Instrumented Tools and Objects: Design, Algorithms, and Applications to Assembly Tasks

by

Matthew N Faulkner

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author ....
Department of Electrical Engineering and Computer Science
September 7, 2010

Certified by ....
Daniela Rus
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Instrumented Tools and Objects: Design, Algorithms, and Applications to Assembly Tasks

by

Matthew N Faulkner

Submitted to the Department of Electrical Engineering and Computer Science
on September 7, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We developed an instrumented tool system comprised of wireless nodes and sensor systems to facilitate distributed robotic assembly tasks. This robotic system was deployed on two separate robotic assembly scenarios: one scenario used programmable autonomous beacons to facilitate precise localization of an assembly robot within a mock airplane wing, while the second used programmable assembly components to simplify sensing and coordination in a distributed, multi-robot assembly task.

An instrumented tool system comprised of two types of programmable nodes (beacons and assembly components) and two types of robot-mounted sensors was designed, implemented, and tested. On-board microprocessors allow each element of the system to perform sensing and communicate over an infrared communication protocol.

Algorithms for sensing and distributed communication were developed to perform local sensing tasks between assembly robots and instrumented materials.

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

Acknowledgments

I am greatly indebted to Daniela Rus for her support and guidance throughout the last year. It has been a pleasure to work along side the members of Prof. Rus' Distributed Robotics Lab, who possess a staggering amount of knowledge and expertise, with a willingness to offer support and advice. I am grateful for their insightful suggestions and practical help during this project.

I owe special thanks to my partners Adrienne Bolger, Seung Kook Yun, Manas Menon, and David Stein, each of whom contributed a facet to this work and played an integral part in carrying this project through design, implementation, and deployment.

This thesis was made possible by support from the Boeing Company.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of multiple coordinated robots for assembly has been well established in the limited context of automated factory assembly. In such scenarios, a restrictive set of conditions are required, including centralized control of the assembly robots, absolute sensing of the position of the robots and workpieces, and precise knowledge of the sequence of actions needed to produce an assembly. In many important cases these conditions cannot be met; while relaxing any of these conditions creates many new challenges, new solutions and applications are available when the control, sensing , and communication tasks can be shared between the assembly robots and the components they are assembling.

This work considers the use of instrumented tools and components, defined as modules containing processing, sensing, and communication, to facilitate robotic assembly in loosely constrained environments. Instrumented components provide an alternative to high-level sensing and absolute task knowledge by allowing the components comprising an assembly to perform local sensing, maintain their state, and respond to queries issued by the assembly robots. The physical location of the modules within an assembly workspace allows distributed algorithms to take on physical significance with respect to specific assembly tasks.

Instrumented tools and components offer many new possibilities for robotic assembly. As an example, consider a group of robots assembling a structure from materials located in a loosely organized supply depot. A robot, unaware of the progress made by other robots, could query the partially completed structure directly for the next task to be performed. The robot, in need of specific materials to perform the task, could broadcast requests for needed parts; a correct part, in turn, echos responses, leading the robot to its location. Returning to the structure with the correct part, the robot performs the assembly task, aided by sensors located on the part specific to the task.

We develop a particular robotic system to address two forms of robotic assembly scenarios: cooperative multi-robot assembly using intelligent assembly fasteners, and precise 1-D robot localization using intelligent positioning beacons. These applications demonstrate the use of embedded sensing and local communication among assembly components to simplify an otherwise prohibitively complex robotic assembly task.

In the first scenario, modular instrumented tools and components are used to facilitate identification, localization, and manipulation tasks. By exploiting processing, communication, and battery power within the materials used for construction, robots with simple and inexpensive end effectors are able to reliably identify different types of parts and accurately grasp and place parts without being given the precise location of the parts.

In the second scenario, a difficult assembly task inside an airplane wing is explored. Programmable, Intelligent Beacons installed within the wingbox interact with an optical sensor installed on a magnetically actuated robot to produce estimates of the robot's position. We demonstrate that the inexpensive beacons provide position information accurate enough to provide ± 0.05", and with low enough latency to close the loop on the system's highly nonlinear dynamics.

Implementing these systems produced many hardware design challenges, and required many trade-offs among competing design goals such as size, computational power, sensing modalities, and communication capabilities. In many cases, these design goals arose from specifics of the tasks at hand, such as weight limits imposed by the robotic manipulator arm, or by the precision needed to align an assembly. An early design choice required all components to possess their own power source, a processor, and to implement a common wireless communication framework. Optical communication was selected as an acceptable compromise of small size, low power, and useful range. Optical communication, as a line-of-sight medium, also allowed communication to double as a sensing modality to detect alignment and proximity.

In addition to hardware challenges, implementing the instrumented component system posed significant software challenges. Working within the hardware design requirements, a software infrastructure for message passing was developed. In particular, embedded algorithms needed to be written to reliably pass messages while both receiver and transmitter independently multiplexed their available communication channels. Higher-level algorithms were written to be robust against the inevitable dropped packet or failed node.

