t )
{
  int txface;

  txACKMsg( RTC_MSG, seq_id, rcvface );       // transmit an ACK msg

  if( !timeValid )                            // if our RTC is invalid
  {
    setTime( t );                             // set the clock
    for( txface = 1; txface <= 6; txface++ )  // for each face
    {
      if( txface != rcvface )                 // if not rcving face,
        txRTCMsg( t, txface );                // forward RTC msg
    }
  }
}


/******************************************************************************/
static void disconFromStructure() {
  int face;

  nbrs[ME].addr.x = 0;                        // forget our location
  nbrs[ME].addr.y = 0;
  nbrs[ME].addr.z = 0;
  nbrs[ME].pop = FALSE;                       // forget that we're populated
  nbrs[ME].p = 0;                             // forget our parent

  for( face = 1; face <= 6; face++ )          // for each potential neighbor,
  {
    if( nbrs[face].pop )                        // if neighbor exists,
    {
      if( (face==1) || (face==2) || (face==3) ) // if ngbr in question is 1,2,3
      {
        msgq_purgeQueue( face );                // immediately tx DRQ msg
        txDRQMsg( face );
      }
      else if( ((face==4) || (face==5) || (face==6)) &&
        (setMagswitchState( face, MSS_OFF )) )  // if ngbr in question is 4,5,6
      {                                         // and can turn magswtch off,
        nbrs[face].con = FALSE;                 // remember that it is off,
        msgq_purgeQueue( face );                // and immediately tx MSS msg
        txMSSMsg( MSS_OFF, face );
      }
    }

    nbrs[face].addr.x = 0;                      // forget neighbor's location
    nbrs[face].addr.y = 0;
    nbrs[face].addr.z = 0;
    nbrs[face].pop = FALSE;                     // forget that neighbor exists
    nbrs[face].p = 0;                           // forget our neighbor's parent
    nbrs[face].txAble = FALSE;                  // forget transmit ability
    nbrs[face].rxAble = FALSE;                  // forget receive ability
  }

  blankLED();                                   // turn LED off
}


/******************************************************************************/
static void genNeighborAddresses() {
  /*
  genNeighborAddresses examines our address (stored in nbrs[ME].addr) and
  computes correct address for each of our six neighbors: nbrs[1-6].addr.
  */

  int face;

  for( face = 1; face <= 6; face++ )           // for each of our six neighbors,
    nbrs[face].addr = nbrs[ME].addr;           // begin by assuming their address is our addres,

  nbrs[RIGHT].addr.x = nbrs[ME].addr.x + 1;    // then, for our right neighbor, increment x coordinate by 1
  nbrs[LEFT].addr.x = nbrs[ME].addr.x - 1;     // for our left neighbor, decrement x coordinate by 1
  nbrs[FRONT].addr.y = nbrs[ME].addr.y + 1;    // for our front neighbor, increment y coordinate by 1
  nbrs[BACK].addr.y = nbrs[ME].addr.y - 1;     // for our back neighbor, decrement y coordinate by 1
  nbrs[TOP].addr.z = nbrs[ME].addr.z + 1;      // for our top neighbor, increment z coordinate by 1
  nbrs[BOTTOM].addr.z = nbrs[ME].addr.z - 1;   // for our bottom neighbor, decrement z coordinate by 1
}


/******************************************************************************/
int addrcmpRaw( struct point* l, int x, int y, int z ) {
  if( (l->x == x) && (l->y == y) && (l->z == z) )
    return TRUE;
  else
    return FALSE;
}


/******************************************************************************/
static int addrcmp(struct point* l, struct point* m) {
  if ( (l->x == m->x) && (l->y == m->y) && (l->z == m->z) )
    return TRUE;
  else
    return FALSE;
}
```

B.3 txmsg.c

This section of code constructs outgoing messages and sends them to the transmission buffer.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include "miche.h"
#include <string.h>

#include "msgs/handlemsg.h"
#include "hal/rtc.h"

#define _txmsg_
#include "msgs/txmsg.h"
#undef _txmsg_

#include "hal/psoc.h"
#include "msgs/msgs.h"
#include "msgqueue.h"
#include "crc.h"
#include "main.h"
#include "debug.h"

unsigned int seq_id = 0;


/******************************************************************************/
unsigned char txACKMsg( int init_msg_type, unsigned int init_seq_id, int face) {

  char msg[MAX_MSG_LENGTH + 1];                // +1 accounts for null terminator
  char init_msg_type_string[TYPE_FIELD_LENGTH + 2];

  if( init_seq_id > 10000 )
    return FALSE;

  switch (init_msg_type) {
    case ACK_MSG: strcpy(init_msg_type_string, "ACK&"); break;
    case DAS_MSG: strcpy(init_msg_type_string, "DAS&"); break;
    case DCA_MSG: strcpy(init_msg_type_string, "DCA&"); break;
    case DRQ_MSG: strcpy(init_msg_type_string, "DRQ&"); break;
    case INC_MSG: strcpy(init_msg_type_string, "INC&"); break;
    case LOC_MSG: strcpy(init_msg_type_string, "LOC&"); break;
    case MSS_MSG: strcpy(init_msg_type_string, "MSS&"); break;
    case NCK_MSG: strcpy(init_msg_type_string, "NCK&"); break;
    case PNG_MSG: strcpy(init_msg_type_string, "PNG&"); break;
    case RST_MSG: strcpy(init_msg_type_string, "RST&"); break;
    case REF_MSG: strcpy(init_msg_type_string, "REF&"); break;
    default:      return FALSE;                      break;
  }

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "ACK&" );
  strcat( msg, init_msg_type_string );
  sprintf( &msg[strlen(msg)], "%d&", init_seq_id );

  return ( txMsg(msg, face) );
}


/*****************************************************************************/
unsigned char txDASMsg( int face ) {
  char msg[MAX_MSG_LENGTH + 1];

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "DAS&" );

  return ( txMsg(msg, face) );
}


/*****************************************************************************/
unsigned char txDCAMsg( int face ) {
  char msg[MAX_MSG_LENGTH + 1];

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "DCA&" );

  return ( txMsg(msg, face) );
}


/*****************************************************************************/
unsigned char txDRQMsg( int face ) {
  char msg[MAX_MSG_LENGTH + 1];

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "DRQ&" );
  return ( txMsg(msg, face) );
}


/*****************************************************************************/
unsigned char fwdINCMsg( int hops, char bd, int face) {
  char msg[MAX_MSG_LENGTH + 1];

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "INC&" );
  sprintf( &msg[strlen(msg)], "%d&", hops);
  sprintf( &msg[strlen(msg)], "%d&", bd);

  return ( txMsg(msg, face) );
}


/*****************************************************************************/
unsigned char txIRTMsg( int face ) {
  char msg[MAX_MSG_LENGTH + 1];
  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "IRT&" );

  return( txMsg( msg, face ) );
}


/******************************************************************************/
unsigned char txLOCMsg( int face ) {
  char msg[MAX_MSG_LENGTH + 1];                 // +1 accounts for null terminator

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "LOC&" );
  sprintf( &msg[strlen(msg)], "%d&%d&%d&",
                    nbrs[face].addr.x, nbrs[face].addr.y, nbrs[face].addr.z );

  return ( txMsg(msg, face) );
}


/******************************************************************************/
unsigned char txMSSMsg( unsigned char mss, int face) {
  char msg[MAX_MSG_LENGTH + 1];

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "MSS&" );
  sprintf( &msg[strlen(msg)], "%d&", mss );

  return ( txMsg( msg, face ) );
}


/******************************************************************************/
unsigned char txNCKMsg( int face ) {
  char msg[MAX_MSG_LENGTH + 1];                 // +1 accounts for null terminator

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "NCK&" );

  return ( txMsg(msg, face) );
}


/******************************************************************************/
unsigned char txPNGMsg( int face ) {
  char msg[MAX_MSG_LENGTH + 1];

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "PNG&" );

  return ( txMsg(msg, face) );
}


/******************************************************************************/
```

```
unsigned char txREFMsg( int type, int uid, int face )
{
  char msg[MAX_MSG_LENGTH + 1];
  char type_str[TYPE_FIELD_LENGTH + 2];
  char rnbrs[7] = {0,0,0,0,0,0,0};
  int i;

  switch( type )
  {
    case INC_MSG: strcpy( type_str, "INC&"); break;
    case LOC_MSG: strcpy( type_str, "LOC&"); break;
    default:      return FALSE;              break;
  }

  for( i = 1; i <= 6; i++ )
  {
    if( nbrs[i].txAble )
      sprintf( &rnbrs[strlen(rnbrs)], "%d", i );
  }

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "REF&" );
  strcat( msg, type_str );
  sprintf( &msg[strlen(msg)], "%d&%d&%d&",
                          nbrs[ME].addr.x, nbrs[ME].addr.y, nbrs[ME].addr.z );
  sprintf( &msg[strlen(msg)], "%d&", uid);
  sprintf( &msg[strlen(msg)], "%s&", rnbrs );

  return ( txMsg(msg, face) );
}


/******************************************************************************/
unsigned char fwdREFMsg
  ( int type, struct point* src, int uid, char* rnbrs, int face )
{
  char msg[MAX_MSG_LENGTH + 1];
  char type_str[TYPE_FIELD_LENGTH + 2];

  switch( type )
  {
    case INC_MSG: strcpy( type_str, "INC&"); break;
    case LOC_MSG: strcpy( type_str, "LOC&"); break;
    default:      return FALSE;              break;
  }

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "REF&" );
  strcat( msg, type_str );
  sprintf( &msg[strlen(msg)], "%d&%d&%d&", src->x, src->y, src->z );
  sprintf( &msg[strlen(msg)], "%d&", uid);
  sprintf( &msg[strlen(msg)], "%s&", rnbrs );

  return ( txMsg(msg, face) );
}

/******************************************************************************/
unsigned char txRSTMsg( unsigned int face ) {
  char msg[MAX_MSG_LENGTH + 1];
  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "RST&" );

  return ( txMsg(msg, face) );
}

/******************************************************************************/
unsigned char txRTCMsg( struct time* t, unsigned int face ) {
  char msg[MAX_MSG_LENGTH + 1];

  if( !validateTime( t ) )
    return FALSE;

  strcpy( msg, "#0&" );
  sprintf( &msg[strlen(msg)], "%d&", seq_id++ % 10000 );
  sprintf( &msg[strlen(msg)], "RTC&" );
  sprintf( &msg[strlen(msg)], "%d&%d&%d&%d&%d&%d&%d&%d&",
           t->year, t->month, t->doy, t->dow, t->dom, t->hour, t->min, t->sec);

  return( txMsg( msg, face ) );
}


/******************************************************************************/
unsigned char txMsg( char* msg, unsigned int face ) {

  if( face > 6 )                            // transmitting on face 0 is txing only on debug uplink
    return FALSE;

  msg[1] = face + 48;                         // tack transmitting face on to start of message

  sprintf( &msg[strlen(msg)], "%x\n", calcCRC( msg, strlen(msg) ) );
                                            // attach the checksum and message terminator (/r)



  if( debugging ) {
    pause( peripheralClockFrequency() / 9000 ); // pause before turning on mux in PSoC to prevent
                                            // tail end of previous character being transmitted
                                            // from being tacked on to the head of the message
                                            // to be transmitted

    if( switchTxFace( face ) ) {            // if PSoC responds to request to switch mux to requested face,
      printf( "%s", msg );                  // send the message
      pause( peripheralClockFrequency() / 500 );// again, wait for the RS-232 to finish before switching mux
      switchTxFace( 0 );                    // set mux to not forward messages to any face
    }
    else {                                  // attempt to switch face unsuccessful,
      switchTxFace(0);                      // try and switch off mux just for good measure
      return;                               // and return failure
    }
  }
  else                                      // otherwise, we're not debugging
    if( getMsgType( msg ) == RST_MSG )      // only tx RST msgs 5 times
      return( msgq_enQueue( msg, 5, face ) );
    else if( getMsgType( msg ) == DCA_MSG )   // only tx DCA msgs 5 times
      return (msgq_enQueue( msg, 5, face ) );
    else
      return( msgq_enQueue( msg, settings.repeatCount, face ) );
                                            // so add the message to a queue
```

```
}
```

B.4 crc.c

This code computes the CRC checksums that are appended to each inter–module message.

```
/*
This code based on code written by Sven Reifegerste and
the excellent CRC tutorial, "A Painless Guide to CRC
Error Detection Algorithms," written by Ross N. Williams
*/

#include <targets/LPC210x.h>
#include "boolean.h"

#define _crc_
#include "crc.h"
#undef _crc_

unsigned int crc_table[256];


/*******************************************************************************/
void generateCRCTable() {
  unsigned int crc, bit, i;
  unsigned char j;

  for (i=0; i<256; i++) {
    crc = (unsigned int)i;
    crc <<= 8;

    for (j=0; j<8; j++) {
      bit = crc & 0x8000;
      crc <<= 1;
      if (bit)
  crc ^= polynomial;
    }

    crc_table[i] = (crc & 0xFFFF);
  }
}


/*******************************************************************************/
unsigned int calcCRC (unsigned char* p, unsigned int len) {

  unsigned int crc = initial_crc;

  while (len--)
    crc = ((crc << 8) | *p++) ^ crc_table[(crc >> 8) & 0xFF];

  crc = (crc << 8) ^ crc_table[(crc >> 8) & 0xFF];
  crc = (crc << 8) ^ crc_table[(crc >> 8) & 0xFF];

  return(crc & 0xFFFF);
}
```

B.5 parsemsg.c

This section, given a received message, extracts the data fields.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include <string.h>

#include "handlemsg.h"
#include "hal/rtc.h"

#include "msgs.h"

#define _parsemsg_
#include "parsemsg.h"
#undef _parsemsg_

#include "hal/psoc.h"
#include "msgqueue.h"
#include "log.h"

#include "crc.h"


/************************* Methods for all messages *************************/
/****************************************************************************/
unsigned char getMsgSeqID( char* raw_msg, unsigned int* seq_id ) {
  if( sscanf( getFieldNPtr( raw_msg, SEQ_ID_FIELD_NUMBER ), "%d&", seq_id ) )
    return TRUE;

  logEvent( PARSE_FAILURE_SEQ_ID, "%s", raw_msg );
  return FALSE;
}


/****************************************************************************/
unsigned int getMsgType( char* raw_msg ) {
  char* type_str_p;
  char type_str[TYPE_FIELD_LENGTH + 1];

  type_str_p = getFieldNPtr( raw_msg, TYPE_FIELD_NUMBER );
                                              // get a pointer to the message type field
  strncpy( type_str, type_str_p, TYPE_FIELD_LENGTH );
                                              // copy the message type field into its its own string
  type_str[TYPE_FIELD_LENGTH] = 0;            // null terminate the string

  if ( strcmp(type_str, "ACK") == 0 )
    return ACK_MSG;
  else if ( strcmp(type_str, "DAS") == 0 )
    return DAS_MSG;
  else if ( strcmp(type_str, "DCA") == 0 )
    return DCA_MSG;
  else if ( strcmp(type_str, "DRQ") == 0 )
    return DRQ_MSG;
  else if ( strcmp(type_str, "INC") == 0 )
    return INC_MSG;
  else if ( strcmp(type_str, "IRT") == 0 )
    return IRT_MSG;
  else if ( strcmp(type_str, "LOC") == 0 )
    return LOC_MSG;
  else if ( strcmp(type_str, "MSS") == 0 )
```

```
    return MSS_MSG;
  else if ( strcmp(type_str, "NCK") == 0 )
    return NCK_MSG;
  else if ( strcmp(type_str, "PNG") == 0 )
    return PNG_MSG;
  else if ( strcmp(type_str, "REF") == 0 )
    return REF_MSG;
  else if ( strcmp(type_str, "RST") == 0 )
    return RST_MSG;
  else if ( strcmp(type_str, "RTC") == 0 )
    return RTC_MSG;
  else
  {
    logEvent( PARSE_FAILURE_TYPE, "%s", raw_msg );
    return UNKNOWN_MSG;
  }
}


/************************* Methods for ACK messages *************************/
/****************************************************************************/
unsigned char getMsgAckedSeqID( char* raw_msg, unsigned int* acked_seq_id )
{
  if( getMsgType( raw_msg ) == ACK_MSG )
    if( sscanf( getFieldNPtr( raw_msg, INIT_SEQ_ID_FIELD_NUMBER ),
                                                      "%d&", acked_seq_id ) )
      return TRUE;

  logEvent( PARSE_FAILURE_ACKED_SEQ_ID, "%s", raw_msg );
  return FALSE;
}


/****************************************************************************/
unsigned int getMsgAckedType(char* raw_msg)
{
  char* init_type_str_p;
  char init_type_str[INIT_TYPE_FIELD_LENGTH + 1] = "";

  if( getMsgType( raw_msg ) == ACK_MSG ) {
    init_type_str_p = getFieldNPtr( raw_msg, INIT_TYPE_FIELD_NUMBER );
    strncpy( init_type_str, init_type_str_p, INIT_TYPE_FIELD_LENGTH);
    init_type_str[INIT_TYPE_FIELD_LENGTH] = 0;

    if ( strcmp(init_type_str, "ACK") == 0 )
      return ACK_MSG;
    else if ( strcmp(init_type_str, "DAS") == 0 )
      return DAS_MSG;
    else if ( strcmp(init_type_str, "DCA") == 0 )
      return DCA_MSG;
    else if ( strcmp(init_type_str, "DRQ") == 0 )
      return DRQ_MSG;
    else if ( strcmp(init_type_str, "INC") == 0 )
      return INC_MSG;
    else if ( strcmp(init_type_str, "IRT") == 0 )
      return IRT_MSG;
    else if ( strcmp(init_type_str, "LOC") == 0 )
      return LOC_MSG;
    else if ( strcmp(init_type_str, "MSS") == 0 )
      return MSS_MSG;
    else if ( strcmp(init_type_str, "NCK") == 0 )
```

```
      return NCK_MSG;
    else if ( strcmp(init_type_str, "PNG") == 0 )
      return PNG_MSG;
    else if ( strcmp(init_type_str, "REF") == 0 )
      return REF_MSG;
    else if ( strcmp(init_type_str, "RTC") == 0 )
      return RTC_MSG;
    else
    {
      logEvent( PARSE_FAILURE_ACKED_TYPE, "%s", raw_msg );
      return UNKNOWN_MSG;
    }
  }
}


/************************* Methods for INC Messages *************************/
/****************************************************************************/
unsigned char getMsgHops(char* raw_msg, unsigned int* hops)
{
  if ( getMsgType(raw_msg) == INC_MSG )
    if( sscanf( getFieldNPtr( raw_msg, HOPS_FIELD_NUMBER ), "%d", hops ) == 1 )
      return TRUE;

  logEvent( PARSE_FAILURE_HOPS, "%s", raw_msg );
  return FALSE;
}


/****************************************************************************/
unsigned char getMsgBranchDirection(char* raw_msg, unsigned int* bd)
{
  if ( getMsgType(raw_msg) == INC_MSG )
    if( sscanf( getFieldNPtr( raw_msg, BD_FIELD_NUMBER ), "%d", bd ) == 1 )
      return TRUE;

  logEvent( PARSE_FAILURE_BD, "%s", raw_msg );
  return FALSE;
}


/************************* Methods for LOC messages *************************/
/****************************************************************************/
unsigned char getMsgDest(char* raw_msg, struct point* dest)
{
  if( getMsgType(raw_msg) == LOC_MSG )
    if( sscanf( getFieldNPtr(raw_msg, DEST_FIELD_NUMBER), "%d&%d&%d&",
                                    &(dest->x),&(dest->y),&(dest->z) ) == 3 )
      return TRUE;

  logEvent( PARSE_FAILURE_DEST, "%s", raw_msg );
  return FALSE;
}


/************************* Methods for MSS Messages *************************/
/****************************************************************************/
unsigned char getMsgMSStatus( char* raw_msg, unsigned char* mss )
{
  if( getMsgType( raw_msg ) == MSS_MSG )
    if( sscanf( getFieldNPtr( raw_msg, MSS_FIELD_NUMBER ), "%d", mss ) == 1)
```

```
      return TRUE;

  logEvent( PARSE_FAILURE_MSS, "%s", raw_msg );               // log failure
  return FALSE;
}


/************************* Methods for REF Messages *************************/
/****************************************************************************/
unsigned char getMsgRefType( char* raw_msg, unsigned int* type )
{
  char type_str[TYPE_FIELD_LENGTH + 1] = "";

  if( getMsgType( raw_msg ) == REF_MSG )
  {
    strncpy( type_str,
         getFieldNPtr( raw_msg, REF_TYPE_FIELD_NUMBER ), REF_TYPE_FIELD_LENGTH);
    type_str[REF_TYPE_FIELD_LENGTH] = 0;        // null terminate the string

    if ( strcmp(type_str, "INC") == 0 )
    {
      *type = INC_MSG;
      return TRUE;
    }
    else if ( strcmp(type_str, "LOC") == 0 )
    {
      *type = LOC_MSG;
      return TRUE;
    }
    else
    {
      logEvent( PARSE_FAILURE_ACKED_TYPE, "%s", raw_msg );
      *type = UNKNOWN_MSG;
      return FALSE;
    }
  }

  return FALSE;
}

/****************************************************************************/
unsigned char getMsgSrc(char* raw_msg, struct point* src)
{
  if( getMsgType(raw_msg) == REF_MSG )
    if( sscanf( getFieldNPtr(raw_msg, SRC_FIELD_NUMBER), "%d&%d&%d&",
                                       &(src->x),&(src->y),&(src->z) ) == 3 )
      return TRUE;

  logEvent( PARSE_FAILURE_SRC, "%s", raw_msg );               // log failure
  return FALSE;
}


/****************************************************************************/
unsigned char getMsgUID(char* raw_msg, unsigned int* UID)
{
  if ( getMsgType(raw_msg) == REF_MSG )
    if( sscanf( getFieldNPtr( raw_msg, UID_FIELD_NUMBER ), "%d", UID) == 1 )
      return TRUE;

  logEvent( PARSE_FAILURE_UID, "%s", raw_msg );               // log failure
  return FALSE;
}


/******************************************************************************/
unsigned char getMsgReachableNbrs( char* raw_msg, char* nbrs )
{
  int nbrs_int;

  if( getMsgType( raw_msg ) == REF_MSG )
  {
    nbrs[0] = 0;
    nbrs[1] = 0;
    nbrs[2] = 0;
    nbrs[3] = 0;
    nbrs[4] = 0;
    nbrs[5] = 0;
    nbrs[6] = 0;

    if( sscanf(
        getFieldNPtr( raw_msg, RNBRS_FIELD_NUMBER ), "%6d", &nbrs_int ) == 0 )
      return TRUE;

    sprintf( nbrs, "%d", nbrs_int );
    return TRUE;
  }

  return FALSE;
}


/************************* Methods for RTC Messages *************************/
/******************************************************************************/
unsigned char getMsgTime( char* raw_msg, struct time* t )
{
  char* fieldptr;

  if( getMsgType( raw_msg ) == RTC_MSG ) {

    fieldptr = getFieldNPtr( raw_msg, YEAR_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->year ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    fieldptr = getFieldNPtr( raw_msg, MONTH_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->month ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    fieldptr = getFieldNPtr( raw_msg, DOY_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->doy ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    fieldptr = getFieldNPtr( raw_msg, DOW_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->dow ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    fieldptr = getFieldNPtr( raw_msg, DOM_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->dom ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    fieldptr = getFieldNPtr( raw_msg, HOUR_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->hour ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    fieldptr = getFieldNPtr( raw_msg, MIN_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->min ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    fieldptr = getFieldNPtr( raw_msg, SEC_FIELD_NUMBER );
    if( !sscanf( fieldptr, "%d", &t->sec ) )
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }

    if( validateTime( t ) )
      return TRUE;
    else
    {
      logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
      return FALSE;
    }
  }

  logEvent( PARSE_FAILURE_TIME, "%s", raw_msg );
  return FALSE;
}


/***************************** Utility functions *****************************/
/*****************************************************************************/
char* getFieldNPtr( char* raw_msg, int n ) {
  int len;
  int i = 0;

  len = strlen(raw_msg);

  if( n > 0) {
    i = 0;

    while( (n > 0) && (i <= len - 1) ) {
      if( raw_msg[i] == FIELD_SEPARATOR )
```

```
        n--;
      i++;
    }
  }
  else if ( n < 0 ) {
    i = len - 1;

    while( (n < 0) && (i >= 0) ) {
      if( raw_msg[i] == FIELD_SEPARATOR )
        n++;
      i--;
    }

    if( i < 0 )
      i = len - 1;
    else
      i = i + 2;
  }

  return &raw_msg[i];
}


/*****************************************************************************/
unsigned char checkCRC(char* raw_msg) {
  unsigned char* rcvd_crc_str_p;
  unsigned int rcvd_crc;

  if( strlen( raw_msg ) < 5 )
    return FALSE;

  rcvd_crc_str_p = getFieldNPtr( raw_msg, -1 );
  sscanf( rcvd_crc_str_p, "%x", &rcvd_crc );

  if ( rcvd_crc == calcCRC(raw_msg,
                         (unsigned int)rcvd_crc_str_p - (unsigned int)raw_msg) )
    return TRUE;
  else
    return FALSE;
}
```

B.6 msgqueue.c

This is the code that implements the transmission buffer for each face.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include "miche.h"
#include "msgs/msgs.h"

#include "hal/psoc.h"

#define _msgqueue_
#include "msgqueue.h"
#undef _msgqueue_

#include "hal/i2c.h"
```

```
struct messageQueue msgq[7];
static unsigned int totalTxCount;
static int paused = FALSE;


/*****************************************************************************/
unsigned int msgq_getTotalTxCount( void )
{
  return totalTxCount;
}


/*****************************************************************************/
void msgq_clearTotalTxCount( void )
{
  totalTxCount = 0;
}


/*****************************************************************************/
int msgq_queueEmpty( unsigned int face )
{
  if( msgq[face].cnt == 0 )
    return TRUE;
  else
    return FALSE;
}


/*****************************************************************************/
int msgq_getQueueContents( char **msg, unsigned int face )
{
  if( face > 6 )
    return FALSE;

  if( msgq[face].cnt == 0 )
    return FALSE;

  *msg = msgq[face].msg;
  return TRUE;
}



/*****************************************************************************/
void msgq_incAttempts( unsigned int face, int increment )
{
  if( msgq[face].cnt == 0 )
    return;

  msgq[face].cnt += increment;
}


/*****************************************************************************/
void msgq_pauseTx()
{
  paused = TRUE;
}
```

```
/*****************************************************************************/
void msgq_restartTx()
{
  paused = FALSE;
}


/*****************************************************************************/
int msgq_enQueue( char* msg, unsigned int count, unsigned int face ) {
  int i;
  unsigned int type;

  if( face > 6 )
    return FALSE;

  type = getMsgType( msg );

  if( (type == ACK_MSG) || (type == NCK_MSG) )
  {
    sprintf( &msgq[face].ack_msg[0], "%s", msg );
    return TRUE;
  }

  /* this only happens if message to be enqueued in not a (N/A)CK message */

  msgq[face].cnt = 0;
  sprintf( msgq[face].msg, "%s", msg );
  msgq[face].cnt = count;
  getMsgSeqID( msg, &msgq[face].seq_id );
  msgq[face].init_type = type;
  return TRUE;
}


/*****************************************************************************/
int msgq_handleACKMsg( unsigned int face, unsigned int acked_type,
  unsigned int acked_seq_id )
{
  if( face > 6 )
    return FALSE;

  if( acked_type == msgq[face].init_type )
  {
    if( acked_seq_id == msgq[face].seq_id )
    {
      msgq[face].cnt = 0;
      return TRUE;
    }
  }
  return FALSE;
}


/*****************************************************************************/
void msgq_purgeQueue(unsigned int face ) {
  if( face > 6)
    return;

  msgq[face].cnt = 0;
}
```

```
/******************************************************************************/
void msgq_purgeQueues() {
  int face;

  for( face = 1; face <= 6; face++ )
    msgq[face].cnt = 0;
}


/******************************************************************************/
void msgq_reTx( void ) {
  int face;
  unsigned char ack;
  char *txmsg;
  int msg_valid;
  int oneshot;

  if( paused )
    return;

  for( face = 0; face <= 6; face++ )          // for actual cube faces,
  {
    msg_valid = FALSE;                        // assume no msg to tx.

    if( msgq[face].ack_msg[0] != 0 )          // if (N/A)CK msgs waiting,
    {
      txmsg = msgq[face].ack_msg;               // get a pointer to it
      msg_valid = TRUE;                       // indicate we have a msg to tx
      oneshot = TRUE;                         // we'll only tx it once
    }
    else if( msgq[face].cnt > 0 )              // otherwise, if there is any
    {                                         // other type of msg to tx,
      txmsg = msgq[face].msg;                   // get a pointer to it
      msg_valid = TRUE;                       // indicate we have a msg to tx
      oneshot = FALSE;                        // we'll be tx'ing many times
    }

    if( (face == 0) && (msg_valid) )          // if valid msg for uplink face,
    {
      printf( "%s", txmsg );                    // just send the message
      if( oneshot )                           // if it was a (N/A)CK message,
        msgq[face].ack_msg[0] = 0;            // prevent it from being re-tx'd
      else                                    // for all other types,
        msgq[face].cnt--;                     // dec. repeat count
    }
    else if( (face != 0) && (msg_valid) )     // for all real cube faces, with
    {                                         // valid messages pending,
      pause( peripheralClockFrequency() / 9000 );
                                              // pause before attempting a mux
                                              // switch to allow prev. msg to
                                              // finish

      ack  = txI2CAddress(SLA, WRITE);        // initiate a master tx exchange
      ack &= txI2CData(SWITCH_TX_FACE);       // transmit switch face command
      ack &= txI2CData(face);                 // tx new dest. of mux output
      stopI2C();                              // terminate the exchange

      if( ack )                               // if mux change successful,
      {
        printf( "%s", txmsg );                  // send the message
        pause( peripheralClockFrequency() / 500 );
                                            // again, wait before switch mux
        ack  = txI2CAddress(SLA, WRITE);    // initiate a master tx exchange
        ack &= txI2CData(SWITCH_TX_FACE);   // transmit switch face command
        ack &= txI2CData(0);                // disable all mux outputs
        stopI2C();                          // terminate the exchange

        if( oneshot )                       // if it was a (N/A)CK message,
          msgq[face].ack_msg[0] = 0;        // prevent it from being re-tx'd
        else                                // for all other types,
          msgq[face].cnt--;                 // dec. repeat count

        totalTxCount++;                     // inc. total # msg tx'd
      }
      else                                  // else mux switch unsuccessful,
      {
        ack  = txI2CAddress(SLA, WRITE);    // initiate a master tx exchange
        ack &= txI2CData(SWITCH_TX_FACE);   // transmit switch face command
        ack &= txI2CData(0);                // try to deactivate mux
        stopI2C();                          // terminate the exchange
      }
    }
  }
}
```

B.7 debug.c

The functions contained here implement a debugging interface that can also be used to set some of the system configuration parameters.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include "miche.h"

#define _debug_
#include "debug.h"
#undef _debug

#include "hal/psoc.h"
#include "hal/orientation.h"
#include "main.h"
#include "usrint.h"
#include "util.h"
#include "hal/magswitch.h"

int debugging = FALSE;

void startDebugInterface()
{
  disableUARTISR();
  msgq_pauseTx();
  debugging = TRUE;
  pulsateLED(4000);
  readSettingsFromPSoC();
  displayMainMenu();
  handleMainMenu();

  blankLED();
  debugging = FALSE;
  enableUARTISR();
  msgq_restartTx();
}


static void displayMainMenu()
{
  printf("\e[H");
  printf("\e[2J");
  printf("                              Miche Self-Test System\n");
  printf("\n");
  printf("                                    Main Menu:\n");
  printf("--------------------------------------------------------------------------------\n");
  printf("\n");
  printf("\n");
  printf("\t\t\t C\t Communication\n");
  printf("\n");
  printf("\t\t\t O\t Orientation\n");
  printf("\n");
  printf("\t\t\t M\t Magswitches\n");
  printf("\n");
  printf("\t\t\t L\t debug Log\n");
  printf("\n");
  printf("\t\t\t U\t set UID (     )\n");
  printf("\n");
  printf("\t\t\t W\t Write settings to flash\n");
  printf("\n");
  printf("\t\t\t Q\t Quit\n");
  printf("\e[23;1f");
  printf("--------------------------------------------------------------------------------\n");
  printf(">");

  printf("\e[15;43f");
  printf("%d", settings.uid);

  printf("\e[24;1f");
}


static void handleMainMenu()
{
  int choice;
  unsigned int new_uid;

  printf("\e[15;43f");
  printf("%d", settings.uid);

  printf("\e[24;1f");

  while ( (choice = toupper( getchar() )) != 'Q')
  {
    printf("%c", choice);
    pause(INPUT_PAUSE);
    switch( choice ) {
      case 'C':
        displayCommMenu();
        handleCommMenu();
        break;
      case 'O':
        displayOrientationMenu();
        handleOrientationMenu();
        break;
      case 'M':
        displayMagswitchMenu();
        handleMagswitchMenu();
        break;
      case 'L':
        displayDebugLogMenu();
        handleDebugLogMenu();
        break;
      case 'U':
        printf("\e[15;43f");
        new_uid = getNumber(16, settings.uid);
        if (new_uid != settings.uid) {
          settings.uid = new_uid;
          settings.modified = TRUE;
        }
        break;
      case 'W':
        writeSettingsToPSoC();
        break;
    }
    printf("\e[24;1f\e[2K\e[24;1f>"); // redisplay the prompt
  }

  if (settings.modified)
  {
    printf("\e[24;1f\e[2K\e[24;1fSettings have not been saved, do you want to save them? (y/n)>");
    do {
      printf("\e[24;63f \e[24;63f");
      choice = toupper( getchar() );
      printf("%c", choice);
      pause(INPUT_PAUSE);
    } while ( (choice != 'Y') && (choice != 'N') );

    if (choice == 'Y')
      writeSettingsToPSoC();
  }

  printf("\e[2J"); // Q was pressed: clear the screen
  printf("\e[H");  // and move the cursor home
}


static void displayCommMenu()
{
  printf("\e[H");
  printf("\e[2J");
  printf("                              Miche Self-Test System\n");
  printf("\n");
  printf("                           Communication Subsystem Menu:\n");
  printf("--------------------------------------------------------------------------------\n");
  printf("\n");
  printf("\n");
  printf("\t\t\t T\t Transmit message\n");
  printf("\n");
  printf("\t\t\t F\t query receive Fifos\n");
  printf("\n");
  printf("\t\t\t C\t set Comparator thresholds:\n");
  printf("\n");
  printf("\t\t\t R\t set transmit Repeat count:\n");
  printf("\n");
  printf("\t\t\t B\t Back to main menu\n");
  printf("\e[23;1f");
  printf("--------------------------------------------------------------------------------\n");
  printf(">");
}

static void handleCommMenu()
{
  int choice;
  unsigned char face;
  unsigned char message[MAX_MSG_LENGTH + 1];
  unsigned char new_thold;
  unsigned int new_repeat_count;
  int i;



  printf("\e[11;61f%d", settings.comparatorThreshold);
  printf("\e[13;61f%d", settings.repeatCount);

  printf("\e[24;1f\e[2K\e[24;1f>");  // clear the input line

  while ( (choice = toupper( getchar() )) != 'B')
  {
    printf("%c", choice);
    pause(INPUT_PAUSE);
    if (choice == 'T')
    {
      printf("\e[24;1f\e[2K\e[24;1fSelect face (1-6)> ");
      do {
        printf("\e[24;19f \e[24;19f");          // clear the last input
  face = toupper( getchar() );           // get a new character
  printf("%c", face);                    // print it
  pause(INPUT_PAUSE);                    // pause so that the user can see what was typed
        if (face == 27)                        // if ESC is pressed,
          break;                               // stop asking for a face
      } while ( !( (face >= '1') && (face <= '6') ) );

      if (face != 27)
      {

        face = (face - 48);                    // convert ascii codes into real numbers

        printf("\e[24;1f\e[2K");               // clear the prompt
        printf("\e[24;1fType message>");       // ask user for message content

        i = 0;
        do {
          choice = toupper( getchar() );
          if ((choice == 8) && (i > 0))
          {
            printf("\e[D \e[D");
            i--;
          }
          else
          {
```

```
          message[i] = choice;
          printf("%c",message[i]);
          i++;
        }
      } while ( (i < (MAX_MSG_LENGTH - 1)) && (message[i-1] != 13) );

      if (message[i-1] != 13) {              // if the message has reached maximum length without
                                             // the user pressing enter,
        while (getchar() != 13);             // loop until user presses enter to complete the message
        message[i] = 0;
      }
      else
        message[i-1] = 0;                    // tack a null character onto end of message

      printf("\e[24;1f\e[2K");               // clear the prompt line
      if (!txMsg(message, face))
        printf("\e[24;1fFailure: Message could not be sent!");
      pause(1000000);
    }                                        // if (face != 27)
  }                                          // if (choice == 'T')

  else if (choice == 'F')
  {
    printf("\e[24;1f\e[2K\e[24;1f");
    printf("Select face (1-6)>");
    do {
      printf("\e[24;19f \e[24;19f");         // clear the last input
face = toupper( getchar() );           // get a new character
printf("%c", face);                    // print it
pause(INPUT_PAUSE);                    // pause so that the user can see what was typed
      if (face == 27)                        // if ESC is pressed,
        break;                               // stop asking for a face
    } while ( !( (face >= '1') && (face <= '6') ) );

    if (choice != 27)
    {
      printf("\e[24;1f\e[2K\e[24;1f");
      if (readPSoCRxQueue((face-48), message))
      {
        if (message[0] == 0)
          printf("No new message on face %c.", face);
        else
          printf("%s",message);
        getchar();                           // wait for user to acknowledge by pressing any key
      }
      else
      {
        printf("Failure communicating with PSoC");
        pause(3*INPUT_PAUSE);
      }
    }
  }


  else if (choice == 'C')
  {
    printf("\e[11;61f");
    new_thold = getNumber(8, settings.comparatorThreshold);
                                             // get an 8 bit (3 digit) number from the user
    printf("\e[24;1f\e[2K\e[24;1f");
    if (new_thold != settings.comparatorThreshold)
```

```
      {
        if (setComparatorThreshold(new_thold)) {
          printf("New threshold successfully set.");
          settings.comparatorThreshold = new_thold;
          settings.modified = TRUE;
        }
        else
          printf("PSoC not responding");
        pause(3*INPUT_PAUSE);
      }
    }

    else if( choice == 'R' )
    {
      printf("\e[13;61f");
      new_repeat_count = getNumber(16, settings.repeatCount);
      if( new_repeat_count != settings.repeatCount)
      {
        settings.repeatCount = new_repeat_count;
        settings.modified = TRUE;
      }
    }

    printf("\e[24;1f\e[2K\e[24;1f>");  // clear the input line

  }

  displayMainMenu();  // B was pressed, redisplay the main menu
}

static void displayOrientationMenu()
{
  printf("\e[H");
  printf("\e[2J");
  printf("                              Miche Self-Test System\n");
  printf("\n");
  printf("                            Orientation Subsystem Menu:\n");
  printf("------------------------------------------------------------------------------\n");
  printf("\n");
  printf("\t\t O\t turn accelerometer\n");
  printf("\n");
  printf("\t\t X\t configure X-axis      inversion:\n");
  printf("\t\t\t\t\t\t neutral:        neutral:\n");
  printf("\t\t\t\t\t\t upright: \n");
  printf("\n");
  printf("\t\t Y\t configure Y-axis      inversion:\n");
  printf("\t\t\t\t\t\t neutral:        neutral:\n");
  printf("\t\t\t\t\t\t upright: \n");
  printf("\n");
  printf("\t\t R\t Reset min and max accelerometer values\n");
  printf("\n");
  printf("\t\t B\t Back to main menu\n");
  printf("\n");
  printf("\t\tBottom face: \t\t\t\tTilt:\n");
  printf("\t\tX: \tCurrent:       \t Min:       \t Max:     \n");
  printf("\t\tY: \tCurrent:       \t Min:       \t Max:     \n");
  printf("\e[23;1f");
  printf("------------------------------------------------------------------------------\n");
  printf(">");
}
```