The remaining chapters are arranged as follows. Chapter 2 reviews related work in the fields of programmable matter, industrial automation, and robot sensing. Chapter 3 describes the hardware modules developed and outlines the design trade-offs made. Chapter 4 explains the embedded software layer run on the modules to perform sensing and communication, as well as the higher-level interface and control software needed to integrate the components into their robotic systems. Chapter 5 describes one application of the programmable matter system to accurately perform closed loop control of a magnetically actuated robot. A second application, reliable part identification and grasping of modular construction materials, is explored in Chapter 6. The behavior and performance of the systems are quantified and analyzed through a series of experiments, presented in Chapter 7. Finally, insights and lessons learned through creating these systems are discussed in Chapter 8.

Chapter 2

Related Work

Nearly all robotic assembly tasks involve a few operations that must be performed repeatedly and reliably. These operations include identification and position sensing of the assembly materials. Even in scenarios less structured than a traditional factory floor, robots typically manipulate materials drawn from a small, fixed set of specialized materials, and so it is natural to consider ways that these materials may be enhanced to simplify and increase the reliability of part identification and position sensing tasks. Our work builds upon previous and ongoing work to enhance construction materials with communication, and relates to other approaches to part identification and position sensing, such as computer vision and RFID.

Our work uses communication-enhanced materials for accurate identification, position sensing, and grasping. The concept of embedding communication capabilities into materials has been well established by RFID tags. These tags store a static piece of information, and communicate passively when energized by a nearby RFID sensor. RFID has been used for reliable part identification of individual materials [9] [11]. New approaches have explored the use of RFID as a position sensor. Foveated RFID has been explored as a means of high-acuity short-range part detection [3], while RFID transceiver characteristics have been used to provide estimated part position sensing [10]. However, unlike RFID, our assembly materials contain power, processing, and active communication. The ability to process and store information creates many new applications beyond the capabilities of RFID.

The ability to access and manipulate information stored in communication-enhanced materials creates new possibilities for robotic assembly. The value of tracking fixed information about assembly materials through an industrial manufacturing process has been demonstrated in work flow and inventory optimizations [8]. By extending to dynamic information, our materials can capture the up-to-date state of an assembly in progress. For example, [12] uses building blocks with embedded computation and communication which store environmental information used to speed up distributed robotic construction tasks.

Our work shares several goals with vision-based solutions to manipulation. Several approaches to vision-based object recognition are currently being pursued [4], with barcodes and fiducial markers being perhaps the simplest. Model-based approaches are able to take advantage of the CAD models typically produced prior to creating assembly materials [6]. Computer vision has also provided means for closed-loop positioning of robot manipulators [1]. These approaches do provide part identification and position sensing, but, unlike our system, incur a high computational cost and cannot store information in the assembly materials.

Finally, our work using communication-enhanced positioning beacons for accurate positioning and closed-loop control of a magnetically actuated robot extends previous work in industrial automation in magnetic actuation and optical positioning. Magnetic levitation with optical positioning is considered in [2] and [7]. Our work extends this work to consider the use of multiple beacons with communication and processing for optical positioning.

Chapter 3

Electronics for Instrumented Tools and Objects

We developed a hardware platform composed of three electronic modules that allow a robot to perform accurate assembly operations. The platform contains two types of autonomous nodes, the Intelligent Positioning Beacons and the Intelligent Assembly Components, which integrate sensing and communication into materials used for assembly tasks. The platform also includes an Interface Module with infrared transceivers that can communicate between a robot and the autonomous nodes. Additionally, the interface module is capable of using an optical sensor for precise linear positioning. These three electronic modules were deployed in two different robotic assembly systems: Chapter 5 describes the use of the Intelligent Positioning Beacon and the interface module to achieve accurate closed-loop linear motion, while Chapter 6 describes the use of the Intelligent Assembly Components and the interface module to perform accurate part identification and manipulation.

This chapter will introduce and describe each of the electronic devices developed. First, the core infrastructure of the electronics system is described in Section 3.1. This infrastructure is comprised of the embedded microcontroller which enables each element of the system to perform processing and communication, as well as the infrared wireless communication interface that links the system together. Then, Section 3.2 describes the individual electronic modules: the Intelligent Positioning Beacon, the Intelligent Assembly Component, and the Interface Module.

3.1 Core Electronic Infrastructure

This section defines the processing and communication systems common to each element of the electronic assembly system. Processing is provided by an 8-bit microcontroller within each element. Communication between elements is achieved using serial communication over modulated infrared signals.

3.1.1 AVR Processor

A microprocessor allows every component of our system to store information about its state, control sensors, and communicate with other components. Each hardware component contains an 8-bit Atmega8 AVR processor made by Atmel. The Atmega8 provides 8-Kbyte of flash program storage, 512 bytes of EEPROM, 1-Kbyte SRAM, an SPI serial interface and a 10-bit A/D-converter. The Atmega8 is operated at a system clock rate of 14.7456MHz using an external crystal oscillator to provide an accurate clock source for error-free communication baud rates.

The hardware peripheral features utilized by our system include the Atmega's two 8-bit timers, its 16-bit timer, the interrupt-driven USART, and the 10-bit A/D-converter. These features are controlled in software to provide modulated infrared UART serial communication, PWM control of the average current supplied to the infrared LED transmitter, multiplexing of communication signals to four infrared receiver/transmitter units, and clocking and analog sampling of a linear photodiode sensor.

USART

The first hardware peripheral is the Universal Synchronous/Asynchronous Receiver Transmitter (USART). The hardware USART is used to serially transmit 8-bit characters, either wirelessly over infrared or via an RS-232 interface to a desktop computer. It is used asynchronously (UART) at 2400 bits per second (bps). Because a single byte requires approximately four milliseconds to transmit at this data rate, it is important that interrupts be used to notify the processor when a transmission is completed rather than waiting. The TX Data Register Empty interrupt indicates that a new byte may be loaded for transfer. This allows an entire array of data to be transmitted with minimal overhead. Similarly, the RX Complete interrupt allows the receiver to buffer each received byte and perform error detection with minimal overhead.

Timer 0

The next peripheral used is the 8-bit Timer0. Timer0 is used to generate the 184kHz clock source and control waveform needed to interface with the TAOS 1410r linear photodiode sensor used for optical positioning, described in Section 3.2.3. The Timer0 Interrupt Service Routine (ISR) defines the rising and falling edges of the clock waveform, allowing initiation of analog-to-digital conversion to be precisely synchronized with the analog signal clocked out of the photodiode. Further, the control signals produced by the Timer0 ISR provide flexibility in defining the sensor's sampling interval, in effect the "white balance" of the sensor, and allows subsampling of the sensor for higher data rates.

Timer 1

Another timer, the 16-bit Timer1, is used by all modules to produce the 38kHz carrier frequency needed for modulated IR communication. Timer1 is configured to produce phase-correct Pulse Width Modulation (PWM) automatically on an output pin using Timer1's hardware output compare unit. Phase-correct PWM allows the duty cycle, which directly influences the average power usage of the infrared transmitter, to be configured during use without introducing variation in the position of the pulses, thus providing more consistent modulation.

Timer 2

The third timer, the 8-bit Timer2, is used by all modules to generate 1ms timing events. These events are used primarily for defining the period of time allowed for messages to be received.

Analog-to-Digital Converter

The final peripheral of the AVR is the 10-bit Analog-to-Digital converter. The Atmega8 provides an 8-channel ADC (for TQFP and QFN packages only) that can perform up to 15000 samples per second (15ksps) at full resolution, with higher sample rates being possible at reduced resolution. The ADC can be configured to run constantly, known as Free-Running mode, or to perform single conversions, known as Single-Shot mode. The ADC converts an analog input to a digital value through successive approximation, requiring an ADC clock frequency between 50kHz and 200kHz (for maximum resolution) to be derived from the system clock. A single ADC conversion requires 25 ADC clocks when the ADC is first initialized to activate internal hardware, and 13 ADC clocks for subsequent conversions.

We use the AVR ADC in Single-Shot mode to convert analog sensor readings from an optical sensor to digital values. It was found that the full precision of the ADC was unneccessary, but that the maximum full-resolution sample rate of 15ksps was too slow. Consequently, the ADC clock source was set to 3.6468MHz, providing a sample rate of 280ksps. We found that the loss of accuracy was entirely acceptable; we discard the two least significant bits of the converted value.

3.1.2 Modulated Infrared Communication

The elements of the programmable matter systems interact with each other using wireless optical communication. These interactions include such messaging operations as querying the state of individual components, or to updating information stored in the components. However, optical communication, as a line-of-sight medium, implicitly conveys additional sensory information regarding proximity and unobstructed paths between communicating components. Our system utilizes the field of view of the optical components to determine the identity and relative position of nearby parts.

Creating a sensory system from optical communication requires specification of the range and field of view of the optical transmitters and receivers. Short range and narrow fields of view provide valuable sensory information by heavily constraining the possible locations where a part within communication contact may be, but at the expense of making establishing communication difficult in the first place. In the context of robotic assembly, the trade-off can be made by considering that certain interactions, such as manipulation, only take place between components within arm's reach of each other. This defines an upper limit on necessary range. Similarly, the needed field of view can be bounded by the degree of precision needed to align interacting components.

In light of these trade-offs, 38kHz modulated infrared communication was selected for the communication range, low power, and wide field of view of available receivers and transmitters. A similar communication protocol, IRdA, was considered for its higher data rate, but ultimately rejected on the basis of the short range and narrow field of view of available transceivers; however, in similar applications it may be a viable option.

Our system uses an infrared LED and a 38kHz demodulating infrared receiver to form a transmitter/receiver pair. The Atmega8's single UART serial channel is combined with a 38kHz modulation waveform, and multiplexed to provide 4 separate communication channels, each routed to a transmitter/receiver pair. Only one channel can be used at any time. While the Atmega8 provides full duplex operation (simultaneous transmitting and receiving) which is preserved by the multiplexer, our communication protocol software limits to half-duplex communication (alternating transmitting and receiving). This limitation was imposed to prevent a module from receiving reflected light from its own transmissions, which was observed to be a significant source of noise. Each IR channel is capable of line-of-sight communication at 2400 bits per second (bps). While the UART hardware is capable of much higher data rates, the demodulating receivers impose the 2400bps limit.