```
static void handleOrientationMenu ()
{
  int choice = ' ';
  unsigned char accelState;
  unsigned char newTiltSwPos, prevTiltSwPos;
  unsigned int newXAccel, prevXAccel, newYAccel, prevYAccel;
  unsigned int xAccelMin = 100000;
  unsigned int xAccelMax = 0;
  unsigned int yAccelMin = 100000;
  unsigned int yAccelMax = 0;
  unsigned int inverted_thold, neutral_low_thold, neutral_high_thold, upright_thold;

  accelState = isAccelerometerOn();
  if ( accelState )
    printf("\e[6;45fOff");
  else
    printf("\e[6;45fOn ");

  printf("\e[8;59f%d",  settings.xAxis.invertedThreshold);
  printf("\e[9;59f%d",  settings.xAxis.neutralLowThreshold);
  printf("\e[9;75f%d",  settings.xAxis.neutralHighThreshold);
  printf("\e[10;59f%d", settings.xAxis.uprightThreshold);

  printf("\e[12;59f%d", settings.yAxis.invertedThreshold);
  printf("\e[13;59f%d", settings.yAxis.neutralLowThreshold);
  printf("\e[13;75f%d", settings.yAxis.neutralHighThreshold);
  printf("\e[14;59f%d", settings.yAxis.uprightThreshold);

  printf("\e[20;30f ");
  printf("\e[20;30f%d", getBottomFace());


  while (choice != 'B')
  {
    if (accelState != isAccelerometerOn())
      if ( isAccelerometerOn() )
      {
        accelState = 1;
        printf("\e[6;45fOff");
      }
      else
      {
        accelState = 0;
        printf("\e[6;45fOn ");
      }

    newTiltSwPos = getTiltSwitchState();        // get the newest state of the tilt switch
    if (newTiltSwPos != prevTiltSwPos)          // if the tilt switch has changed state,
    {
      if (newTiltSwPos == 0)                    // and if it is grounded,
        printf("\e[20;63fOpen  ");              // then display that the tilt switch is open
      else                                      // otherwise,
        printf("\e[20;63fClosed");              // display that the tilt switch is closed
      prevTiltSwPos = newTiltSwPos;             // update for the next iteration
    }

    newXAccel = getAcceleration(XAXIS);         // get the newest x acceleration from the accelerometer
    if (newXAccel != prevXAccel)                // if the new value is different than the previous
    {
      printf("\e[21;34f     ");                 // clear the old value on the scren to prevent artifacts
      printf("\e[21;34f%d", newXAccel);         // display the new x acceleration
```

```
}
prevXAccel = newXAccel;                     // update for the next iteration

newYAccel = getAcceleration(YAXIS);         // get the newest y acceleration from the accelerometer
if (newYAccel != prevYAccel)                // if the new value is different than the previous,
{
  printf("\e[22;34f     ");                 // clear the old value on screen to prevent artifacts
  printf("\e[22;34f%d", newYAccel);         // display the new y acceleration
}
prevYAccel = newYAccel;                     // update for the next iteration

if (newXAccel < xAccelMin)                  // if the new x acceleration is less than previous min,
{
  xAccelMin = newXAccel;                    // update the minimum with the newest value
  printf("\e[21;47f      ");                // clear the old value on screen to prevent artifacts
  printf("\e[21;47f%d", xAccelMin);         // display the new minimum
}

if (newXAccel > xAccelMax)                  // if the new x acceleration is greater than previous max,
{
  xAccelMax = newXAccel;                    // update the maximum with the newest value
  printf("\e[21;63f      ");                // clear the old value on screen to prevent artifacts
  printf("\e[21;63f%d", xAccelMax);         // display the new maximum
}

if (newYAccel < yAccelMin)
{
  yAccelMin = newYAccel;
  printf("\e[22;47f      ");
  printf("\e[22;47f%d", yAccelMin);
}

if (newYAccel > yAccelMax)
{
  yAccelMax = newYAccel;
  printf("\e[22;63f      ");
  printf("\e[22;63f%d", yAccelMax);
}

printf("\e[20;30f ");
printf("\e[20;30f%d", getBottomFace());

if ( U0LSR & 0x01 )                         // if there is a character in the receive FIFO
  {
    choice = toupper( getchar() );          // retrieve it
    printf("\e[24;2f%c", choice);           // print the character
    pause(INPUT_PAUSE);                     // pause so that the user can see what was typed

    if (choice == 'O')                      // if the user typed an 'O',
    {
      if ( isAccelerometerOn() )            // if the accelerometer is already on,
        accelerometerOff();                 // turn it off
      else                                  // otherwise, it must be off
        accelerometerOn();                  // so turn it on
    }
    else if (choice == 'X')                 // otherwise, if the user typed an 'X',
    {
      printf("\e[8;59f");                   // move back to the beginning of the field
      inverted_thold = getNumber(14, inverted_thold);
                                            // get a 14-bit (5 digit) number from the user
      settings.xAxis.invertedThreshold = inverted_thold;
  do {
    printf("\e[9;59f");
    neutral_low_thold = getNumber(14, neutral_low_thold);
    if (neutral_low_thold < inverted_thold)
      printf("\e[24;1fLower neutral threshold must be larger than inversion threshold.");
  } while (neutral_low_thold < inverted_thold);
  settings.xAxis.neutralLowThreshold = neutral_low_thold;


  do {
    printf("\e[9;75f");
    neutral_high_thold = getNumber(14, neutral_high_thold);
    if (neutral_high_thold < neutral_low_thold)
      printf("\e[24;1fUpper neutral threshold must be larger than lower neutral threshold.");
  } while ( neutral_high_thold < neutral_low_thold );
  settings.xAxis.neutralHighThreshold = neutral_high_thold;

  do {
    printf("\e[10;59f");
    upright_thold = getNumber(14, upright_thold);
    if (upright_thold < neutral_high_thold)
      printf("\e[24;1fUpright threshold must be larger than upper neutral threshold.");
  } while (upright_thold < neutral_high_thold);
  settings.xAxis.uprightThreshold = upright_thold;

  settings.modified = TRUE;
}

else if (choice == 'Y')
{
  printf("\e[12;59f");                  // move back to the beginning of the field
  inverted_thold = getNumber(14, inverted_thold);
                                        // get a 14-bit (5 digit) number from the user
  settings.yAxis.invertedThreshold = inverted_thold;

  do {
    printf("\e[13;59f");
    neutral_low_thold = getNumber(14, neutral_low_thold);
    if (neutral_low_thold < inverted_thold)
      printf("\e[24;1fLower neutral threshold must be larger than inversion threshold.");
  } while (neutral_low_thold < inverted_thold);
  settings.yAxis.neutralLowThreshold = neutral_low_thold;


  do {
    printf("\e[13;75f");
    neutral_high_thold = getNumber(14, neutral_high_thold);
    if (neutral_high_thold < neutral_low_thold)
      printf("\e[24;1fUpper neutral threshold must be larger than lower neutral threshold.");
  } while ( neutral_high_thold < neutral_low_thold );
  settings.yAxis.neutralHighThreshold = neutral_high_thold;

  do {
    printf("\e[14;59f");
    upright_thold = getNumber(14, upright_thold);
    if (upright_thold < neutral_high_thold)
      printf("\e[24;1fUpright threshold must be larger than upper neutral threshold.");
  } while (upright_thold < neutral_high_thold);
  settings.yAxis.uprightThreshold = upright_thold;
```

```
        settings.modified = TRUE;

      }

      else if (choice == 'R')
      {
        xAccelMin = 99999;                  // make the min. x acceleration as large as possible
        xAccelMax = 0;                      // make the max. x acceleration as small as possible
        yAccelMin = 99999;                  // make the min. y acceleration as large as possible
        yAccelMax = 0;                      // make the max. y acceleration as small as possible

        printf("\e[21;47f      ");          // clear the previous minimum x acceleration
        printf("\e[21;47f%d", xAccelMin);   // print the new minimum x acceleration
        printf("\e[21;63f      ");          // clear the previous maximum x acceleration
        printf("\e[21;63f%d", xAccelMax);   // print the new maximum x acceleration
        printf("\e[22;47f      ");          // clear the previous minimum y acceleration
        printf("\e[22;47f%d", yAccelMin);   // print the new minimum y acceleration
        printf("\e[22;63f      ");          // clear the previous maximum y acceleration
        printf("\e[22;63f%d", yAccelMax);   // print the new maximum y acceleration
      }
      printf("\e[24;1f\e[2K\e[24;1f>");     // reprint the input prompt
    }
  }
  displayMainMenu();
}


static void displayMagswitchMenu()
{
  printf("\e[H");
  printf("\e[2J");
  printf("                              Miche Self-Test System\n");
  printf("\n");
  printf("                             Magswitch Subsystem Menu:\n");
  printf("--------------------------------------------------------------------------------\n");
  printf("\n");
  printf("\n");
  printf("\t\t 4\t monitor/configure magswitch on face 4\n");
  printf("\n");
  printf("\t\t 5\t monitor/configure magswitch on face 5\n");
  printf("\n");
  printf("\t\t 6\t monitor/configure magswitch on face 6\n");
  printf("\n");
  printf("\t\t B\t Back to main menu\n");
  printf("\e[23;1f");
  printf("--------------------------------------------------------------------------------\n");
  printf(">");
}

static void handleMagswitchMenu()
{
  int choice;

  while ( (choice = toupper( getchar() )) != 'B')
  {
    printf("%c", choice);
    pause(INPUT_PAUSE);
    if ( (choice == '4') || (choice == '5') || (choice == '6') )
    {
      displayMagswitchSubmenu(choice - 48);
      handleMagswitchSubmenu(choice - 48);
    }
    else
      printf("\e[2K\e[24;1f>");  // clear the input line
  }

  displayMainMenu();
}


static void displayMagswitchSubmenu(int face)
{
  printf("\e[H");
  printf("\e[2J");
  printf("                              Miche Self-Test System\n");
  printf("\n");
  printf("                            Magswitch %d Subsystem Menu:\n", face);
  printf("--------------------------------------------------------------------------------\n");
  printf("\n");
  printf("\t\t C\t Connect magswitch\n");
  printf("\n");
  printf("\t\t D\t Disconnect magswitch\n");
  printf("\n");
  printf("\t\t T\t Turn magswitch\n");
  printf("\n");
  printf("\t\t R\t Reset minimum and maximum hall values\n");
  printf("\n");
  printf("\t\t B\t Back to magswitch menu\n");
  printf("\n");
  printf("\n");
  printf("\n");
  printf("\n");
  printf("\n");
  printf("\n");
  printf("\n");
  printf("\t State:    \t Cur value:    \t Min value:    \t Max value:    \n");
  printf("\e[23;1f");
  printf("--------------------------------------------------------------------------------\n");
  printf(">");
}

static void handleMagswitchSubmenu(unsigned char face)
{
  int choice = ' ';
  char motorState;
  char stickState;
  unsigned char hall_cur, hall_prev;
  unsigned char hall_max = 0;
  unsigned char hall_min = 255;
  unsigned int new_thold;
  unsigned char connect_thold, disconnect_thold;

  if ( getMotorState( face ) == MOTOR_OFF )
    printf("\e[10;41foff");
  else
    printf("\e[10;41fon ");

  if ( getMagswitchState( face ) == MSS_ON )
    printf("\e[22;17fConnect");
  else if ( getMagswitchState( face ) == MSS_OFF )
    printf("\e[22;17fDiscon.");
  else
```

```
    printf("\e[22;17fUnknown");

  while (choice != 'B')
  {

    if( getMotorState( face ) == MOTOR_OFF )
      printf("\e[10;41foff");
    else
      printf("\e[10;41fon ");

    if ( getMagswitchState( face ) == MSS_ON )
      printf("\e[22;17fConnect");
    else if ( getMagswitchState( face ) == MSS_OFF )
      printf("\e[22;17fDiscon.");
    else
      printf("\e[22;17fUnknown");

    /*
    Update the minimum and maximum Hall sensor values.  We only bother updating
    any field on the screen if its value has changed.  The saves unnecessary
    instructions and makes the screen look a lot better because it prevents
    so much flickering.
    */

    readHallSensor( face, &hall_cur );

    if (hall_cur != hall_prev)                // only update current reading necessary -- for aesthetics
    {
      printf("\e[22;37f    ");                // clear field to prevent artifacts
      printf("\e[22;37f%d", hall_cur);        // print the new sensor reading
    }
    if (hall_cur > hall_max)                  // check if newest reading greater than old maximum
    {
      hall_max = hall_cur;                    // update the maximum
      printf("\e[22;69f    ");                // clear the field to prevent artifacts
      printf("\e[22;69f%d", hall_max);        // print the new maximum
    }
    if (hall_cur < hall_min)                  // check if the newest reading is less than old minimum
    {
      hall_min = hall_cur;                    // update the minimum
      printf("\e[22;53f    ");                // clear the field to prevent artifacts
      printf("\e[22;53f%d", hall_min);        // print the new minimum
    }

    hall_prev = hall_cur;                     // save the reading so that we have something to
                // compare against on the next iteration

    if ( U0LSR & 0x01 )                       // if there is a character in the receive FIFO
    {
      choice = toupper( getchar() );          // retrieve it
      printf("\e[24;2f%c", choice);           // print the character
      pause(INPUT_PAUSE);                     // pause so that the user can see what was input

      if (choice == 'C')
      {
        if ( !setMagswitchState(face, MSS_ON) )
        {
          printf("\e[24;1fFailure: Magswitch could not connect!");
          pause(1000000);
        }
      }
      else if (choice == 'D')
      {
        if ( !setMagswitchState(face, MSS_OFF) )
        {
          printf("\e[24;1fFailure: Magswitch could not disconnect!");
          pause(1000000);
        }
      }
      else if (choice == 'T')
      {
        if ( getMotorState( face ) == MOTOR_ON )
          setMotorState( face, MOTOR_OFF );
        else
          setMotorState( face, MOTOR_ON);
      }
      else if (choice == 'R')
      {
        hall_min = 255;
        printf("\e[22;53f   ");
        printf("\e[22;53f%d", hall_min);

        hall_max = 0;
        printf("\e[22;69f   ");
        printf("\e[22;69f%d", hall_max);
      }

      printf("\e[24;1f\e[2K\e[24;1f>");       // clear and reprint the prompt
    }
  }
  displayMagswitchMenu();
}


/*****************************************************************************/
void displayDebugLogMenu() {
  printf("\e[H");
  printf("\e[2J");
  printf("                              Miche Self-Test System\n");
  printf("\n");
  printf("                                  Debug Log Menu:\n");
  printf("--------------------------------------------------------------------------------\n");
  printf("\n\n");
  printf("\t\t\t D\t Display log\n");
  printf("\n");
  printf("\t\t\t C\t Clear log\n");
  printf("\n");
  printf("\t\t\t B\t Back to main menu\n");
  printf("\n");
  printf("\e[23;1f");
  printf("--------------------------------------------------------------------------------\n");
  printf(">");
}


/*****************************************************************************/
void handleDebugLogMenu() {
  int choice;

  printf("\e[24;1f");                          // move cursor to the prompt

  while ( (choice = toupper( getchar() )) != 'B')
  {
    printf("%c", choice);
    pause(INPUT_PAUSE);
    switch( choice ) {
      case 'D':
        printf("\e[H");                   // move teh cursor home
        printf("\e[2J");                  // clear the screen
        printDebugLog();                  // print the error log
        while( getchar() != 13 );         // wait for the user to hit enter
        displayDebugLogMenu();            // redisplay the debug log menu
        break;
      case 'C':
        clearDebugLog();                  // clear the error log
        break;
    }
    printf("\e[24;1f\e[2K\e[24;1f>");     // redisplay the prompt
  }

  printf("\e[2J");                        // B was pressed: clear the screen
  printf("\e[H");                         // and move the cursor home

  displayMainMenu();                      // redisplay the main menu
}
```

B.8 usrint.c

This code controls the user LED and is used to parse data from the serial port for the debugging interface.

```
#include <targets/LPC210x.h>
#include <stdio.h>
#include "miche.h"

#define _usrint_
#include "usrint.h"
#undef _usrint_

#include "hal/ioctrl.h"

unsigned char ledMode;
unsigned char countDir, ledOnOff;         // used by ISR to keep track of LED's state


/******************************************************************************/
void initLED()
{
  /*
  initLED() sets up the ARM chip so that user LED can easily be controlled.  To
  control the LED, the following routines use the PWM module to modulate the
  LED's intensity and Timer 1 to either update the PWM duty-cycle or turn the
  LED on and off.
  */

  PWMPR = 0x0;                            // no prescalar on PWMTC
  PWMPCR = 0x400;                         // enable PWM2 output (single edge)
  PWMTCR = 0x2;                           // reset the PWMTC
  PWMTCR = 0x0;                           // bring PWMTC out of reset (but don't start it)
  PWMMCR = 0x2;                           // reset PWMTC on PWMMR0 match
  PWMLER = 0x5;                                  // enable MR2 latch so that changes to MR2 will take effect

  blankLED();                                    // initially turn LED off
}


/******************************************************************************/
void blankLED()
{
  /*
  blankLED() simply turns off the user status LED
  */

  ledMode = LED_OFF;                             // record what mode we are chaning to

  //VICIntEnClr = 0x20;                            // disable T1 interrupts
  PWMTCR = 0x0;                                  // stop the PWMTC
  //T1TCR = 0x0;                                   // stop the T1TC
  setTMR1MR0Period( 0 );                         // disable calls to the updateLED() routine from the T1 ISR

  IOSET = USERLED;                              // turn LED off
  setGPIOMode(USERLED,GPIO);                     // setup USERLED as GPIO so that previous statment
                                                 // has a noticeable effect
}


/******************************************************************************/
void solidLED(unsigned char intensity)
{
  /*
  solidLED(intensity) turns on the user status LED with a given intensity.
  The intensity can range between 0 and 255.  The PWM module is used to control
  LED's intensity.  Timer 1 is not used.
  */

  ledMode = LED_SOLID;                           // record what mode we are chaning to

  setGPIOMode(USERLED, PWM2);                    // setup USERLED as PWM2 so that PWM module
                                                 // will have control over LED

  //VICIntEnClr = 0x20;                            // disable T1 interrupts
  PWMTCR = 0x2;                                  // stop and reset the PWMTC
  //T1TCR = 0x0;                                   // stop the T1TC
  setTMR1MR0Period( 0 );                         // disable calls to the updateLED() routine from the T1 ISR

  PWMMR0 = 0xFF;                                 // setup the PWM period
  PWMMR2 = (255-intensity);                      // setup the PWM duty-cycle

  PWMTCR = 0x1;                                  // enable the PWMTC
}


/******************************************************************************/
void pulsateLED(unsigned int speed)
{
  /*
  pulsateLED(speed) pulsates the LED with the provided speed in mHz.
  */

  ledMode = LED_PULSATE;                         // indicate what mode we're changing to
  setGPIOMode(USERLED, PWM2);                  // setup USERLED as PWM2 so that PWM module
                                               // will have control over LED

  //VICIntEnClr = 0x20;                          // disable T1 interrupts
  PWMTCR = 0x2;                                // stop and reset the PWMTC
  //T1TCR = 0x2;                                 // stop and reset the T1TC

  //T1MR0 = speed;                               // setup the T1 period which controls how
                                               // often the PWM duty-cycle is updated
  setTMR1MR0Period( speed );

  PWMMR0 = 276;                                // set the PWM period
  PWMMR2 = 275;                                // set initial PWM duty-cycle very large so
                                               // that the LED begins in an off state

  countDir = LED_COUNT_DOWN;                   // indicate that we start by counting down
                                               // (the period of PWM2 will initially decrease)

  //VICIntEnable = 0x20;                         // enable TMR1 interrupts
  PWMTCR = 0x1;                                // enable the PWMTC
  //T1TCR = 0x01;                                // enable the T1TC
}


/******************************************************************************/
void blinkLED(unsigned int speed, unsigned char intensity)
{
  /*
  blinkLED(speed, intensity) blinks the LED with the provided speed in mHz and
  at the given intensity which ranges from 0-255.
  */

  ledMode = LED_BLINK;                         // record what mode we're chaning to

  setGPIOMode(USERLED,PWM2);                   // setup P0.7 as PWM2

  //VICIntEnClr = 0x20;                          // disable T1 interrupts
  PWMTCR = 0x2;                                // stop and reset the PWMTC
  //T1TCR = 0x2;                                 // stop and reset the T1TC

  //T1MR0 = speed;                               // set flash rate
  setTMR1MR0Period( speed );

  PWMMR0 = 0xFF;                               // setup PWM period
  PWMMR2 = (255-intensity);                    // setup PWM duty-cycle

  ledOnOff = LED_ON;                           // indicate that the LED is on so that ISR knows
                                               // to turn it off