While unmodulated infrared or IRdA communication are possible alternatives, modulated infrared communication is appropriate for our network communication because of its small size, low power, long range, and wide field of view. In particular, modulated IR allows data to be transmitted at up to 7 feet with minimal error using as little as 20mA pulsed current. Each transmitter / receiver pair has a half-power field of view of approximately 120 degrees; that is, half of full signal strength is provided at up to 60 degrees from centerline. In practice, a field of view of nearly 180 degrees can be obtained if communicating parts are within a foot of each other.

3.2 Electronic Modules

This section describes in detail the three modules which comprise the instrumented assembly system: the Intelligent Assembly Components, the Intelligent Positioning Beacon, and the Interface Module.

3.2.1 Intelligent Assembly Components

Robotic construction tasks require a robot to correctly identify materials and accurately manipulate them. The Intelligent Assembly Components drastically simplify part identification and accurate manipulation by using local, line-of-sight communication as a substitute for complex sensors such as cameras or laser scanners. In conjunction with an instrumented gripper, optical communication can be used to obtain and modify information stored in each assembly component, providing a robust alternative to machine vision identification schemes. Utilizing the field of view of the optical transceivers, communication also provides accurate position information that can be used to manipulate the assembly components.

An Intelligent Assembly Component is an instrumented construction material comprised of a processing and communication node PCB, a rechargeable battery, and a modular assembly material. Together, these parts form an autonomous “smart part”.

Assembly Node PCB

At the heart of the Intelligent Assembly Component is a node PCB with an Atmega8 AVR microcontroller and four bidirectional infrared transmitter/receiver channels. The node contains the unique identification number of the part as well as data about the part, such as its physical characteristics, its role in an assembly, etc. The node also provides nearly omnidirectional wireless communication via its four infrared transmitters and receivers. These allow the part to communicate with robots or other parts reliably at distances of up to 60cm. Using a 3.7v 210mA rechargeable Lithium-Polymer and switching DC-DC converter, the node PCB can be run at 5v for between 4 and 15 hours continuously, depending on the ratio of time idle and time transmitting.

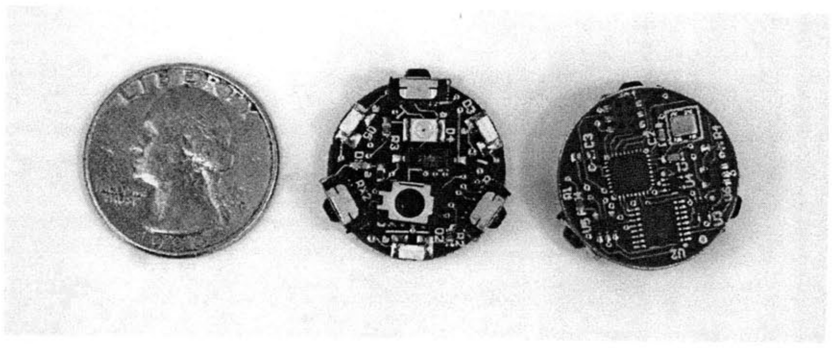
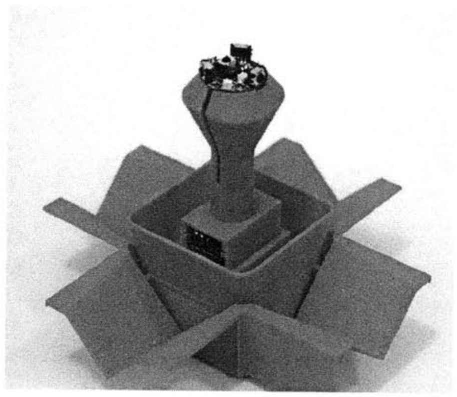


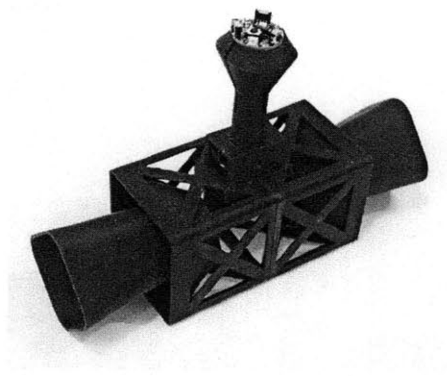
Figure 3-1: Intelligent Assembly Components contain a Processing/Communication node PCB

Modular Assembly Material

Construction of 3D structures is simplified by the use of two interlocking modular assembly materials which contain the Assembly Node PCB and a rechargeable battery. Modular struts are linked together with modular junctions to form 3D scaffold structures. A modular junction is depicted in Figure 3-2(a), and a strut in Figure 3-2(b). The use of these modular components is described in detail in Chapter 6.



(a) Modular Junction.



(b) Modular Strut.

Figure 3-2: The junction joins six struts in the Up, Down, North, South, East, and West directions.

3.2.2 Intelligent Positioning Beacon

Nearly all robots used for assembly and manufacturing require precise knowledge of their position in order to function. This information may be provided by proprioception, such as sensing joint angles using an optical encoder, or by an external positioning system, such as machine vision or laser positioning system. These positioning techniques impose limitations on the design of the robot and on the types of work envelopes in which it can function.

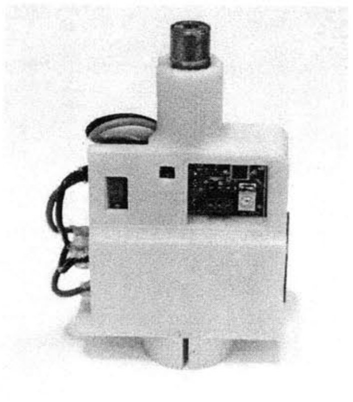


Figure 3-3: Intelligent Positioning Beacon

The Intelligent Positioning Beacon offers a new programmable matter solution for accurately positioning a robot within a work envelope. The intelligent positioning beacon serves to provide a reference signal that is precise enough to accurately determine a robot's position. This "local landmark" functionality is extended to providing ubiquitous position information by installing a network of intelligent beacons throughout a space. Individual beacons can be activated or deactivated, so that the robot is always capable of detecting one beacon. Each beacon uniquely addressable and capable of wireless infrared communication, allowing a robot to query the location of its beacon. Used in conjunction with a precise optical sensor, this system can reliably and accurately determine the position of a robot.

The intelligent positioning beacon is an autonomous instrumented component, with processing, two infrared communication channels, rechargeable battery, and a micro laser beacon module. The hardware elements of the beacon are depicted in Figure 3-4.

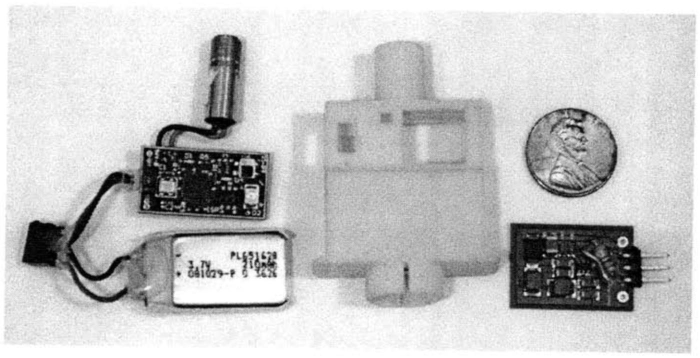


Figure 3-4: The components of an Intelligent Positioning Beacon.

Beacon Node PCB

The Beacon node PCB contains an Atmega8 AVR microcontroller and two bidirectional infrared transmitter/receiver channels. The node contains the unique identification number of the beacon and its position coordinates. The node PCB is capable of activating and deactivating one laser module. Using a 3.7v 210mA rechargeable Lithium-Polymer and switching DC-DC converter, the node PCB can be run at 5v for between 3 and 15 hours continuously, depending on the amount of time the laser is active, and the amount of infrared transmission. The PCB measures 0.5" by 1.0".

Micro Laser Module

Each beacon uses an MM650 micro laser module from U.S. Lasers to signal the robot. The laser module contains a 650nm 5mw red laser diode, an automatic power control circuit, and an adjustable colliminating lens. It measures 6.4mm in diameter, and 17mm in length.

The micro laser module was found to be vastly superior to directly driving a laser LED. In particular, laser LEDs are notoriously sensitive to drive current and are easily damaged without a proper current control feedback. Additionally, laser LEDs are not naturally colliminated; the beam of a raw laser LED typically diverges at 10 to 30 degrees. This divergence is a direct function of the semiconductor die size, and a precise beam can only be achieved through the use of a colliminating lens.

3.2.3 Robot Interface Module

The Robot Interface Module provides the link between a robot and the network of programmable matter beacons or fasteners. The interface module is also capable of processing an analog optical sensor to obtain precise positioning in conjunction with a positioning beacon.

The Robot Interface Module is comprised of a PCB with one serial port channel, three bi-directional modulated infrared serial channels, and (optionally) a TAOS TSL 1410r photo-diode array optical sensor. The serial port is used to communicate with the robot, while the three infrared channels allow directional communication with intelligent beacons and assembly components. Only one of the communication channels may be active at any moment, and so the interface module serves to route or multiplex communication between the robot and each infrared channel.

Each of the module's three infrared communciation channels is located on a small infrared receiver/transmitter board, measuring 0.5" by 0.5". These small boards allow the interface module's communication channels to be placed where they are needed, such as within an instrumented gripper.

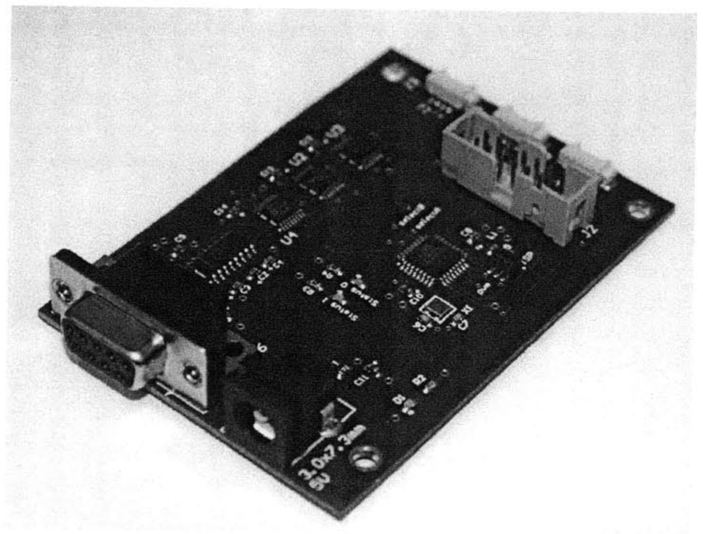


Figure 3-5: Robot Interface module.