  PWMTCR = 0x1;                                // enable PWMTC
  //VICIntEnable = 0x20;                         // enable TMR1 interrupts
  //T1TCR = 0x01;                                // enable T1TC

}

/******************************************************************************/
void ledUpdate () {
  /*
  ledUpdate() is called on a regular basis by the timer1 ISR.  The period is set
  by a call to setTMR1MR0Period(). Depending on which mode the LED is operating
  in, ledUpdate() makes regular changes to the state of the LED itself and the
  PWM channel that controls it.
  */

  switch (ledMode) {
    case LED_PULSATE:                         // if we're pulsating the LED,
      if ( countDir == LED_COUNT_UP ) {       // and if we're counting up (increasing duty-cycle)
        if (PWMMR2 < (PWMMR0 - 1))            // and if duty-cycle less than (max - 1),
          PWMMR2 += 1;                        // then increase duty-cycle
        else {                                // otherwise, duty-cycle is maximized,
          PWMMR2 -= 1;                        // decrease duty-cycle
          countDir = LED_COUNT_DOWN;          // indicate that we're now decreasing duty-cycle
        }
      } else {                                // otherwise, we were already decreasing duty-cycle
        if (PWMMR2 > 0)                       // if duty cycle is still greater than 0,
          PWMMR2 -= 1;                        // decrease it
        else {                                // otherwise, duty cycle was 0
          PWMMR2 += 1;                        // start increasing duty-cycle
          countDir = LED_COUNT_UP;            // indicate that we're now increasing duty-cycle
        }
      }
      break;

    case LED_BLINK:                           // otherwise, if we're blinking the LED,
      if ( ledOnOff == LED_ON ) {             // and if the LED is already on,
        IOSET = USERLED;                      // set USERLED high to turn off led if its is GPIO
        setGPIOMode(USERLED,GPIO);            // set USERLED as GPIO so previous statement takes effect
        ledOnOff = LED_OFF;                   // remember our status for the next ISR
      } else {                                // otherwise, LED was off already
        setGPIOMode(USERLED,PWM2);            // configure USERLED as PWM2 to turn on LED
        ledOnOff = LED_ON;                    // remember our status for the next ISR
      }
      break;

  }
}


/******************************************************************************/
unsigned int getNumber(unsigned char bits, unsigned int initial)
{
  /*
  getNumber(bits,initial) uses the serial port to ask the user for an unsigned
  integer that is bits long.  It expects that the cursor is already placed at
  the beginning of the number which is to be changed.  The initial parameter
  passed to getNumber is used as the function's return value if the user does
  not enter a new number or if the user cancels the input sequence by pushing
  escape.  Otherwise, getNumber automatically calculates the maximum number of
  digits given the number of bits desired and restricts the user to entering a
  number of this length.  If the number that the user enters is the maximum
  number of digits but exceeds the largest integer that can be stored in the
  given number of bits, the function rounds the user's entry down to
  (2^bits - 1)--the maximum integer that can be  stored in that many bits.
  The function also supports the backspace key in the case that the user makes a
  mistake in entering his number.
  */

  unsigned char key;
  unsigned int result = 0;
  unsigned char max_digits, digits_remaining;
  unsigned char digits[10];                   // input array of all digits entered
  int i,j;                                    // simple counters
```

```
switch (bits)                             // find the maximum number of digits in given bits
{
  case 0:                                       return 0;
  case 1:  case 2:  case 3:           max_digits = 1;  break;
  case 4:  case 5:  case 6:           max_digits = 2;  break;
  case 7:  case 8:  case 9:           max_digits = 3;  break;
  case 10: case 11: case 12: case 13: max_digits = 4;  break;
  case 14: case 15: case 16:          max_digits = 5;  break;
  case 17: case 18: case 19:          max_digits = 6;  break;
  case 20: case 21: case 22: case 23: max_digits = 7;  break;
  case 24: case 25: case 26:          max_digits = 8;  break;
  case 27: case 28: case 29:          max_digits = 9;  break;
  case 30: case 31: case 32: default: max_digits = 10; break;
}

digits_remaining = max_digits;
i = 0;                                    // initially 0 digits entered

while(1)
{
  key = getchar();                        // get keyboard input

  if ( ( (key == 13)||(key == 27) ) &&    // if ESC or RET typed
       (digits_remaining == max_digits) ) // and nothing else has been typed,
    return initial;                       // return whatever was in the field previously

  else if  ( key == 27 )                  // otherwise, if ESC after digits have been entered
  {
    printf("\e[%dD",(max_digits - digits_remaining));
                                          // move cursor back the number of digits typed
    for (i=0; i<(max_digits - digits_remaining); i++)
      printf(" ");                        // clear each of the digits entered
    printf("\e[%dD",(max_digits - digits_remaining));
    printf("%d%", initial);               // reprint the original value of the field
    return initial;                       // return the original value of the field
  }
  else if (key == 13)                     // otherwise, if RET after digits have been entered
  {
    for (;i>0;i--)                        // for each digits entered starting with the first
      result = 10*result + digits[max_digits - digits_remaining - i];
                                          // add it to the previous sum multiplied by 10
    if ( result > ((1<<bits) - 1) )       // if the result is bigger than largest int that can
                                          // fit in given number of bits,
    {
      printf("\e[%dD",max_digits);        // move cursor back the number of digits typed
      for (i=0; i<max_digits ; i++)       // clear each of the digits entered
        printf(" ");
      printf("\e[%dD",max_digits);        // again, move cursor back the number of digits typed
      printf("%d",((1<<bits) - 1));       // print largest number than can be stored in given bits
      return ((1<<bits) - 1);             // return largest number than can be stored in given bits
    }
    else                                  // otherwise, number entred isn't "too big"
      return result;                      // return that number
  }
  else if (key == 8)                      // otherwise, backspace pressed
  {
    printf("\e[D \e[D");                  // move cursor back, overwrite char, move cursor back
    digits_remaining++;                   // increment number of possible digits remaining
    i--;                                  // decrement number of digits entered
  }
```

```
    else if ( (isdigit(key)) && (max_digits == digits_remaining) )
                                            // otherwise, if user has typed his first digit,
    {
      putchar( key );                       // echo the character back to the screen
      digits[i] = (key - 48);               // place it into the input array
      digits_remaining--;                   // decrement number of digits remaining
      i++;                                  // increment number of digits entered
      for (j=0; j<digits_remaining; j++)    // clear whatever number was under the cursor initially
        printf(" ");
      printf("\e[%dD",digits_remaining);
    }
    else if ( isdigit(key) && (digits_remaining > 0) )
                                            // otherwise, user entered non--first digit and the
                                            // user has not already entered max number of digits
    {
      putchar( key );                       // echo the digit back to the display
      digits[i] = (key - 48);               // place digit into the input array
      digits_remaining--;                   // decrement the number of digits remaining
      i++;                                  // increment number of digits entered
    }
  }
}
```

B.9 log.c

This code implements the event logging facility of the system.

```
#include <targets/LPC210x.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include "miche.h"
#include "boolean.h"

#include "hal/rtc.h"

#define _log_
#include "log.h"
#undef _log_

#include "hal/eeprom.h"
#include "hal/psoc.h"


/********************************************************************************/
int logEvent( int e, char* fmt, ... ) {
  va_list ap;
  char* fmt_p;
  char* str_p;
  struct eeRecord entry;
  struct time t;
  unsigned int eeResponse[2];
  unsigned int i, ctime;

  __ARMLIB_disableIRQ();                    // disable interrupts

  entry._id = EE_REC_ID;                    // allows EEPROM routines to identify individual records
  if( getTime( &t ) ) {                       // if the RTC is set correctly,
    compressTime( &t, &ctime );               // compress the relevant time info into a word
    entry.date = ctime;                       // store the compressed time in the log entry
  }
  else                                        // otherwise, RTC time is invalid
    entry.date = 0;                           // so insert a dummy time into the log entry

  entry.e = e;                                // store the error number in the log entry

  for( i = 0; i < MAX_MSG_LENGTH; i++ )       // zero out the message field to ensure that the
    entry.msg[i] = 0;                         // strlen function call to follow operates correctly

  va_start( ap, fmt );                        // initialize ap to point to the first unknown arg
  for( fmt_p = fmt; *fmt_p; fmt_p++ ) {       // for each character in the format string
    if( *fmt_p != '%')                        // if it is not a % character,
      entry.msg[strlen(entry.msg)] = *fmt_p;  // copy it to the message string of the log entry
    else                                      // otherwise,
      switch( *++fmt_p  ) {
        case 'd':                             // if the % is followed by a 'd',
          sprintf( &entry.msg[strlen(entry.msg)], "%d", va_arg( ap, int ) );
                                              // print the int passed in the parameter list to
          break;                              // the message string of the log entry

        case 's':                             // if the % is followed by a 's',
          for( str_p = va_arg( ap, char * ); *str_p; str_p++ )
            entry.msg[strlen(entry.msg)] = *str_p;
          break;

        default:                              // if we don't recognize the format identifier,
          entry.msg[strlen(entry.msg)] = *fmt_p;// just print it
          break;
      }
  }

  ee_write( (unsigned int)(&entry), eeResponse );
                                              // write the record to flash

  __ARMLIB_enableIRQ();                       // re-enable interrupts

  if( (eeResponse[0] == 0) || (eeResponse[0] == 10) )
                                              // if there was no error, or simply a compare error,
    return TRUE;                              // return success
  else                                        // otherwise, if any other error is encountered
    return FALSE;                             // return failure
}


/******************************************************************************/
void printDebugLog() {
  struct eeRecord* entry;
  struct time t;
  unsigned int eeResponse[2];
  int rec_cnt, i;

  ee_count(0, eeResponse);
  rec_cnt = eeResponse[1];

  if( rec_cnt == 0 ) {
    printf( "Log is empty.\n" );
    return;
  }
  printf( "  DATE      TIME    E#                        MESSAGE \n" );
  printf( "--------------------------------------------------------------------------------\n");

  for( i = 0; i < rec_cnt; i++ ) {
    ee_readn(i, eeResponse);

    if( eeResponse[0] == 0 ) {
      entry = (struct eeRecord *)eeResponse[1];
      uncompressTime( &(entry->date), &t );

      printf( "%02d/%02d/%02d ", t.month, t.dom, (t.year - 2000) );
      printf( "%02d:%02d:%02d ", t.hour, t.min, t.sec );
      printf( " %02d  ", entry->e );
      printf( "%s\n", entry->msg );
    }
  }
}


/******************************************************************************/
int clearDebugLog() {
  unsigned int eeResponse[2];

  __ARMLIB_disableIRQ();

  ee_erase(0, eeResponse);

  __ARMLIB_enableIRQ();

  if( eeResponse[1] == 0 )
    return TRUE;
  else
    return FALSE;
}


/******************************************************************************/
void compressTime( struct time* t, unsigned int* ctime ) {
  *ctime = 0;

  *ctime |= ((t->year - 2000)<<26) & 0xFC000000;// year
  *ctime |= (t->month << 22) & 0x03C00000;      // month
  *ctime |= (t->dom << 17) & 0x003E0000;        // day of the month
  *ctime |= (t->hour << 12) & 0x0001F000;       // hours
  *ctime |= (t->min << 6) & 0x00000FC0;         // minutes
  *ctime |= (t->sec) & 0x0000003F;              // seconds
}


/******************************************************************************/
void uncompressTime( unsigned int* ctime, struct time* t ) {
  t->year = ((*ctime & 0xFC000000) >> 26) + 2000;
  t->month = (*ctime & 0x03C00000) >> 22;
  t->dom = (*ctime & 0x003E0000) >> 17;
  t->hour = (*ctime & 0x0001F000) >> 12;
  t->min = (*ctime & 0x00000FC0) >> 6;
  t->sec = (*ctime & 0x0000003F);
}
```

B.10 rtc.c

This code manages the real time clock used to time stamp messages in the event log.

```
#include <targets/LPC210x.h>
#include "miche.h"
#include "boolean.h"

#define _rtc_
#include "rtc.h"
#undef _rtc_


unsigned char timeExpired;
int timeValid;

/*******************************************************************************/
void initRTC()
{
  /*
  initRTC() initializes the real time clock with the correct time base and
  interrupt priority defined by rtcVICVectAddr and rtcVICVectCntl.  Define these
  macros to change the RTC's interrupt priority.
  */

  int i;                                    // used to store integer part of RTC prescalar

  timeValid = FALSE;                        // indicate that the time stored in the RTC is invalid
  CCR = 0x000000020;                        // keep the RTC in reset

  i = (peripheralClockFrequency() / 32768) - 1; // calculate the integer part of the RTC prescalar
  PREINT = i;                               // load integer part of the RTC prescalar
  PREFRAC = peripheralClockFrequency() - (32768 * (i + 1));
                                            // calculate and load the fractional part of prescalar
  ILR = 0x03;                               // clear pending interrupts

  VICIntSelect &= ~0x2000;                  // assign RTC IRQ status (as opposed to FIRQ)
  rtcVICVectAddr = (unsigned int)rtcISR;    // assign RTC ISR interrupt 5th priority
  rtcVICVectCntl = 0x2d;                    // assign RTC int. to 5th priority and enable it
  VICIntEnable = 0x2000;                    // enable RTC interrupts

  CIIR = 0x00;                              // don't generate IRQ on time counter increments
  AMR = 0xff;                               // don't generate IRQ on alarm matchings

  CCR = 0x000000000;                        // bring RTC out of reset, but keep it disabled

  SEC = 0;                                  // initialize RTC with basic defaults
  MIN = 0;
  HOUR = 0;
  DOM = 1;
  DOW = 0;
  DOY = 1;
  MONTH = 1;
  YEAR = 0;

  CCR = 0x00000001;                         // enable the RTC

  timeExpired = TRUE;                       // indicate that there is no timer running
}


/*******************************************************************************/
```

```
int setTime(struct time* now) {

  if( !validateTime( now ) )                    // if the time is not valid
    return FALSE;                               // return failure

  if( timeExpired == TRUE ) {                   // otherwise, if the timer isn't in use
    CCR = 0x00000000;                           // pause the RTC
    SEC = now->sec;                             // setup all the time registers
    MIN = now->min;
    HOUR = now->hour;
    DOM = now->dom;
    DOW = now->dow;
    DOY = now->doy;
    MONTH = now->month;
    YEAR = now->year;
    timeValid = TRUE;                           // indicate that the RTC contains a valid time
    CCR = 0x00000001;                           // re-enable the RTC
    return TRUE;
  }

  return FALSE;
}


/******************************************************************************/
int getTime( struct time* t ) {
  if( timeValid == TRUE ) {
    t->year = YEAR;
    t->month = MONTH;
    t->doy = DOY;
    t->dow = DOW;
    t->dom = DOM;
    t->hour = HOUR;
    t->min = MIN;
    t->sec = SEC;
    return TRUE;
  }

  return FALSE;
}


/******************************************************************************/
void measureSeconds(unsigned int seconds)
{
  /*
  measureSeconds(seconds) uses the RTC to measure the specified number of
  seconds.  Initially, it sets timeExpired to false.  When time is expired,
  calls to isTimeExpired return true.  The timer can be cancelled by calling
  cancelRTC();
  */

  if (seconds <= 59)                            // if we can measure in seconds only
  {
    ALSEC = (SEC + seconds) % 60;               // set the alarm
    AMR = 0xfe;                                 // unmask the seconds comparison of the alarm
    timeExpired = FALSE;                        // indicate that time has not expired
  }
  else if (seconds <= 3599) {                   // only bother timing up to 3599 seconds
    ALMIN = (MIN + (unsigned int)(seconds / 60)) % 60;
                                                // setup the number of minutes to be timed
    ALSEC = (SEC + (seconds - 60*(unsigned int)(seconds / 60))) % 60;
                                            // setup the number of seconds to be timed
    AMR = 0xfc;                             // unmask the min. and sec. fields of the alarm register
    timeExpired = FALSE;                    // indicate that time has not expired
  }

  //CCR = 0x000000001;                        // start the RTC
  //now, it should never have stopped
}

/******************************************************************************/
void cancelRTC()
{
  /*
  Prevents the RTC from generating an interrupt and sets timeExpired to true.
  */

  AMR = 0xff;                               // clear all alarms
  //CCR = 0x00000000;                         // stop the tick counter
  timeExpired = TRUE;                       // indicate that time has (artifically) expired
}


/******************************************************************************/
unsigned char isTimeExpired()
{
  /*
  isTimeExpired() returns true if the timer setup by a call to measureSeconds()
  has expired.  Otherwise it returns false.
  */

  return timeExpired;
}


/******************************************************************************/
int validateTime( struct time* t ) {
  int invalidTime = FALSE;

  if( t->sec > 59 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Seconds invalid");
    #endif

    invalidTime = TRUE;
  }

  if( t->min > 59 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Minutes invalid");
    #endif

    invalidTime = TRUE;
  }

  if( t->hour > 23 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Hours invalid");
    #endif

    invalidTime = TRUE;
  }

  if( t->dom > 31 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Day of month invalid");
    #endif

    invalidTime = TRUE;
  }

  if( t->dow > 6 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Day of week invalid");
    #endif

    invalidTime = TRUE;
  }

  if( t->doy > 366 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Day of year invalid");
    #endif

    invalidTime = TRUE;
  }

  if( t->month > 12 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Month invalid");
    #endif

    invalidTime = TRUE;
  }

  if( t->year > 4095 ) {
    #ifdef _logerrors_
      logError(INVALID_TIME, "Year invalid");
    #endif

    invalidTime = TRUE;
  }


  if( invalidTime )
    return FALSE;
  else
    return TRUE;
}


/******************************************************************************/
void rtcISR(void)
{
  /*
  rtcISR() is the interrupt service routine for the RTC.  It is called when
  the unmasked alarm registers match the current time.  The routine prevents
  further RTC interrupts (until another timer is started) and it sets
  timeExpired to true.
  */

  AMR = 0xff;                                    // disable further interrupts
  timeExpired = TRUE;                        // indicate that the required amount of time has passed
  //CCR = 0x00000000;                          // disable the RTC
  ILR = 0x00000002;                          // clear the interrupt
  VICVectAddr = 0;                           // reset the VIC
}
```

B.11 eeprom.c

This code interfaces the debugging log to the ARM's flash memory.

```
#include <targets/LPC210x.h>
#include "miche.h"

#define _eeprom_
#include "eeprom.h"
#undef _eeprom_

#include "main.h"
#include "clocks.h"

IAP iap_entry;

/************************************************************************/
/*                                                                      */
/* function:                */
/*  void ee_erase(unsigned int command_ee,unsigned int result_ee[]) */
/*                                                                      */
/* type: void                                                           */
/*                                                                      */
/* parameters:                                                          */
/*  command_ee    - Not used.                          */
/*  result_ee[0] - Returns a response to the last IAP command used. */
/*                 0 - EEPROM successfully erased.      */
/*                 For all other response values, see microcontroller   */
/*                 User Manual, IAP Commands and Status Codes Summary.  */
/*  result_ee[1] - Not used.                                 */
/*                                                                      */
/* constants defined in LPC2k_ee.h used in this function:              */
/*  EE_SEC_L   - microcontroller's Flash sector where EEPROM begins */
/*  EE_SEC_H   - microcontroller's Flash sector where EEPROM ends */
/*                                                                      */
/* description:               */
/*  This function erases LPC2000 on-chip Flash sectors selected to act  */
/*  as an EEPROM. All Flash sectors between EE_SEC_L abd EE_SEC_H */
/*  (including these sectors) will be erased using the In Application */
/*  Programming (IAP) routines (see User Manual for more details).  */
/*                                                                      */
/************************************************************************/
void ee_erase(unsigned int command_ee,unsigned int result_ee[]){
  unsigned int i;
  unsigned int command_iap[5];
  unsigned int result_iap[3];

  command_iap[0]=50;                  //prepare sectors from EE_SEC_L to EE_SEC_H for erase
  command_iap[1]=EE_SEC_L;
  command_iap[2]=EE_SEC_H;
  iap_entry=(IAP) IAP_LOCATION;
  iap_entry(command_iap,result_iap);
  command_iap[0]=52;                    //erase sectors from EE_SEC_L to EE_SEC_H
  command_iap[1]=EE_SEC_L;
  command_iap[2]=EE_SEC_H;
  command_iap[3]=processorClockFrequency() / 1000;
  iap_entry=(IAP) IAP_LOCATION;
  iap_entry(command_iap,result_iap);

  command_iap[0]=53;                    //blankcheck sectors from EE_SEC_L to EE_SEC_H
  command_iap[1]=EE_SEC_L;
  command_iap[2]=EE_SEC_H;
  iap_entry=(IAP) IAP_LOCATION;
  iap_entry(command_iap,result_iap);

  result_ee[0]=result_iap[0];
  return;
}

/************************************************************************/
/*                                                                      */
/* function:              */
/*  void ee_write(unsigned int command_ee,unsigned int result_ee[])     */
/*                                                                      */
/* type: void                                                           */
/*                                                                      */
/* parameters:              */
/*  command_ee   - An address of a content of ee_data type that has */
/*                 to be programmed into EEPROM.                        */
/*  result_ee[0] - Returns a response to the last IAP command used. */
/*                 0 - data successfully programmed in EEPROM.    */
/*                 501 - no space in EEPROM to program data.            */
/*                 For all other response values, see microcontroller   */
/*                 User Manual, IAP Commands and Status Codes Summary.  */
/*  result_ee[1] - Not used.                                     */
/*                                                                      */
/* constants defined in LPC2k_ee.h used in this function:               */
/*  EE_BUFFER_SIZE     - IAP buffer size; must be 256 or 512  */
/*  NO_SPACE_IN_EEPROM     - EEPROM is full                             */
/*  EE_BUFFER_MASK     - parameter used for interfacing with IAP  */
/*  EE_REC_SIZE        - ee_data structure size in bytes          */
/*  EE_SEC_L       - micro's Flash sector where EEPROM begins */
/*  EE_SEC_H       - micro's Flash sector where EEPROM ends */
/*                                                                      */
/* description:             */
/*  This function writes a single structure of ee_data type into the  */
/*  EEPROM using an In Application  Programming (IAP) routines (see */
/*  User Manual for more details). command_ee contains an address of  */
/*  this structure. EEPROM is scanned for the last (if any) record  */
/*  identifier (EE_REC_ID), and a new record is added next to it.       */
/*                                                                      */
/************************************************************************/
void ee_write(unsigned int command_ee,unsigned int result_ee[]){
  int location;
  unsigned int *source, *destination, i;
  unsigned char ee_buffer[EE_BUFFER_SIZE];
  unsigned int command_iap[5], result_iap[3];

  location = ee_locate();
  if (location == -1){
    result_ee[0]=NO_SPACE_IN_EEPROM;
  }
  else{
    for (i=0;i<EE_BUFFER_SIZE;i++) ee_buffer[i]=0xFF;

    destination = (unsigned int *) ((&ee_buffer[0])+((unsigned int)location & EE_BUFFER_MASK));
    source = (unsigned int *) command_ee;
    for(i=EE_REC_SIZE/4;i>0;i--)
                  *(destination++) = *(source++);

    command_iap[0]=50;              //prepare sectors from EE_SEC_L to EE_SEC_H for erase
    command_iap[1]=EE_SEC_L;
    command_iap[2]=EE_SEC_H;
    iap_entry=(IAP) IAP_LOCATION;
    iap_entry(command_iap,result_iap);

    command_iap[0]=51;              //copy RAM to flash/eeprom
    command_iap[1]=(unsigned int) (location & EE_START_MASK);
    command_iap[2]=(unsigned int) (&ee_buffer[0]);
    command_iap[3]=EE_BUFFER_SIZE;
    command_iap[4]=processorClockFrequency() / 1000;
                                              // cclk in kHz
    iap_entry=(IAP) IAP_LOCATION;
    iap_entry(command_iap,result_iap);

    command_iap[0]=56;              //compare RAM and flash/eeprom
    command_iap[1]=(unsigned int) source;
    command_iap[2]=(unsigned int) destination;
    command_iap[3]=EE_REC_SIZE;
    iap_entry=(IAP) IAP_LOCATION;
    iap_entry(command_iap,result_iap);

    result_ee[0]=result_iap[0];
  }
  return;
}

/************************************************************************/
/*                                                                      */
/* function:                */
/*  void ee_read(unsigned int command_ee,unsigned int result_ee[])  */
/*                                                                      */
/* type: void                                                           */
/*                                                                      */
/* parameters:                */
/*  command_ee   - Not used.                                            */
/*  result_ee[0] - Returns a response.          */
/*                 0 - data successfully found in EEPROM.    */
/*                 500 - no data/records available in EEPROM.    */
/*  result_ee[1] - an address of the last record of ee_data type  */
/*                 in EEPROM.                                           */
/*                                                                      */
/* constants used in this function:                                     */
/*  NO_RECORDS_AVAILABLE - EEPROM is empty/no records identifiable      */
/*          with a record identifier (EE_REC_ID) found */
/*  EE_ADR_L     - flash address from where EEPROM begins */
/*  EE_REC_SIZE    - size (in bytes) of a ee_data structure       */
/*                                                                      */
/* description:               */
/*  This function scans an EEPROM content looking for the last record   */
/*  that can be identified with a record identifier (EE_REC_ID). When   */
/*  such data is found, its address is passed as result_ee[1].    */
/*                                                                      */
```

```
/*************************************************************************/
void ee_read(unsigned int command_ee,unsigned int result_ee[]){
  int location;

  location = ee_locate();
  if (location == EE_ADDR_L){
    result_ee[0]=NO_RECORDS_AVAILABLE;
  }
  else{
    result_ee[0]=0;
    result_ee[1]=(unsigned int)(location - EE_REC_SIZE);
  }
  return;
}

/*************************************************************************/
/*                                                                       */
/* function:                  */
/*  void ee_readn(unsigned int command_ee,unsigned int result_ee[])     */
/*                                                                       */
/* type: void                                                            */
/*                                                                       */
/* parameters:                                                           */
/*  command_ee   - An index of a record in EEPROM that should be read.  */
/*  result_ee[0] - Returns a response.          */
/*                 0 - data successfully found in EEPROM.    */
/*                 502 - requested index is out of EEPROM's memory   */
/*  result_ee[1] - an address of the specified record                   */
/*                                                                       */
/* constants used in this function:                                      */
/*  INDEX_OUT_OF_RANGE - index of a record is out of EEPROM's range */
/*  EE_ADR_L           - micro's Flash address from where EEPROM begins */
/*  EE_ADR_H           - micro's Flash address where EEPROM ends  */
/*  EE_REC_SIZE        - size (in bytes) of a ee_data structure         */
/*                                                                       */
/* description:               */
/*  This function returns in result_ee[1] an address of an EEPROM   */
/*  record index specified in command_ee. Index can not be less than 0. */
/*                                                                       */
/*************************************************************************/
void ee_readn(unsigned int command_ee,unsigned int result_ee[]){
  if(command_ee>((EE_ADDR_H+1-EE_ADDR_L)/EE_REC_SIZE)){
    result_ee[0]=INDEX_OUT_OF_RANGE;}
  else{
    result_ee[0]=0;
    result_ee[1]=(unsigned int)(EE_ADDR_L+EE_REC_SIZE*command_ee);
  }
  return;
}

/*************************************************************************/
/*                                                                       */
/* function:                */
/*  void ee_count(unsigned int command_ee,unsigned int result_ee[])     */
/*                                                                       */
/* type: void                                                            */
/*                                                                       */
/* parameters:                */
/*  command_ee   - Not used.           */
/*  result_ee[0] - Returns a response. Always 0.      */
/*  result_ee[1] - number of records of ee_data type in EEPROM.   */
/*                                                                      */
/* constants defined in LPC2k_ee.h used in this function:              */
/*  EE_ADR_L  - micro's Flash address from where EEPROM begins  */
/*  EE_REC_SIZE - size (in bytes) of a ee_data structure         */
/*                                                                      */
/* description:              */
/*  This function returns number of records of ee_data type in EEPROM.  */
/*                                                                      */
/************************************************************************/
void ee_count(unsigned int command_ee,unsigned int result_ee[]){
  result_ee[0]=0;
  result_ee[1]=(unsigned int)((ee_locate()-EE_ADDR_L)/EE_REC_SIZE);
  return;
}

/************************************************************************/
/*                                                                      */
/* function:                */
/*  void ee_locate()              */
/*                                                                      */
/* type: int                                                            */
/*                                                                      */
/* parameters: none             */
/*                                                                      */
/* constants used in this function:                                     */
/*  EE_ADR_L    - micro's Flash address from where EEPROM begins  */
/*  EE_ADR_H  - micro's Flash address where EEPROM ends   */
/*  EE_REC_ID   - a record indicator used to identify valid data  */
/*  EE_REC_SIZE - size (in bytes) of a ee_data structure         */
/*                                                                      */
/* description:              */
/*  This function returns an address as of which new record can be  */
/*   added into Flash/EEPROM. The function is called only if it is      */
/*   known in advance that at least one record can be added. Searching  */
/*   is based on divide by two method that provides the fastest   */
/*   processing time.            */
/*                                                                      */
/************************************************************************/
static int ee_locate() {
  unsigned int addr_l, addr_m, addr_r, size;
  addr_l = EE_ADDR_L;
  if ((*((unsigned char *)addr_l))==0xFF) return(addr_l);
  addr_r = EE_ADDR_H+1;
  if ((*((unsigned char *)(addr_r-EE_REC_SIZE)))==EE_REC_ID) return(-1);
  size = addr_r-addr_l;
  while(size != EE_REC_SIZE){
    addr_m = (addr_r+addr_l)/2;
    if ((*((unsigned char *)addr_m))==0xFF)
      addr_r = addr_m;
    else
      addr_l = addr_m;
    size = size/2;
  }
  return(addr_r);
}
```

B.12 psoc.c

This code implements the high–level I^2C routines used to communicate with the PSoC.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include "miche.h"

#define _psoc_
#include "psoc.h"
#undef _psoc_

#include "i2c.h"
#include "main.h"
#include "util.h"


/*******************************************************************************/
void resetPSoC()
{
  /*
  resetPSoC() resets the PSoC chip.
  */

  IOSET = PSOC_RESET;                          // activate reset
  pause(500);                                  // wait
  IOCLR = PSOC_RESET;                          // deactivate
}


/*******************************************************************************/
unsigned char getRxBufferStatus( unsigned char *rx_status )
{
  unsigned char ack;

  __ARMLIB_disableIRQ();                       // disable interrupts

  ack  = txI2CAddress( SLA, WRITE );           // tell PSoC we want status of RX buffers
  ack &= txI2CData( GET_RX_STATUS );
  stopI2C();

  if( !ack ) {                                 // if PSoC doesn't acknowledge,
    __ARMLIB_enableIRQ();                      // re-enable interrupts
    *rx_status = 0x00;                         // indicate no messages avail
    return FALSE;                              // and return failure
  }

  ack  = txI2CAddress( SLA, READ );            // read the status of the RX buffers
  rxI2CDataNACK( rx_status );

  __ARMLIB_enableIRQ();                        // re-enable interrupts

  if( ack )                                    // return the status of the operation
    return TRUE;
  else {
    rx_status = 0x00;                          // indicate no messages avail
    return FALSE;
  }
}
```

```
/*****************************************************************************/
unsigned char switchTxFace(unsigned char face)
{
  /*
  switchTxFace(face) controls which face re-transmits (over IR) the serial data
  that is emitted by UART0 of the ARM chip.  Face is a number ranging from 1-6.
  The function returns true if the PSoC acknowledged each byte of the I2C
  exchange and false otherwise.
  */

  __ARMLIB_disableIRQ();

  unsigned char ack;

  ack  = txI2CAddress(SLA, WRITE);             // if we can initiate a master transmitter exchange,
  ack &= txI2CData(SWITCH_TX_FACE);            // transmit the switch face command
  ack &= txI2CData(face);                      // transmit the new face on which to re-broadcast
  stopI2C();                                   // terminate the exchange

  __ARMLIB_enableIRQ();

  return ack;                                  // return indication of success or failure
}


/*****************************************************************************/
unsigned char setComparatorThreshold(unsigned char threshold)
{
  /*
  setComparatorThreshold(threshold) update the threshold of six comparators
  on the communication interface board.  Two of these comparators are internal
  to the PSoC and the other four exist on a separate chip and have their
  inverting inputs driven by a DAC on the PSoC.  If the routine successfully
  update the thresholds, it also modified the global configuration settings
  and indicates that they have been changed.  It returns true if the I2C
  sequence was successful and false otherwise.
  */

  unsigned char ack;

  __ARMLIB_disableIRQ();

  ack  = txI2CAddress(SLA, WRITE);             // initiate a master transmitter exchange
  ack &= txI2CData(SET_COMPARATOR_THOLD);      // send the command to set the comparator threshold
  ack &= txI2CData(threshold);                 // send the new comparator value
  stopI2C();                                   // terminate the exchange

  if (ack)                                     // if the previous exchage was successful,
    settings.comparatorThreshold = threshold;  // update the comparator threshold in the global config.

  __ARMLIB_enableIRQ();

  return ack;                                  // return with indication of success or failure
}


/*****************************************************************************/
unsigned char readPSoCRxQueue(unsigned char face, unsigned char* data_ptr)
{
```

```
/*
readPSoCRxQueue(face, *data_ptr) attempts to retrieve the contents of one of
the 6 queues on each of cube's faces.  It places the contents of the queue on
the specified face in a location pointed to the *data_ptr.  Additionally, the
function places a null character at the end of any received string.  If
successful, even if there is nothing new in the queue, the function returns
true.  If the PSoC does not respond properly, the function returns false and
also places a null character at the address passed in data_ptr to indicate
a 0 length string has been received.
*/

unsigned char ack;
unsigned char data_length;                  // number of bytes in the queue
int i;                                      // simple loop counter

__ARMLIB_disableIRQ();

ack  = txI2CAddress(SLA, WRITE);            // initiate a master transmitter exchange
ack &= txI2CData(READ_RX_QUEUE);            // transmit the command to read a receive queue
ack &= txI2CData(face);                     // indicate which face we plan to query
stopI2C();                                  // terminate the I2C exchange

if (!ack) {                                 // if the slave did not acknowledge,
  *data_ptr = 0;                            // null terminate the non-existent string to prevent the
                                            // calling procedure for thinking that a new (duplicate)
                                            // message was received if the communication fails and the
                                            // caller does not bother to check the return value
  __ARMLIB_enableIRQ();
  return FALSE;                             // return failure
}

if ( txI2CAddress(SLA, READ) ) {            // if the slave responds to a master receiver command,
  rxI2CData(&data_length);                  // the first byte received contains the number of
                                            // data bytes to follow
  if (data_length == 0) {                   // if no data bytes are to follow (ie. queue empty)
    rxI2CDataNACK(data_ptr);                // read an extra byte (w/ NACK) to make the PSoC happy
    stopI2C();                              // terminate the I2C exchange
    *data_ptr = 0;                          // null terminate the (non-existent) string of characters
                                            // that was just received
    __ARMLIB_enableIRQ();
    return TRUE;                            // return success even though nothing received
  }

  if( data_length > MAX_MSG_LENGTH )
    data_length = MAX_MSG_LENGTH;

  for (i = 0; i < (data_length - 1); i++)   // otherwise, loop one less times than the number
                                            // of data bytes to be received
    rxI2CData(data_ptr++);                  // on each iteration, receive a byte and store it
                                            // in the location pointed to by data_ptr and also
                                            // increment data_ptr

  rxI2CDataNACK(data_ptr++);                // NACK the last data byte to make PSoC happy
  *data_ptr = 0;                            // null terminate the string of characters just received
  stopI2C();                                // terminate the I2C transaction
  __ARMLIB_enableIRQ();
  return TRUE;                              // return success
} else {
  stopI2C();                                // otherwise, slave didn't respond to read request
  *data_ptr = 0;                            // null terminate the non-existent string to prevent the
                                            // calling procedure for thinking that a new (duplicate)
                                            // message was received if the communication fails and the
```

```
                                        // caller does not bother to check the return value
    __ARMLIB_enableIRQ();
    return FALSE;                       // return failure
  }
}


/*****************************************************************************/
unsigned char readHallSensor (unsigned char face, unsigned char *value_ptr)
{
  /*
  readHallSensor(face, *value_ptr) reads the value of the Hall effect sensor on
  the specified face and stores its result in the location pointed to by the
  value_ptr parameter.  It returns true if successful and false otherwise.
  */

  unsigned char ack;

  __ARMLIB_disableIRQ();

  ack  = txI2CAddress(SLA, WRITE);              // start a write sequence
  ack &= txI2CData(MONITOR_HALL_SENSOR);       // indicate we want to monitor a hall sensor
  ack &= txI2CData(face);                      // indicate which face
  stopI2C();                                   // finish the I2C transaction

  if( !ack )                                    // if the previous transaction wasn't successful
    return FALSE;                              // return failure
  else if ( txI2CAddress(SLA, READ) )          // otherwise, if the slave accepts a read sequence
  {
    rxI2CDataNACK(value_ptr);                  // retrieve the latest value from the hall sensor
    stopI2C();                                 // finish the I2C transaction

    if( *value_ptr == 0 ) {
      __ARMLIB_enableIRQ();
      return FALSE;
    }

    __ARMLIB_enableIRQ();
    return TRUE;                               // return success
  }
  else                                         // the slave didn't accept the read transaction
  {
    stopI2C();                                 // vacate the I2C bus
    __ARMLIB_enableIRQ();
    return FALSE;                              // return failure
  }
}


/*****************************************************************************/
int saveMSSToPSoC( unsigned char state )
{
  unsigned char ack;

  __ARMLIB_disableIRQ();

  ack  = txI2CAddress( SLA, WRITE );
  ack &= txI2CData( SAVE_MSS );
  ack &= txI2CData( state );
  stopI2C();
  __ARMLIB_enableIRQ();

  return ack;
}

/******************************************************************************/
int loadMSSFromPSoC( unsigned char *state )
{
  unsigned char ack;

  __ARMLIB_disableIRQ();

  ack  = txI2CAddress( SLA, WRITE );
  ack &= txI2CData( LOAD_MSS );
  stopI2C();

  if( !ack )
  {
    __ARMLIB_enableIRQ();
    return FALSE;
  }

  else if( txI2CAddress( SLA, READ ) )
  {
    rxI2CDataNACK( state );
    stopI2C();
    __ARMLIB_enableIRQ();
    return TRUE;
  }

  else
  {
    stopI2C();
    __ARMLIB_enableIRQ();
    return FALSE;
  }
}


/******************************************************************************/
unsigned char writeSettingsToPSoC() {
  unsigned char ack;

  __ARMLIB_disableIRQ();

  ack  = txI2CAddress(SLA, WRITE);
  ack &= txI2CData(SAVE_CFG_DATA);

  ack &= txI2CData(settings.comparatorThreshold);

  ack &= txI2CData( (unsigned char)( (settings.repeatCount >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.repeatCount >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.repeatCount >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.repeatCount      ) & 0xFF ) );

  ack &= txI2CData( settings.unused1 );
  ack &= txI2CData( settings.unused2 );

  /*
  ack &= txI2CData( settings.face4.disconnectThreshold );
  ack &= txI2CData( settings.face4.connectThreshold );
  ack &= txI2CData( settings.face5.disconnectThreshold );
  ack &= txI2CData( settings.face5.connectThreshold );
  ack &= txI2CData( settings.face6.disconnectThreshold );
  ack &= txI2CData( settings.face6.connectThreshold );
  */

  ack &= txI2CData( (unsigned char)( (settings.xAxis.invertedThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.invertedThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.invertedThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.invertedThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralLowThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralLowThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralLowThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralLowThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralHighThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralHighThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralHighThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.neutralHighThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.xAxis.uprightThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.uprightThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.uprightThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.xAxis.uprightThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.yAxis.invertedThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.invertedThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.invertedThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.invertedThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralLowThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralLowThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralLowThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralLowThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralHighThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralHighThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralHighThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.neutralHighThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.yAxis.uprightThreshold >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.uprightThreshold >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.uprightThreshold >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.yAxis.uprightThreshold      ) & 0xFF ) );

  ack &= txI2CData( (unsigned char)( (settings.uid >> 24) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.uid >> 16) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.uid >>  8) & 0xFF ) );
  ack &= txI2CData( (unsigned char)( (settings.uid      ) & 0xFF ) );

  stopI2C();

  if (ack)
    settings.modified = FALSE;

  __ARMLIB_enableIRQ();

  return ack;
}
```

```
/******************************************************************************/
unsigned char readSettingsFromPSoC() {
  unsigned char ack;

  __ARMLIB_disableIRQ();

  ack  = txI2CAddress(SLA, WRITE);
  ack &= txI2CData(READ_CFG_DATA);
  stopI2C();

  if (!ack) {
    __ARMLIB_enableIRQ();
    return FALSE;
  }
  else if ( txI2CAddress(SLA, READ) )
  {
    rxI2CData( &settings.comparatorThreshold );

    rxI2CData( (unsigned char *)&settings.repeatCount + 3);
    rxI2CData( (unsigned char *)&settings.repeatCount + 2);
    rxI2CData( (unsigned char *)&settings.repeatCount + 1);
    rxI2CData( (unsigned char *)&settings.repeatCount + 0);

    rxI2CData( &settings.unused1 );
    rxI2CData( &settings.unused2 );

    /*
    rxI2CData( &settings.face4.disconnectThreshold );
    rxI2CData( &settings.face4.connectThreshold );
    rxI2CData( &settings.face5.disconnectThreshold );
    rxI2CData( &settings.face5.connectThreshold );
    rxI2CData( &settings.face6.disconnectThreshold );
    rxI2CData( &settings.face6.connectThreshold );
    */

    rxI2CData( (unsigned char *)&settings.xAxis.invertedThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.xAxis.invertedThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.xAxis.invertedThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.xAxis.invertedThreshold + 0 );

    rxI2CData( (unsigned char *)&settings.xAxis.neutralLowThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.xAxis.neutralLowThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.xAxis.neutralLowThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.xAxis.neutralLowThreshold + 0 );

    rxI2CData( (unsigned char *)&settings.xAxis.neutralHighThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.xAxis.neutralHighThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.xAxis.neutralHighThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.xAxis.neutralHighThreshold + 0 );

    rxI2CData( (unsigned char *)&settings.xAxis.uprightThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.xAxis.uprightThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.xAxis.uprightThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.xAxis.uprightThreshold + 0 );

    rxI2CData( (unsigned char *)&settings.yAxis.invertedThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.yAxis.invertedThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.yAxis.invertedThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.yAxis.invertedThreshold + 0 );
    rxI2CData( (unsigned char *)&settings.yAxis.neutralLowThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.yAxis.neutralLowThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.yAxis.neutralLowThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.yAxis.neutralLowThreshold + 0 );

    rxI2CData( (unsigned char *)&settings.yAxis.neutralHighThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.yAxis.neutralHighThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.yAxis.neutralHighThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.yAxis.neutralHighThreshold + 0 );

    rxI2CData( (unsigned char *)&settings.yAxis.uprightThreshold + 3 );
    rxI2CData( (unsigned char *)&settings.yAxis.uprightThreshold + 2 );
    rxI2CData( (unsigned char *)&settings.yAxis.uprightThreshold + 1 );
    rxI2CData( (unsigned char *)&settings.yAxis.uprightThreshold + 0 );

    rxI2CData( (unsigned char *)&settings.uid + 3);
    rxI2CData( (unsigned char *)&settings.uid + 2);
    rxI2CData( (unsigned char *)&settings.uid + 1);
    rxI2CDataNACK( (unsigned char *)&settings.uid);

    stopI2C();

    __ARMLIB_enableIRQ();
    return TRUE;
  } else {
    stopI2C();
    __ARMLIB_enableIRQ();
    return FALSE;
  }
}
```