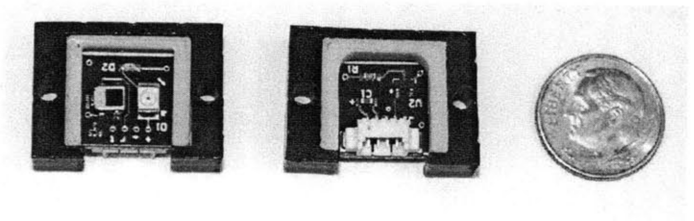


Figure 3-6: Infrared Receiver/Transmitter board allows placement of communciation channels onto the robot.

Optical Sensing

Optical position sensing is performed by the robot interface module by controlling a TAOS TSL 1410r optical sensor. The interface module produces the clock source, control signals and analog-to-digital conversion needed to detect the position of a beacon's laser on the sensor.

The TAOS TSL 1410r sensor consists of 1280 photodiode pixels arranged in a linear array with 400 Dot-Per-Inch pitch. The 1280 pixels are sectioned into two banks of 640 pixels. The sensor contains control logic to perform integration of each pixel output over the sensing interval, and two 640-bit shift registers that are used to clock out the analog pixel values. The sensor requires a clock signal between 5kHz and 8000kHz to control the sampling and shift register.

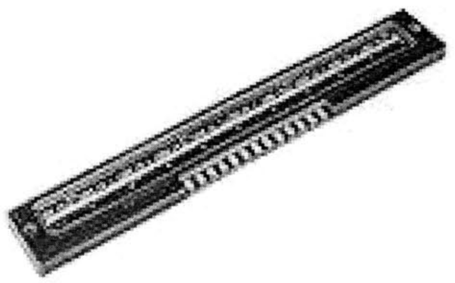


Figure 3-7: Optical Positioning Linear PhotoDiode Sensor

The robot interface module can control individual beacons to activate or deactivate their lasers. By detecting the pixel position of the laser, a 1D position is obtained, relative to the location of the active laser beacon. The module uses communication with the beacon to request the position, in world coordinates, of the beacon. In this way, the module is capable of producing positions, in world coordinates, in 0.0025" pixel increments.

Chapter 4

Communication Software

Integrating the instrumented hardware components described in Chapter 3 into robotic assembly systems required developing software to control the components, and specifying the communication protocol by which the components would interact with eachother and the robotic system. The first section of this chapter will outline the elements of the embedded C software run on the AVR microcontroller in eacd hardware component. This software provides routines to interface with sensors and a packet layer abstraction for sending and receiving messages over the infrared communication links. Section 4.2 describes the messaging protocol implemented on top of the packet layer, and describes the message types used.

4.1 Embedded AVR Software

In order to create robotic systems utilizing a collection of instrumented components, a software infrastructure was needed to control the sensor systems and provide networked wireless communication. In this way, applications can be built which use message passing remote procedure calls to interact with and sense their environment.

This section will begin with a description of the embedded software used to control the optical sensor and detect the position of a beacon's laser beam. Then, the communication infrastructure used to form, transmit, an receive packets of data is described.

4.1.1 Optical Sensing for Laser Beam Detection

The Robot Interface Module interfaces with a TAOS TSL 1410r linear photo-diode array sensor by utilizing the Atmega8's 8-bit Timer0 and its 10-bit analog-to-digital converter (ADC). The control waveform for the sensor and the ADC conversion is performed by a state machine. The state machine's transition events are defined by Timer0 interrupt service routines, and the sampling is performed by ADC interrupt service routines. Together, these allow an entire sensor scan to be performed in the background of other processes with minimal overhead.

The embedded software abstraction for the photo-diode sensor can be divided into a set of routines which perform initializations of hardware resources, and a sampling state machine that is invoked in response to regular timer overflow interrupts.

The rate of sensor readings and the "white balance" of the sensor are determined by a set of routines which initialize the properties of the 8-bit Timer0 and the ADC. Timer0 is incremented with each tick of the system clock, running at 14.74560MHz, and is configured to overflow at a rate of 368kHz. Each overflow event defines a rising or falling edge of the photo-diode clock signal, producing a clock signal of 184kHz. Overflow events also mark the transitions of the sampling state machine and the beginning of analog-to-digital conversions. The ADC performs successive approximation, and so a clock source is needed to define the sample-and-hold times. A clock of 3.6864MHz is configured at initialization. An ADC conversion requires 13 clock cycles, so this clock rate provides 283,000 samples per second. This allows all 1280 pixels of the photodiode to be sampled over 200 times per second.

4.1.2 Communication Packet Layer

The ability of a module to receive and respond to messages requires several steps; these steps are abstracted away by a communication packet layer. This software layer is responsible for creating packets from message data strings, controlling the multiplexer, modulation, and USART to transmit packets, and verifying the integrity of received packets.