B.13 i2c.c

This code handles all of the generic low–level I^{2}C functions.

```
#include <targets/LPC210x.h>
#include "miche.h"
#include "boolean.h"

#define _i2c_
#include "i2c.h"
#undef _i2c_

#include "ioctrl.h"

/******************************************************************************/
void initI2C(void)
{
  /*
  initI2C() configure the I2C controller using the frequency defined by
  I2C_FREQUENCY.  It must be called before any routine that uses the I2C
  interface.
  */

  I2CONCLR = 0xFF;                               // reset the I2C controller

  setGPIOMode(I2CSCL,I2C);                       // put I2CSCL pin in I2C mode
  setGPIOMode(I2CSDA,I2C);                       // put I2CSDA pin in I2C mode
  I2SCLL = peripheralClockFrequency() / (2 * I2C_FREQUENCY);
  I2SCLH = peripheralClockFrequency() / (2 * I2C_FREQUENCY);
                                             // configure the I2C bus clock rate
                                             // I2SCLL and I2SCLH are equal so that
                                             // clock has 50% duty cycle

  I2CONSET = I2EN;                           // enable master mode
}


/******************************************************************************/
unsigned char txI2CAddress(unsigned char address, unsigned char rd_wr)
{
  /*
  TxI2CAddress(address, rd_wr) must be called to initiate a I2C exchange.  Only
  the 7 least significant bits in address are considered when addressing a
  slave.  The rd_wr parameter is used to specify whether the master will be
  reading from the slave or writing to it.  If the slave acknowledges the
  master, the function returns true.  Otherwise, it returns false.
  */


  I2CONCLR = 0xff;                           // reset the I2C controller
  I2CONSET = I2EN;                           // enable master mode

  if (rd_wr == READ)                         // if the master is performing a write,
    I2CONSET = AA | STA;                     // allow the master to ack the data that the slave
                                             // returns and send a start condition
  else                                       // otherwise,
    I2CONSET = STA;                          // just send a start condition

  while (I2STAT != StartComplete);           // wait until the start condition has been transmitted
                                             // if code freezes here, it indicates that the ARM cannot
                                             // pull the SDA line low for some reason.

  I2DAT = (address << 1) | rd_wr;            // I2DAT[7:1] contain slave address, bit 0 contains R/~W bit
  I2CONCLR = STAC | SIC;                     // set to not resend the start condition,
                                             // instead we'll send the data in I2DAT
  while ( !(I2CONSET & SI) );                // wait for the I2C interrupt flag to go high

  if ( ((rd_wr == WRITE) && (I2STAT == SLA_W_ACKed)) ||
       ((rd_wr == READ) && (I2STAT == SLA_R_ACKed)) )
                                             // if the slave acknowledged,
    return TRUE;                             // return true
  else                                       // otherwise,
    return FALSE;                            // return false
}


/******************************************************************************/
unsigned char txI2CData(unsigned char data)
{
  /*
  TxI2CData(data) should be called after an I2C master transmitter exchange has
  already been initiated by an appropriate and successful call to TxI2CAddress.
  The data parameter will be sent to the slave.  The function return true if the
  slave acknolwedges the data and false otherwise.
  */


  I2DAT = data;                              // load the data to be transmitted
  I2CONCLR = SIC;                               // clear the interrupt flag to send data

  while ( !(I2CONSET & SI) );                   // wait for the I2C interrupt flag to go high


  if (I2STAT == TxData_ACKed)                   // if data was ACKed by the slave
    return TRUE;                                // return success
  else                                          // otherwise,
    return FALSE;                               // return failure
}


/******************************************************************************/
void rxI2CData(unsigned char *data)
{
  /*
  RxI2CData(*data) should be called after an I2C master receiver exchange has
  already been initiated by an appropriate and successful call to TxI2CAddress.
  The location pointed to by the *data pointer will be filled with whatever data
  the slave returns.  After receiving 8 bits of data, the master will
  acknowledge its receipt.  The function does not return any indication of
  success because there is no way to determine whether the master is actually
  reading data from the slave or just reading the default state (high) of the
  SDA line.
  */

  I2CONSET = AA;                                // allow master to acknowledge the slave (PSoC)
  I2CONCLR = SIC;                               // clear the I2C interrupt to unstall bus

  while( !(I2CONSET & SI) );                    // wait for the I2C interrupt flag to go high

  *data = (char)I2DAT;                          // move the recieved data to the supplied variable
}


/******************************************************************************/
void rxI2CDataNACK(unsigned char *data)
{
  /*
  RxI2CDataNACK(*data) should be called after an I2C master receiver exchange
  has already been initiated by an appropriate and successful call to
  TxI2CAddress.  The location pointed to by the *data pointer will be filled
  with whatever data the slave returns.  After receiving 8 bits of data, the
  master will not acknowledge its receipt.  This feature may be useful for
  some slave devices. The function does not return any indication of success
  because there is no way to determine whether the master is actually reading
  data from the slave or just reading the default state (high) of the SDA line.
  */

  I2CONCLR = AAC;                               // prevent master from acknowledging the slave (PSoc)
                                                // used to indicate to PSoC firmware that master
                                                // is in the process of requesting the last byte of
                                                // a transfer
  I2CONCLR = SIC;                               // clear the I2C interrupt to unstall bus

  while( !(I2CONSET & SI) );                    // wait for the I2C interrupt flag to go high

  *data = (char)I2DAT;                          // move the recieved data to the supplied variable
}
```

```
/******************************************************************************/
void stopI2C(void)
{
  /*
  stopI2C() puts a stop condition on the I2C bus.  It releases (brings high) the
  data line while the clock line is already high.  It signals to all other
  devices on the bus that a transaction has completed.  It must be called after
  completing an I2C exchange or else slave devices will hang.
  */

  //__ARMLIB_disableIRQ();                        // if interrupts aren't disabled, this function hangs...wtf
  I2CONCLR = SIC;                               // clear the I2C interrupt
  I2CONSET = STO;                               // send stop condition
  while( I2CONSET & STO );                      // wait for the stop condition to be completed
  //__ARMLIB_enableIRQ();                         // re-enable interrupts
}
```

B.14 magswitch.c

The code contained below is the Magswitch control code.

```
#include <targets/LPC210x.h>
#include "miche.h"
#include "boolean.h"
#include "main.h"
#include "debug.h"

#define _magswitch_
#include "magswitch.h"
#undef _magswitch_

#include "ioctrl.h"
#include "psoc.h"

static int curState[3] = {MSS_UNKNOWN, MSS_UNKNOWN, MSS_UNKNOWN};
static int hallAvg[3] = {0, 0, 0};

/******************************************************************************/
void initMagswitches()
{
  setGPIOMode(MOTOR4,GPIO);                     // configure the motor pins as GPIO
  setGPIOMode(MOTOR5,GPIO);
  setGPIOMode(MOTOR6,GPIO);

  IOCLR  = (MOTOR4 | MOTOR5 | MOTOR6);          // make sure that all motors are off
  IODIR |= (MOTOR4 | MOTOR5 | MOTOR6);          // make the motor pins outputs

  // loadMagswitchStates();

  /*
  setMagswitchState( 4, MSS_OFF );
  setMagswitchState( 5, MSS_OFF );
  setMagswitchState( 6, MSS_OFF );
  */
}


/******************************************************************************/
```

```
int getMotorState( int face )
{
  /*
  getMotorState(face) returns MOTOR_ON if the motor on the specified face is
  activated.  It returns MOTOR_OFF otherwise.  Valid faces are 4-6 which
  correspond to ActiveFace1, ActiveFace2, and ActiveFace3, respectively.
  */

  if ( (face == 4) && (IOSET & MOTOR4) )
    return MOTOR_ON;
  else if ( (face == 5) && (IOSET & MOTOR5) )
    return MOTOR_ON;
  else if ( (face == 6) && (IOSET & MOTOR6) )
    return MOTOR_ON;
  else
    return MOTOR_OFF;
}


/****************************************************************************/
void setMotorState( int face, int state )
{
  if (face == FACE4)
  {
    if( state == MOTOR_ON )
    {
      IOSET = MOTOR4;
      curState[0] = MSS_UNKNOWN;
    }
    else
      IOCLR = MOTOR4;
  }
  else if (face == FACE5)
  {
    if( state == MOTOR_ON )
    {
      IOSET = MOTOR5;
      curState[1] = MSS_UNKNOWN;
    }
    else
      IOCLR = MOTOR5;
  }
  else if (face == FACE6)
  {
    if( state == MOTOR_ON )
    {
      IOSET = MOTOR6;
      curState[2] = MSS_UNKNOWN;
    }
    else
      IOCLR = MOTOR6;
  }
}


/****************************************************************************/
int getMagswitchState( int face )
{
  if( face == 4 )
    return curState[0];
  else if( face == 5)
```

```
    return curState[1];
  else if( face == 6)
    return curState[2];
  else
    return MSS_UNKNOWN;
}


/******************************************************************************/
int setMagswitchState( int face, int newState )
{
  int success;

  if( curState[face-4] == newState )
    return TRUE;

  if( newState == MSS_OFF )
  {
    success  = findMSExtreme( face, MSS_MAX );
    success &= findMSExtreme( face, MSS_MIN );
  }
  else if( newState == MSS_ON )
  {
    success  = findMSExtreme( face, MSS_MIN );
    success &= findMSExtreme( face, MSS_MAX );
  }

  // saveMagswitchStates();

  return success;
}


/******************************************************************************/
static int findMSExtreme( int face, int extreme )
{
  unsigned char new_val;
  int prev_avg, new_avg, old_avg;
  int approach = FALSE;
  static int reset = TRUE;
  int reps = 0;

  static int slope[3] =
            {MSS_SLOPE_UNKNOWN, MSS_SLOPE_UNKNOWN, MSS_SLOPE_UNKNOWN};

  if( (face != 4) && (face !=5) && (face != 6) )// if invalid face provided,
    return;                                  // return without doing anything

  face = face - 4;                           // allow easy matrix indexing

  measureSeconds(MS_TIMEOUT);
  setMotorState( face + 4, MOTOR_ON );

  disableUARTISR();
  msgq_pauseTx();

  while( !isTimeExpired() )
  {
    readHallSensor( face + 4,&new_val );

    prev_avg = hallAvg[face];
```

```
new_avg = updateHallAvg( face + 4, new_val );

if( new_avg == prev_avg )
  reps++;
else
  reps = 0;

old_avg = updateHallAvgHistory( face + 4, new_avg );

/* this delay is critical to good disconnections */
pause(6500);

if( (extreme == MSS_MAX) && approach )
{
  if( (slope[face] == MSS_SLOPE_POS) && (new_avg > CON_THOLD) &&
      ( (new_avg < old_avg) || (reps >= 4 ) ) )
  {
    slope[face] = MSS_SLOPE_NEG;
    curState[face] = MSS_ON;
    break;
  }
  else if( (slope[face] == MSS_SLOPE_NEG) && (new_avg > old_avg ) )
    slope[face] = MSS_SLOPE_POS;
  else if( (slope[face] == MSS_SLOPE_UNKNOWN) && (new_avg > old_avg) )
    slope[face] = MSS_SLOPE_POS;
  else if( (slope[face] == MSS_SLOPE_UNKNOWN) && (new_avg < old_avg) )
    slope[face] = MSS_SLOPE_NEG;
}

else if( (extreme == MSS_MAX) && (!approach) )
{
  if( new_avg > old_avg )
  {
    slope[face] = MSS_SLOPE_POS;
    approach = TRUE;
  }
  else if( new_avg < old_avg )
    slope[face] = MSS_SLOPE_NEG;
}

else if( (extreme == MSS_MIN) && approach )
{
  if( (slope[face] == MSS_SLOPE_NEG) && (new_avg < DISCON_THOLD) &&
      ( (new_avg > old_avg) || (reps >= 4 ) ) )
  {
    slope[face] = MSS_SLOPE_POS;
    curState[face] = MSS_OFF;
    break;
  }
  else if( (slope[face] == MSS_SLOPE_POS) && (new_avg < old_avg) )
    slope[face] = MSS_SLOPE_NEG;
  else if( (slope[face] == MSS_SLOPE_UNKNOWN) && (new_avg > old_avg) )
    slope[face] = MSS_SLOPE_POS;
  else if( (slope[face] == MSS_SLOPE_UNKNOWN) && (new_avg < old_avg) )
    slope[face] = MSS_SLOPE_NEG;
}

else if( (extreme == MSS_MIN) && (!approach) )
{
  if( new_avg < old_avg )
  {
        slope[face] = MSS_SLOPE_NEG;
        approach = TRUE;
      }
      else if( new_avg > old_avg )
        slope[face] = MSS_SLOPE_POS;
    }
  }

  if( !debugging )
  {
    enableUARTISR();
    msgq_restartTx();
  }

  setMotorState( face + 4, MOTOR_OFF );
  cancelRTC();

  if( ( (curState[face] == MSS_OFF) && (extreme == MSS_MIN) ) ||
      ( (curState[face] == MSS_ON) && (extreme == MSS_MAX) ) )
    return TRUE;
  else
    return FALSE;
}


/******************************************************************************/
static int saveMagswitchStates()
{
  unsigned char state = 0;
  int face;

  for( face = 0; face <= 2; face++ )
  {
    if( curState[face] == MSS_OFF )
      state |= (MSS_OFF << (2*face) );
    else if( curState[face] == MSS_ON )
      state |= (MSS_ON << (2*face) );
    else if( curState[face] == MSS_UNKNOWN )
      state |= (MSS_UNKNOWN << (2*face) );
    else
      state |= (MSS_UNKNOWN << (2*face) );
  }

  return( saveMSSToPSoC( state ) );
}


/******************************************************************************/
static int loadMagswitchStates()
{
  unsigned char state = 0;
  int face;

  if( loadMSSFromPSoC( &state ) )
  {
    for( face = 0; face <= 2; face++ )
    {
      switch( ((state >> (2*face)) & 0x03) )
      {
        case MSS_OFF:     curState[face] = MSS_OFF;     break;
        case MSS_UNKNOWN: curState[face] = MSS_UNKNOWN; break;
        case MSS_ON:      curState[face] = MSS_ON;      break;
        default:          curState[face] = MSS_UNKNOWN; break;
      }
    }
    return TRUE;
  }
  return FALSE;
}


/******************************************************************************/
static int updateHallAvg(int face, int new_val)
{
  static int sum[3] = {0, 0, 0};
  static int n[3] = {0, 0, 0};                 // number of values stored in each history
  static int head[3] = {0, 0, 0};              // ptr to newest memeber of each history
  static int tail[3] = {0, 0, 0};              // ptr to oldest memeber of each history
  static int hist[3][HIST_LENGTH];
  static int empty = TRUE;

  if( (face != 4) && (face !=5) && (face != 6) )// if invalid face provided,
    return;                                    // return without doing anything

  face = face - 4;                             // allow easy matrix indexing

  if( n[face] < HIST_LENGTH )                  // if history buffer is not full,
  {
    n[face]++;                                 // increment its length
    sum[face] += new_val;                      // add newest value to sum
  }
  else                                         // otherwise, if history buffer is full,
  {
    sum[face] -= hist[face][tail[face]];       // subract oldest value from sum
    sum[face] += new_val;                      // add newest value to sum
  }

  head[face] = (head[face] + 1) % HIST_LENGTH;  // increment the head pointer
  hist[face][head[face]] = new_val;            // update value stored at head ptr

  if( (head[face] == tail[face]) || empty )    // if head ptr overtook tail ptr,
  {
    tail[face] = (tail[face] + 1) % HIST_LENGTH;// increment the tail pointer
    empty = FALSE;
  }

  if( sum[face] != 0 )                         // if the sum is not 0,
    hallAvg[face] = sum[face] / n[face];       // compute the average
  else                                         // otherwise,
    hallAvg[face] = 0;                         // set avg. to 0

  return hallAvg[face];
}


/******************************************************************************/
static int updateHallAvgHistory( int face, int new_val )
{
  static int n[3] = {0, 0, 0};                 // number of values stored in each history
  static int head[3] = {0, 0, 0};              // ptr to newest memeber of each history
  static int tail[3] = {0, 0, 0};              // ptr to oldest memeber of each history
  static int hist[3][AVG_HIST_LENGTH];
  static int empty = TRUE;

  if( (face != 4) && (face !=5) && (face != 6) )// if invalid face provided,
    return;                                   // return without doing anything

  face = face - 4;                            // allow easy matrix indexing

  if( n[face] < AVG_HIST_LENGTH )                 // if history buffer is not full,
    n[face]++;                                // increment its length

  head[face] = (head[face] + 1) % AVG_HIST_LENGTH;
                                              // increment the head pointer
  hist[face][head[face]] = new_val;           // update value stored at head ptr

  if( (head[face] == tail[face]) || empty )     // if head ptr overtook tail ptr,
  {
    tail[face] = (tail[face] + 1) % AVG_HIST_LENGTH;
                                              // increment the tail pointer
    empty = FALSE;
  }

  return hist[face][tail[face]];
}
```