A packet is a specially formatted string of characters that contains unique start (<), end (>), and separator ($) characters. It contains a 16-bit cyclic redundancy (CRC) checksum, transmitted as four hexadecimal characters, that is used on the receiver end to verify the integrity of the packet. Packets may be of variable length, but are required to be shorter than a maximum packet length, typically 48 characters. By convention, packets may contain only printable ASCII characters.

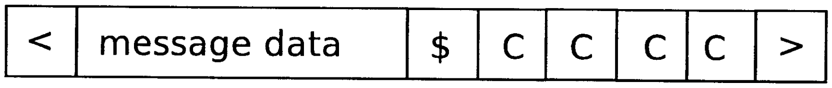


Figure 4-1: A packet is defined by its unique start, end, and checksum separator characters, with a four-byte hexadecimal checksum.

Transmitting a message involves verifying that the transmit buffer does not contain a message currently being transmitted, computing a CRC checksum, formatting the message data and four-byte hexadecimal CRC checksum according to the packet syntax (see Figure 4-1), buffering the packet for transmit, selecting the transmitter channel, and enabling the transmit interrupt service routine. Once enabled, the transmit interrupt service routine will transmit one byte at a time until the packet end character is reached; it will then disable the transmit interrupt, and set a *transmit_complete* flag. Interrupt-driven transmission and the *transmit_complete* flag allow the main program to perform other tasks while a message is transmitting.

Receiving a packet is slightly more complicated than transmitting for three reasons. First, the program should not listen indefinitely for a message to be received. Second, the receiver does not know the length of the packet it is receiving. And third, the received message may have been corrupted during transmission. A software receiver state machine, depicted in Figure 4-2, addresses these three issues.

To receive a packet, a timeout timer is initialized. This timer halts the attempted receive in the event that no packet arrives, or if a partial packet is received. This timer is reset when the start byte of a packet first arrives to ensure that the receiver does not time out while receiving is in progress. The receiver state machine is activated by enabling the receive interrupt. Each time a character arrives on the active IR channel, an interrupt is generated. The state machine begins in the *WaitForStartByte* state, and then transitions into the $BufferData$ state when a start character is received. Each subsequent character received is used to incrementally construct a checksum, and is placed into the receive buffer. The $BufferData$ state ends when the checksum separator character \$ is received. This character marks the transition into the *CompareChecksums* state. The next four bytes received are the hexadecimal representation of the message data's CRC checksum. When all four bytes have been received, this checksum is compared to the checksum computed during the $BufferData$ state. Any difference indicates that the message has been corrupted.

The checksum of the received packet is computed incrementally each time a byte is received in order to uniformly distribute the computation across many interrupt service routines. If, instead, the checksum were computed all at once, that interrupt service routine would take significantly longer than the others. Other interrupts are suppressed while servicing the receive interrupt, so long computations in one receive service routine could cause other interrupt-driven processes to behave irregularly.

Once the packet layer is in place, the program executed by each hardware module can be reduced to a simple listen-respond model, where the device's behavior is defined by the way it responds to different kinds of messages.