B.15 orientation.c

This code processes values from the accelerometer to inform the module of its orientation with respect to gravity.

```
#include <targets/LPC210x.h>
#include "miche.h"
#include "boolean.h"

#define _orientation_
#include "orientation.h"
#undef _orientation_

#include "hal/ioctrl.h"
#include "main.h"


static unsigned char xAxisTriggered, yAxisTriggered;
                                              // used by Timer0 ISR to keep track of which falling
                                              // edges from the accelerometer it has seen

static unsigned int xAxisHighPeriod, yAxisHighPeriod;
                                              // hold x and y acceleration values


/*******************************************************************************/
void initAccelerometer()
{
  /*
  X component of acceleration is on Capture0.2 and Y component of acceleration
  is on Capture0.1.
  */
  T0PR = (peripheralClockFrequency() /
          (2 * ACCELEROMETER_FREQ * NOMINAL_COUNT) ) - 1;
                                               // configure prescalar to provide a standard number
                                               // of pulses (NOMINAL COUNT) when experiencing no accel.

  setGPIOMode(ACCELY,CAP0_1);                  // configure ACCELY pin as timer 0, capture channel 1
  setGPIOMode(ACCELX,CAP0_2);                  // configure ACCELX pin as timer 1, capture channel 2


  IODIR |= ACCPD;                             // P0.15 is an output used to turn on/off the accelerometer
  IOCLR = ACCPD;                              // ensure that the accelerometer is off

  T0IR = 0xff;                                 // clear pending TMR0 interrutps

  VICIntSelect &= ~0x10;                       // assign TMR0 IRQ status
  tmr0VICVectAddr = (unsigned int)tmr0ISR;     // configure the TMR0 interrupt vector
  tmr0VICVectCntl = 0x24;                      // assign TMR0 to a priority and enable it
  VICIntEnable = 0x10;                         // enable TMR0 interrupts

  T0TCR = 0x02;                                // reset the counter and the prescalar counter
  T0CCR = 0x28;                                // enable capture event and interrupt on rising edge of chnl 1
  T0TCR = 0x01;                                // enable the timer

  VICVectAddr = 0;

}


/*******************************************************************************/
void initTiltSwitch()
{
  IODIR &= ~UP_DN;                        // tilt switch connected to P0.13--make sure it an input
}


/*******************************************************************************/
unsigned char getTiltSwitchState()
{
  if (IOPIN & UP_DN)
    return 0;
  else
    return 1;
}


/*******************************************************************************/
void accelerometerOn()
{
  IOSET = ACCPD;
}


/*******************************************************************************/
void accelerometerOff()
{
  IOCLR = ACCPD;
}


/*******************************************************************************/
```

```
unsigned char isAccelerometerOn()
{
  if (IOSET & ACCPD)
    return TRUE;
  else
    return FALSE;
}


/******************************************************************************/
unsigned int getAcceleration(unsigned char axis)
{
  if (axis == XAXIS)
    return xAxisHighPeriod;
  else if (axis == YAXIS)
    return yAxisHighPeriod;
  else return 0;
}


/******************************************************************************/
unsigned char getBottomFace()
{
  if ( (xAxisHighPeriod <= settings.xAxis.invertedThreshold) &&
       (yAxisHighPeriod >= settings.yAxis.neutralLowThreshold) &&
       (yAxisHighPeriod <= settings.yAxis.neutralHighThreshold) )
    return 2;

  else if ( (xAxisHighPeriod >= settings.xAxis.uprightThreshold) &&
            (yAxisHighPeriod >= settings.yAxis.neutralLowThreshold) &&
            (yAxisHighPeriod <= settings.yAxis.neutralHighThreshold) )
    return 4;

  else if ( (yAxisHighPeriod <= settings.yAxis.invertedThreshold) &&
            (xAxisHighPeriod >= settings.xAxis.neutralLowThreshold) &&
            (xAxisHighPeriod <= settings.xAxis.neutralHighThreshold) )
    return 5;

  else if ( (yAxisHighPeriod >= settings.yAxis.uprightThreshold) &&
            (xAxisHighPeriod >= settings.xAxis.neutralLowThreshold) &&
            (xAxisHighPeriod <= settings.xAxis.neutralHighThreshold) )
    return 3;

  else if ( (xAxisHighPeriod >= settings.xAxis.neutralLowThreshold) &&
            (xAxisHighPeriod <= settings.xAxis.neutralHighThreshold) &&
            (yAxisHighPeriod >= settings.yAxis.neutralLowThreshold) &&
            (yAxisHighPeriod <= settings.yAxis.neutralHighThreshold) )
    if (getTiltSwitchState())
      return 1;
    else
      return 6;

  else
    return 0;
}


/******************************************************************************/
void tmr0ISR()
{
  if ((T0CCR & 0x08) && (T0IR & 0x20))          // if we're waiting for rising edge on channel 1,
                                         // and capture event on channel 1 occured */
  {
    T0IR &= 0xff;                        // clear all match and capture interrupt flags
    T0TCR |= 0x02;                       // reset the timer counter and prescalar counter
    T0TCR &= ~0x02;
    T0CCR = 0x1b0;                       // enable capture and interrupt on falling edge of both
                                         // channel 1 and 2

    xAxisTriggered = FALSE;              // indicate that neither axis has transitioned from high to low
    yAxisTriggered = FALSE;
  }
  else                                   // otherwise, deteced a falling edge detected on some chnl
  {
    if ((T0CCR & 0x10) && (T0IR & 0x20))      // if falling edge on capture channel 1
    {
      T0IR &= 0x20;                      // clear the capture channel 1 interrupt flag
      yAxisTriggered = TRUE;             // denote that the y-axis signal has fallen
      yAxisHighPeriod = T0CR1;           // store the time at which it fell
      T0CCR &= ~0x38;                    // disable any further capture events on channel 1
    }
    else if ((T0CCR & 0x80) && (T0IR & 0x40))  // otherwise, falling edge on capture channel 2
    {
      T0IR &= 0x40;                      // clear the capture channel 2 interrupt flag
      xAxisTriggered = TRUE;             // denote that the x-axis signal has fallen
      xAxisHighPeriod = T0CR2;           // store the time at which it fell
      T0CCR &= ~0x1c0;                   // disable any further capture events on channel 2
    }

    if (xAxisTriggered && yAxisTriggered)     // if both channels have fallen
    {
      T0CCR |= 0x28;                     // re-enable capture and interrupt on posedge of chnl 1
                                         // to capture the beginning of the next cycle
    }
  }

  VICVectAddr = 0;                       // reset the VIC
}
```

B.16 tmr1.c

This code is the interrupt handler for Timer 1 and, among other tasks, prompts the transmission buffers to resend their data.

```
#include <targets/LPC210x.h>
#include "boolean.h"
#include "miche.h"

#define _tmr1_
#include "tmr1.h"
#undef _tmr1_

#include "hal/psoc.h"
#include "msgs/msgqueue.h"
#include "debug.h"

unsigned int TMR1MR0Period;
unsigned int TMR1MR1Period;


/******************************************************************************/
```

```
void initTMR1() {

  T1PR = 0x0;                                  // no prescalar on T1TC
  T1MCR = 0x9;                                 // enable interrupt on MR0 or MR1 match                              //
  T1CCR = 0x0;                                 // not using T1 to capture signals
  T1EMR = 0x0;                                 // T1 not connected to any GPIO pins
  T1TCR = 0x2;                                 // reset the T1TC
  T1TCR = 0x0;                                 // bring the T1TC out of reset (but don't start it)
  T1IR = 0xFF;                                 // clear any pending T1 interrupts

  VICIntSelect &= ~0x20;                       // assign TMR1 IRQ status
  VICVectAddr1= (unsigned int)tmr1ISR;         // assign TMR1 interrupt 2nd priority
  VICVectCntl1 = 0x25;                         // assign TMR1 interrupt (#5) to 2nd IRQ slot and enable it
  VICIntEnable = 0x20;

  T1TCR = 0x01;                                // enable T1TC

  VICVectAddr = 0x0;

}


/*******************************************************************************/
void setTMR1MR0Period( unsigned int period ) {

  if( period == 0)
    T1MCR &= ~0x01;
  else {
    VICIntEnClr = 0x20;
    T1TCR = 0x00;
    TMR1MR0Period = period;
    T1MR0 = T1TC + TMR1MR0Period;
    T1MCR |= 0x01;
    VICIntEnable = 0x20;
    T1TCR = 0x01;
  }

}


/*******************************************************************************/
void setTMR1MR1Period( unsigned int period ) {

  if( period == 0 )
    T1MCR & ~0x08;
  else {
    VICIntEnClr = 0x20;
    T1TCR = 0x00;
    TMR1MR1Period = period;
    T1MR1 = T1TC + TMR1MR1Period;
    T1MCR |= 0x08;
    VICIntEnable = 0x20;
    T1TCR = 0x01;
  }

}


/*******************************************************************************/
void tmr1ISR() {
  /*
  tmr1ISR() is called by the processor when a timer 1 interrupt occurs.
  Depending on which channel a match occurred, the ISR calls different routines
  for other parts of the software that depend on receiving regular interrupts.
  Instead of ever resetting the timer/counter, the newly triggered match
  register is incremented by the appropriate period.
  */

  if( T1IR & 0x01 ) {                           // if match on channel 0
    VICIntEnClr = 0x20;
    ledUpdate();                                // call the LED subroutine that depends on interrupts
    T1TCR = 0x00;                               // disable the TC temporarily
    T1IR = 0x01;                                // clear the match0 interrupt
    T1MR0 = T1TC + TMR1MR0Period;               // increment the match register since we never reset the TC
    VICIntEnable = 0x20;
  }
  else if( T1IR & 0x02 ) {                      // otherwise, if match on channel 1
    VICIntEnClr = 0x20;
    if( debugging == FALSE )
      msgq_reTx();                              // call the message transmission routine
    T1TCR = 0x00;                               // disable the TC temporarily
    T1IR = 0x02;                                // clear the match1 interrupt
    T1MR1 = T1TC + TMR1MR1Period;               // increment teh match register since we never reset the TC
    VICIntEnable = 0x20;
  }
  else                                          // otherwise,
    T1IR = 0xFF;                                // clear all other possible tmr1 interrupts

  T1TCR = 0x01;

  VICVectAddr = 0x0;                            // reset the ISR to prepare for next interrupt
}
```

B.17 uart.c

This section of code implements low-level UART control functions.

```
#include <targets/LPC210x.h>
#include "miche.h"
#include "boolean.h"

#define _uart_
#include "uart.h"
#undef _uart_

#include "clocks.h"
#include "ioctrl.h"
#include "hal/psoc.h"

int uplinkBufferFull;
char uplinkBuffer[MAX_MSG_LENGTH + 1];
int ulBufP;

/*******************************************************************************/
void initUART(unsigned int baud)
{
  /*
  initUART(baud) configures UART0 with the given baud rate specified in bps.
  It must be called before any of the stdio routines that interact with the user
  such as printf and getchar.
  */

  unsigned int divisor;

  divisor = peripheralClockFrequency() / (16 * baud);
                                             // compute the correct divisor given the baud rate

  U0LCR = 0x83;                              // 8 bit, 1 stop bit, no parity, enable divison latch access
  U0DLL = divisor & 0xFF;                    // setup divisor low byte
  U0DLM = (divisor >> 8) & 0xFF;             // setup divisor high byte
  U0LCR &= ~0x80;                            // disable divisor latch access

  setGPIOMode(TXMAIN,UART0);                 // configure TXMAIN pin of ARM as UART0
  setGPIOMode(RX_DBG,UART0);                 // configure RX_DBG pin of ARM as UART0

  U0FCR = 0x07;                              // enable and clear the transmit and receive FIFOs
                                             // interrupt generated when RX FIFO is half full

  U0IER = 0x01;                              // enable receive data avail and char. receive time-out IRQs
                                             // do not enable Rx line status or THRE interrupts
  VICIntSelect &= ~0x40;                     // assign UART0 IRQ status (as opposed to FIRQ)
  uart0VICVectAddr = (unsigned int)uart0ISR; // associate ISR with a priority level
  uart0VICVectCntl = 0x26;                   // associate UART0 interrupts with a priority level
  VICIntEnable = 0x40;                       // enable UART0 interrupts

  uplinkBufferFull = FALSE;                  // indicate that uplink buffer is empty
  ulBufP = 0;                                // reset the buffer pointer
  uplinkBuffer[0] = 0;                       // invalidate any data in the uplink buffer
}


/******************************************************************************/
void disableUARTISR( void ) {
  VICIntEnClr = 0x40;                      // disable UART0 interrupts
}


/******************************************************************************/
void enableUARTISR( void ) {
  VICIntEnable = 0x40;                       // enable UART0 interrupts
}


/******************************************************************************/
void readUplinkQueue( char* raw_msg ){
  if( uplinkBufferFull ) {                   // if the uplink buffer is full,
    strcpy( raw_msg, uplinkBuffer );         // return a copy of the uplink buffer,
    ulBufP = 0;                              // and reset the buffer pointer
    uplinkBufferFull = 0;                    // indicate that the buffer is no longer full,
  }
  else
    raw_msg[0] = 0;                          // clear the first character of the returned string to
                                             // indicate that a new message was not received in full
}


/******************************************************************************/
void __putchar(int ch)
{
  /*
  __putchar(ch) is called by assembly routines such as printf to place a single
  character on the UART0 TxD line.
  */

  if (ch == '\n') {                           // if transmitting a new line,
    while ((U0LSR & 0x20) == 0);              // loop while the THR still contains valid data
    U0THR = '\r';                             // then send a carriage return
  }

  while ((U0LSR & 0x20) == 0);                // loop while the THR still contains valid data
  U0THR = ch;                                 // and then load THR with next byte to be sent
}


/*******************************************************************************/
unsigned char __getchar(void)
{
  /*
  __getchar() is called by assembly routines to get a character from the serial
  port.
  */

  while ((U0LSR & 0x01) == 0);                // wait until the receive FIFO contains a character
  return U0RBR;                               // return that character
}


/*******************************************************************************/
void uart0ISR( void ) {
  unsigned char newchar;

  if( ( ( U0IIR & 0x0E ) == 0x04 ) || ( ( U0IIR & 0x0E ) == 0x0C ) ) {
                                              // if RDA or CTI
    while( U0LSR & 0x01 ) {                   // while there's a new character avail.
      newchar = U0RBR;                        // grab it from the hardware buffer

      if( !uplinkBufferFull ) {
        if( (ulBufP == 0) && (newchar != '#') ) // if 1st char in msg not '#'
          continue;                           // get next character

        uplinkBuffer[ulBufP] = newchar;       // put the next char. in the message buffer

        if( uplinkBuffer[ulBufP] == 0x0d ) {  // if the new character was a CR
          uplinkBuffer[ulBufP] = 0;           // replace the CR with a null terminator
          uplinkBufferFull = 1;               // indicate that a complete message was received
        }
        ulBufP = (ulBufP + 1) % (MAX_MSG_LENGTH + 1);
                                              // otherwise, increment the uplink buffer ptr
                                              // wrapping around to 0 if we still haven't received
                                              // a message terminator and then get the next char
      }
    }
  }

  VICVectAddr = 0x0;                          // reset the interrupt controller
}
```

B.18 ioctrl.c

This section contains low–level input/output control functions.

```
#include <targets/LPC210x.h>
#include "boolean.h"

#define _ioctrl_
#include "ioctrl.h"
#undef _ioctrl_


/******************************************************************************/
unsigned char setGPIOMode(unsigned int pin, unsigned int mode)
{
  unsigned int mask;

  if ( !checkValidPin(pin) )                // if not passed a valid pin representation,
    return FALSE;                           // return to avoid messing anything up

  if ( (mode >= 0) && (mode <= 2) ) {
    if ( (pin >= 0x00000001) && (pin <= 0x00008000) ) {
      mask = pin * pin;
      mask = mask | (mask << 1);
      PINSEL0 = (PINSEL0 & ~mask) | (mode << (2*lg(pin)));
      return TRUE;
    } else if ( (pin >= 0x00010000) && (pin <= 0x80000000) ) {
      pin = pin >> 16;
      mask = pin * pin;
      mask = mask | (mask << 1);
      PINSEL1 = (PINSEL1 & ~mask) | (mode << (2*lg(pin)));
      return TRUE;
    }
  }
}


/******************************************************************************/
unsigned char getGPIOMode(unsigned int pin)
{
  unsigned int mask;

  if ( (pin >= 0) && (pin <= 15) ) {
    mask = 0x3 << (2*lg(pin));
    return (PINSEL0 & mask) >> (2*lg(pin));
  } else if ( (pin >= 16) && (pin <= 31) ) {
    mask = 0x3 << (2*lg(pin >> 16));
    return (PINSEL1 & mask) >> (2*lg(pin >> 16));
  } else
    return 0x3;
}


/******************************************************************************/
static unsigned char checkValidPin(unsigned int pin)
{
  /*
  checkValidPin(pin) returns true if the provided integer representation
  of a pin is valid.  In other words, it must contain a single 1.  For example,
  pin 13 is represented as 0x00002000 or 00000000000010000000000000b.  If the
  provided parameter is not a valid representation of a pin, the function
  returns false.
  */

  int i = 0;                              // counter: number of bits considered
  int sum = 0;                            // present sum of bits

  do {
    sum += (pin & 0x00000001);            // increment sum by the LSB
    pin = pin >> 1;                           // shift right to update LSB
    i++;                                  // indicate that one more bit has been considered
  } while ( (sum <= 1) && (i < 32) );    // repeat until we find 2 "ones" or all bits examined

  if (sum == 1)                           // if we only encountered 1 "one",
    return TRUE;                          // return true
  else                                    // otherwise,
    return FALSE;                         // return false
}


/******************************************************************************/
static unsigned char lg(unsigned int x)
{
  /*
  lg(x) computes the base 2 logarithm of a number that is already a power of 2.
  It is used to compute a pin number given a mask representation of a pin.  For
  example, given 0x0010 it returns 4.
  */

  unsigned char y = 0;

  while ( !(x & 0x1) ) {                  // loop while the LSB of x isn't a 1
    x = x >> 1;                              // rotate x right so that we can examine the next bit
    y++;                                  // increment the counter

    if (y == 32)                          // catch the case where x = 0
      break;                              // to prevent an infinite loop
  }

  return y;                               // return the result
}
```

B.19 clocks.c

These functions manage the ARM's internal clock.

```
#include <targets/LPC210x.h>

#define _clocks_
#include "clocks.h"
#undef _clocks_

#include "boolean.h"

/******************************************************************************/
int initPLL( unsigned int m, unsigned int p )
{
  if( (m < 1) || (m > 32) )
    return FALSE;

  if( (m * OSCILLATOR_CLOCK_FREQUENCY) > MAXIMUM_CCLK_FREQUENCY )
```

```
    return FALSE;

  if( !( (p == 1) || (p == 2) || (p == 4) || (p == 8) ) )
    return FALSE;

  if( ((2 * p * OSCILLATOR_CLOCK_FREQUENCY) < MINIMUM_CCO_FREQUENCY) ||
    ((2 * p * OSCILLATOR_CLOCK_FREQUENCY) > MAXIMUM_CCO_FREQUENCY) )
    return FALSE;

  m = m - 1;                                    // multiplier in PLLCFG is m-1
  switch( p )                                   // configure PLLCFG
  {
    case 1: PLLCFG = m | 0x00; break;
    case 2: PLLCFG = m | 0x20; break;
    case 4: PLLCFG = m | 0x40; break;
    case 8: PLLCFG = m | 0x60; break;
    default: return FALSE;
  }

  PLLCON = 0x01;                                // enable the PLL
  PLLFEED = 0xAA;                               // transmit feed seq to update
  PLLFEED = 0x55;
  while(( PLLSTAT & 0x400 ) == 0 );             // wait for PLL lock
  PLLCON = 0x03;                                // connect (and enable) PLL
  PLLFEED = 0xAA;                               // transmit feed seq to update
  PLLFEED = 0x55;

  return TRUE;
}


/******************************************************************************/
unsigned int processorClockFrequency(void)
{
  /*
  processorClockFrequency() returns the cclk frequency--the frequency at which
  the ARM core is running--in Hertz.  It does account for PLL settings.
  */

  return OSCILLATOR_CLOCK_FREQUENCY * (PLLCON & 1 ? (PLLCFG & 0xF) + 1 : 1);
}


/******************************************************************************/
unsigned int peripheralClockFrequency(void)
{
  /*
  peripheralClockFrequency() returns the pclk frequency--the frequency at which
  all peripherals are operating--in Hertz.
  */

  unsigned int divider;

  switch (VPBDIV & 3) {                         // find the correct pclk divider
      case 0:
        divider = 4;
        break;
      case 1:
        divider = 1;
        break;
      case 2:
        divider = 2;
        break;
  }

  return processorClockFrequency() / divider;   // pclk = cclk / divider
}
```

B.20 util.c

This code is simply a pause function used for some fixed–length delays.

```
#define _util_
#include "util.h"
#undef _util_

void pause(unsigned int time) {
  unsigned int i;

  for (i = 0; i < time; i++);
}
```

Appendix C

PSoC Source Code

This appendix includes all of the C language source code that executes on the Cypress PSoC microprocessor. Because much of the PSoC's functionality is implemented in reconfigurable hardware blocks, this code does not capture the full extent of the PSoC's functionality.

C.1 main.c

This code initializes the PSoC and then just loops checking for new I^{2}C messages.

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

#include "i2c.h"
#include "tx.h"
#include "rx.h"

void main()
{
    initI2C();
    initRxSys();
    initTxSys();
    E2PROM_Start();

    M8C_EnableGInt;

    while(1)
        handleI2C();
}
```

C.2 i2c.c

This code handles all I^{2}C messages received from the ARM.

```
#include <m8c.h>                    // part specific constants and macros
#include "PSoCAPI.h"                // PSoC API definitions for all User Modules


#define _i2c_
#include "i2c.h"
#undef _i2c_

#include "comparator.h"
#include "rx.h"
#include "tx.h"
#include "hall.h"

BYTE I2CMsg[66];                    // EEPROM page is only 64 bytes, but one of the extra
                                    // bytes is for the I2C command, and the other is slack
                                    // so that the default generated I2C routine doesn't
                                    // return a NACK after receiving the last real byte

BYTE DefaultBuffer[2] = {0, 0};     // outdated -- was the buffer that I2C reads without
                                    // prior destination command defaulted to reading

BYTE RXStatusReg = 0;               // holds most important things that ARM cares about for
                                    // quick access: 3 Hall effect sensor values, status of
                                    // the 6 RX buffers

void initI2C() {
        I2C_Start();                // start the I2C handler
        I2C_EnableSlave();          // enable slave mode only
        I2C_EnableInt();            // enable I2C interrupts so that incoming
                                    // I2C messages can be processed
        I2C_InitWrite(I2CMsg,66);   // configure I2C receiver to write any
                                    // received data to the I2CMsg buffer
        I2C_ClrRdStatus();
}

void handleI2C() {
        BYTE status;

        status = I2C_bReadI2CStatus();

    if (status & I2CHW_WR_COMPLETE) {
                switch (I2CMsg[COMMAND]) {
                case READ_RX_BUFFER:
                        switch (I2CMsg[DATA1])
                        {
                                case FACE1:
                                        if ( RX8_1_bCmdCheck() )
                                                RX8_1_MsgSize = RX8_1_bCmdLength();


                                        else
                                                RX8_1_MsgSize = 0;
                                        I2C_InitRamRead(RX8_1_I2CMsg, RX8_1_MsgSize + 1);




                                break;  // case FACE1

                                case FACE2:
                                        if ( RX8_2_bCmdCheck() )
                                RX8_2_MsgSize = RX8_2_bCmdLength();
                        else
                                RX8_2_MsgSize = 0;
                        I2C_InitRamRead(RX8_2_I2CMsg, RX8_2_MsgSize + 1);
                break;  // case FACE2

                case FACE3:
                        if ( RX8_3_bCmdCheck () )
                                RX8_3_MsgSize = RX8_3_bCmdLength();
                        else
                                RX8_3_MsgSize = 0;
                        I2C_InitRamRead(RX8_3_I2CMsg, RX8_3_MsgSize + 1);
                break;  // case FACE3

                case FACE4:
                        if ( RX8_4_bCmdCheck() )
                                RX8_4_MsgSize = RX8_4_bCmdLength();
                        else
                                RX8_4_MsgSize = 0;
                        I2C_InitRamRead(RX8_4_I2CMsg, RX8_4_MsgSize + 1);
                        break;  // case FACE4

                case FACE5:
                        if ( RX8_5_bCmdCheck() )
                                RX8_5_MsgSize = RX8_5_bCmdLength();
                        else
                                RX8_5_MsgSize = 0;
                        I2C_InitRamRead(RX8_5_I2CMsg, RX8_5_MsgSize + 1);
                break;  // case FACE5

                case FACE6:
                        if ( RX8_6_bCmdCheck() )
                                RX8_6_MsgSize = RX8_6_bCmdLength();

                        I2C_InitRamRead(RX8_6_I2CMsg, RX8_6_MsgSize + 1);
                break;  // case FACE6
        }                               // switch (I2CMsg[DATA1])
break;                  // case READ_RX_BUFFER


case SET_COMPARATORS:
        startAllComparators(I2CMsg[DATA1]);
break;                  // case SET_COMPARATORS

case MONITOR_HALL_SENSOR:
        startADC(I2CMsg[DATA1]);
        I2C_InitRamRead(&ADCResult,1);
break;                  // case MONITOR_HALL_SENSOR

case STOP_ADC:
        stopADC();
        AMUX4_Stop();
break;                  // case STOP_ADC

case SET_TX_CHANNEL:
        switchTXChannel(I2CMsg[DATA1]);
break;                  // case SET_TX_CHANNEL

        case SAVE_CFG_DATA:
                E2PROM_bE2Write( 64, &I2CMsg[DATA1], 64, 25 );
        break;
```

```
                case READ_CFG_DATA:
                        I2C_InitFlashRead((const unsigned char *)0x7FC0, 64);
                break;

                case POWER_DOWN:
                        stopADC();
                        AMUX4_Stop();

                        RX8_1_Stop();
                        RX8_2_Stop();
                        RX8_3_Stop();
                        RX8_4_Stop();
                        RX8_5_Stop();
                        RX8_6_Stop();
                        PWM8_Stop();
                        stopAllComparators();
                        DAC8_Stop();

                DigBuf_1_Stop();
                DigBuf_2_Stop();

                I2C_Stop();     //at this point, a POR is necessary to restore PSoC functionality */
        break;

        case GET_RX_STATUS:
                I2C_InitRamRead(&RXStatusReg, 1);
        break;

        case SAVE_MSS:
                E2PROM_bE2Write( 0, &I2CMsg[DATA1], 64, 25 );
        break;

        case LOAD_MSS:
                I2C_InitFlashRead( (const unsigned char *)0x7F80, 64 );
        break;
    }

    I2C_ClrWrStatus();                      // data received and processed--clear write status
    I2C_InitWrite(I2CMsg,66);       // reset write buffer for new command
}

if (status & I2CHW_RD_COMPLETE) {
    if (I2CMsg[COMMAND] == READ_RX_BUFFER) {

        switch (I2CMsg[DATA1]) {
                case FACE1:
                                RX8_1_MsgSize = 0;
                                RXStatusReg &= ~(0x01);
                                RX8_1_CmdReset();
                break; // case FACE1

                case FACE2:
                        RX8_2_MsgSize = 0;
                        RXStatusReg &= ~(0x02);
                        RX8_2_CmdReset();
                break; // case FACE2

                case FACE3:
                        RX8_3_MsgSize = 0;
                        RXStatusReg &= ~(0x04);
```

```
                                RX8_3_CmdReset();
                        break; // case FACE3

                        case FACE4:
                                RX8_4_MsgSize = 0;
                                RXStatusReg &= ~(0x08);
                                RX8_4_CmdReset();
                        break; // case FACE4

                        case FACE5:
                                RX8_5_MsgSize = 0;
                                RXStatusReg &= ~(0x10);
                                RX8_5_CmdReset();
                        break; // case FACE5

                        case FACE6:
                                RX8_6_MsgSize = 0;
                                RXStatusReg &= ~(0x20);
                                RX8_6_CmdReset();
                        break; // case FACE6

                } // switch (I2CMsg[DATA1])

        } // if (I2CMsg[COMMAND] == READ_RX_BUFFER)

        I2C_ClrRdStatus();
        //I2C_InitRamRead(DefaultBuffer, 2);
        I2C_InitRamRead( &RXStatusReg, 1 );
    } // if (status & I2CHW_RD_COMPLETE)
}
```

C.3 comparator.c

This code is used set the output voltage of the digital–to–analog converter which drives the four external comparators. Additionally, it controls the two comparators internal to the PSoC.

```
#include <m8c.h>                        // part specific constants and macros
#include "PSoCAPI.h"                    // PSoC API definitions for all User Modules

#define _comparator_
#include "comparator.h"
#undef _comparator_

void startAllComparators (char vref)
{
        /*
        The routine ensures that all 6 comparators are operating so that the analog
        voltages produced by the IR photodiodes are converted to valid logic levels.
        The routine can also be called to change the threshold values of the
        comparators even if they are already operating.
        */

        CMPPRG_1_Start(CMPPRG_1_HIGHPOWER);
                                        //Start the internal comparators
        CMPPRG_2_Start(CMPPRG_2_HIGHPOWER);
        DAC8_Start(DAC8_HIGHPOWER);     //Start DAC to provide reference for 4 external comparators
        DAC8_WriteStall(vref);          //Write the specified value to the DAC
        setCompRefVoltage(vref);        //Configure the 2 internal comparators
}


void stopAllComparators(void)
{
        /*
        Nothing special--just turnoff the DAC and the comparators to conserver battery.
        */

        CMPPRG_1_Stop();                //Stop the 2 internal comparators
        CMPPRG_2_Stop();
        DAC8_Stop();                    //Stop the DAC which provides the reference voltage
                                        //for the 4 external comparators
}


void setCompRefVoltage (char vref)
{
        /*
        The internal comparators only have 18 possible threshold values.  As a
        result, we have to take the reference voltage passed to the function which
        ranges from 0 to 255 and estimate it using a fixed set of 18 values.  The
        approximation below assumes that the low refernece of the comparator is VSS,
        and the high reference, which is impossible to change is VCC.  It would be
        possible to change the low reference to AGND, but this would require a
        completely new set of if statements.
        */

        if (vref <= 8)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_021); CMPPRG_2_SetRef(CMPPRG_2_REF0_021); }
        else if (vref <= 13)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_042); CMPPRG_2_SetRef(CMPPRG_2_REF0_042); }
        else if (vref <= 24)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_062); CMPPRG_2_SetRef(CMPPRG_2_REF0_062); }
        else if (vref <= 40)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_125); CMPPRG_2_SetRef(CMPPRG_2_REF0_125); }
        else if (vref <= 56)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_188); CMPPRG_2_SetRef(CMPPRG_2_REF0_188); }
        else if (vref <= 72)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_250); CMPPRG_2_SetRef(CMPPRG_2_REF0_250); }
        else if (vref <= 88)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_312); CMPPRG_2_SetRef(CMPPRG_2_REF0_312); }
        else if (vref <= 104)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_375); CMPPRG_2_SetRef(CMPPRG_2_REF0_375); }
        else if (vref <= 119)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_437); CMPPRG_2_SetRef(CMPPRG_2_REF0_437); }
        else if (vref <= 135)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_500); CMPPRG_2_SetRef(CMPPRG_2_REF0_500); }
        else if (vref <= 151)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_562); CMPPRG_2_SetRef(CMPPRG_2_REF0_562); }
        else if (vref <= 167)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_625); CMPPRG_2_SetRef(CMPPRG_2_REF0_625); }
        else if (vref <= 183)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_688); CMPPRG_2_SetRef(CMPPRG_2_REF0_688); }
        else if (vref <= 199)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_750); CMPPRG_2_SetRef(CMPPRG_2_REF0_750); }
        else if (vref <= 215)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_812); CMPPRG_2_SetRef(CMPPRG_2_REF0_812); }
        else if (vref <= 231)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_875); CMPPRG_2_SetRef(CMPPRG_2_REF0_875); }
        else if (vref <= 247)
                { CMPPRG_1_SetRef(CMPPRG_1_REF0_937); CMPPRG_2_SetRef(CMPPRG_2_REF0_937); }
        else
                { CMPPRG_1_SetRef(CMPPRG_1_REF1_000); CMPPRG_2_SetRef(CMPPRG_2_REF1_000); }
}
```

C.4 tx.c

This code is used to switch the output of the digital multiplexer and redirect the RS-232 data stream to a particular face.

```
#include <m8c.h>                        // part specific constants and macros
#include "PSoCAPI.h"                    // PSoC API definitions for all User Modules

void initTxSys()
{
        DBB01OU = 0x00;
        DBB11OU = 0x00;

        PRT2DR = 0x00;                  // P2 pins default to low when not driven by global bus
        PRT2GS = 0x03;                  // P2[7-2] default to not being driven by global bus

        DigBuf_1_Start();
        DigBuf_2_Start();


}

void switchTXChannel(char channel) {
        /*
        Given a channel (1-6), procedure flips bits in the output
        registers of DBB01 and DBB11--the two basic digital blocks
        configured as buffers.  For each register, bit 1 enables
        buffer 1.  Bits 1:0 select which row output the input 1 is
        redirected to.  <00> redirects to output row 0, etc.

        DBB01 is use for TX1 and TX2 channels.  DBB11 is used for
        TX3-TX6.  Only channel can be transmitting at once.  (This
        may be changed in future revisions of software.)

        Pass a parameter outside of the 1-6 range in order to turn off
        all transmitters.
        */

        switch (channel) {
                case 1:
                        DBB11OU = 0x00;
                        DBB01OU = 0x06;
                        PRT2GS = 0x07;
                        break;
                case 2:
                        DBB11OU = 0x00;
                        DBB01OU = 0x07;
                        PRT2GS = 0x0B;
                        break;
                case 3:
                        DBB01OU = 0x00;
                        DBB11OU = 0x04;
                        PRT2GS = 0x13;
                        break;
                case 4:
                        DBB01OU = 0x00;
                        DBB11OU = 0x05;
                        PRT2GS = 0x23;
                        break;
                case 5:
                        DBB01OU = 0x00;
                        DBB11OU = 0x06;
                        PRT2GS = 0x43;
                        break;
                case 6:
                        DBB01OU = 0x00;
                        DBB11OU = 0x07;
                        PRT2GS = 0x83;
                        break;
                default:
                        DBB01OU = 0x00;
                        DBB11OU = 0x00;
                        PRT2GS = 0x03;
                        break;
        }
}
```

C.5 rx.c

This code initializes the six RS-232 receivers and their associated buffers.

```
#include <m8c.h>                        // part specific constants and macros
#include "PSoCAPI.h"                    // PSoC API definitions for all User Modules

#define _rx_
#include "rx.h"
#undef _rx_

#include "comparator.h"

void initRxSys()
{
        BYTE threshold;

        PWM8_Start();   // PWM for UAR's (RX8_1, etc.)

        RX8_1_Start(RX8_1_PARITY_NONE); // start the receiver (no parity bit)
    RX8_1_EnableInt();                  // necessary for the high-level command handling API
        RX8_1_MsgSize = 0;              // indicate 0 initial message size to prevent the ARM
                                        // from reading junk data
    RX8_1_CmdReset();                   // reset the command buffer

    RX8_2_Start(RX8_2_PARITY_NONE);
    RX8_2_EnableInt();
    RX8_2_MsgSize = 0;
    RX8_2_CmdReset();

    RX8_3_Start(RX8_3_PARITY_NONE);
    RX8_3_EnableInt();
    RX8_3_MsgSize = 0;
    RX8_3_CmdReset();

        RX8_4_Start(RX8_4_PARITY_NONE);
        RX8_4_EnableInt();
    RX8_4_MsgSize = 0;
        RX8_4_CmdReset();

        RX8_5_Start(RX8_5_PARITY_NONE);
        RX8_5_EnableInt();
    RX8_5_MsgSize = 0;
        RX8_5_CmdReset();

        RX8_6_Start(RX8_5_PARITY_NONE);
        RX8_6_EnableInt();
    RX8_6_MsgSize = 0;
        RX8_6_CmdReset();

        E2PROM_E2Read(0, &threshold, 1);
        startAllComparators( threshold );
}
```

C.6 hall.c

This code controls the analog–to–digital converter used to sample the values produced by the Hall Effect sensors.

```
#include <m8c.h>
#include <e2prom.h>
#include "amux4.h"
#include "adc.h"

#define _hall_
#include "hall.h"
#undef _hall_

unsigned char ADCResult;                // the ADC ISR stores data here so that it
                                        // can be accessed by the I2C subsystem

void startADC(char channel)
{
        stopADC();                      // stop the ADC if it was already operating

        AMUX4_Start();                  // analog mux allows channel selection

        if ((channel >= 4) || (channel <= 6)) {
                                        // only allow selection of valid channels
                switchHallChannel(channel);
                                        // select the correct input channnel
                ADC_Start(ADC_HIGHPOWER);
                                        // start the ADC
                ADC_fClearFlag();       // invalidate any previous samples
                ADC_GetSamples(0);      // sample forever
        }
}

void switchHallChannel(char channel)
{
        AMUX4_InputSelect(channel - 3); // from the mux's perspective, valid
```

```
                                        // channels are 0-3 (channel 0 is RX_AN2)
                                        // and 1-3 map to Hall4, Hall5, and Hall6,
                                        // respectively
}

void stopADC (void)
{
        ADC_fClearFlag();               // indicate that any old data is no longer valid
        ADC_Stop();                     // turn off the ADC
}
```

Bibliography

[1] Andres Castano, Alberto Behar, and Peter Will. The conro modules for reconfigurable robots. *IEEE Transactions on Mechatronics*, 7(4):403–409, December 2002.

[2] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[3] National Semiconductor Corporation. *LP3981: Micropower, 300mA Ulta Low-Dropout CMOS Voltage Regulator*, April 2005. Document No. DS200203.

[4] ON Semiconductor Corporation. *NCP561: 150mA CMOS Low Iq Low-Dropout Voltage Regulator*, July 2004. Publication Order No. NCP561/D.

[5] Akiya Kamimura, Haruhisa Kurokawa, Eiichi Yoshida, Satoshi Murata, Kohji Tomita, and Shigeru Kokaji. Automatic locomotion design and experiments for a modular robotic system. *IEEE/ASME Transactions on Mechatronics*, 10(3):314–325, June 2005.

[6] Michihiko Koseki, Kengo Minami, and Norio Inou. Cellular robots forming a mechanical structure (evaluation of structural formation and hardware design of "chobie ii"). In *Proceedings of 7th International Symposium on Distributed Autonomous Robotic Systems (DARS04)*, pages 131–140, June 2004.

[7] Efstathios Mytilinaios, David Marcus, Mark Desnoyer, and Hod Lipson. Designed and evolved blueprints for physical self–replicating machines. In *Ninth International Conference on Artificial Life (ALIFE IX)*, pages 15–20, 2004.

[8] Daniela Rus and Marsette Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *International Journal of Robotics Research*, 22(9):699–715, 2003.

[9] Cypress Semiconductor. *PSoC Mixed Signal Array Final Data Sheet: CY8C29466, CY8C29566, CY8C29666, and CY8C29866*, November 2004. Document No. 38-12013 Rev. *G.

[10] Philips Semiconductors. *The I^2C-Bus Specification, Version 2.1*, January 2000. Document Order No. 9398 393 40011.

[11] Philips Semiconductors. *LPC2106/2105/2104 User Manual*, September 2003.

[12] Solarbotics. Gm15–gear motor 15–25:1 6mm planetary gear pager motor. http://www.solarbotics.com/products/index.php?scdfa-250100084-viewDetail-productzq3945zq4categoryzq37=true, June 2006.

[13] Paul White, Victor Zykov, Josh Bongard, and Hod Lipson. Three dimensional stochastic reconfiguration of modular robots. In *Robotics Science and Systems.* MIT, June 8-10 2005.

[14] Mark Yim, Ying Zhang, Kimon Roufas, David Duff, and Craig Eldershaw. Connecting and disconnecting for self–reconfiguration with polybot. In *IEEE/ASME Transaction on Mechatronics, special issue on Information Technology in Mechatronics*, 2003.

[15] Eiichi Yoshida, Shigeru Kokaji, Satoshi Murata, Kohji Tomita, and Haruhisa Kurokawa. Micro self-reconfigurable robot using shape memory alloy. *Journal of Robotics and Mechatronics*, 13(2):212–219, 2001.

[16] Eiichi Yoshida, Satoshi Murata, Shigeru Kokaji, Akiya Kamimura, Kohji Tomita, and Haruhisa Kurokawa. Get back in shape! a hardware prototype self-reconfigurable modular microrobot that uses shape memory alloy. *IEEE Robotics and Automation Magazine*, 9(4):54–60, 2002.