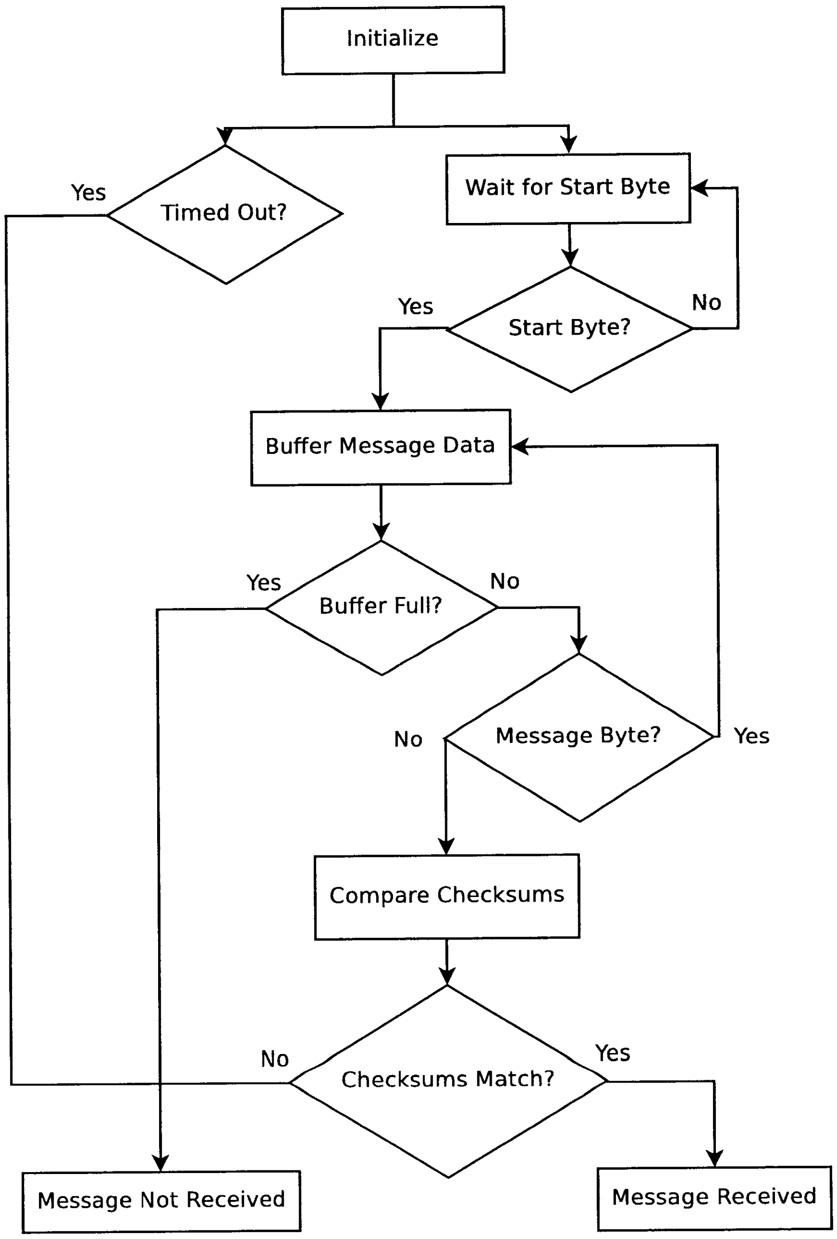


Figure 4-2: Receiver State Machine enforces maximum packet length, receiver timeout interval, and valid CRC checksum.

4.2 Messaging Protocol

Once a packet layer has been implemented, the behavior of a device can be specified by the way it creates and responds to messages. In this section, a collection of messages are described which define the communication between the different elements of our system.

4.2.1 Node Messages

The following messages are implemented by the Intelligent Positioning Beacon, the Intelligent Assembly Component, or both. All messages make use of the unique ID number stored in each autonomous node to indicate the sender or recipient of the message. The protocol reserves the unique ID number 255 as the *Broadcast_ID*; a message sent to this address will be responded to by every device which receives it.

Query Message

The *QueryMessage* is the way that the robotic system accesses information stored within an individual node. Each node contains a nonvolatile ID number in memory, as well as a fixed amount of allocated volatile memory. The *QueryMessage* requests that this information be transmitted.

The only parameter of the *QueryMessage* is an integer ID number used as the address of the message.

<	Q:n	$	C	C	C	C	>

Figure 4-3: Query Message packet syntax

Set Message

The *SetMessage* provides the means for the robot system to update the data stored in the volatile memory of one or more individual nodes. The *SetMessage* requires an integer ID number of the node addressed (or the unique *Broadcast_ID* for all nodes), and a string of data that will become the new data stored in the node. The protocol requires that a node respond with an ACK message, introduced below, when it has set its internal memory in response to a *SetMessage*.

<	Set:n,newData	$	C	C	C	C	>

Figure 4-4: SetMessage packet syntax

ID Message

The *IDMessage* is transmitted in response to a *QueryMessage*. The *IDMessage* indicates the address ID of the transmitting part, and contains the string of data contained in the part volatile memory.

<	ID:n,data	$	C	C	C	C	>

Figure 4-5: Message packet syntax

Set Transmitter Power Message

The parts of our system communicate using modulated infrared light. The specific duty cycle of the modulation - the percent of time that the infrared LED is on - is directly related to the power consumption of the device while transmitting, and is indirectly related to the range of the transmitter. The Set Transmitter Power Message allows this duty cycle to be adjusted in 10% increments between 10% and 90%. This allows the tradeoff between range and power consumption to be updated during use.

The *SetTransmitPowerMessage* takes as parameters the integer address ID of the recipient, and an integer between 1 and 9, inclusive.

<	POWER:n,power	$	C	C	C	C	>

Figure 4-6: Set Transmitter Power Message packet syntax

Activate Laser Message

The Activate Laser Message commands a specific beacon to turn on its laser module. The protocol requires that an ACK message be sent in response to an Activate Laser Message. This message is not implemented by the intelligent assembly components. The only parameter is the integer ID address of the recipient.

<	ON:n	$	C	C	C	C	>

Figure 4-7: Message packet syntax

Deactivate Beacon Message

The $DeactivateLaserMessage$ commands a specific beacon to turn off its laser module. The protocol requires that an ACK message be sent in response to a $DeactivateLaserMessage$. This message is not implemented by the intelligent assembly components. The only parameter is the integer ID address of the recipient.

<	OFF:n	$	C	C	C	C	>

Figure 4-8: Message packet syntax

Multi-hop Forwarding Message

The $Multi - hopForwardingMessage$ allows the range of communication to be extended beyond simple line-of-sight transmissions. The Multi-hop Forwarding Message uses a time-to-live number to implement a limited flood routing system. In this system, a forwarding message is created with a time-to-live number. When received, the recipient processes the forwarded message, then decrements the time-to-live number. If the decremented time-to-live number is greater than zero, a new Multi-hop Forwarding Message is transmitted using this number. This allows messages to propagate through a network of nodes while avoiding indefinite propagation.

The protocol requires that a node delay for a random interval of time before forwarding. This helps avoid network congestion. Currently, nodes delay for between 0 and 2 seconds.

The $Multi-hopForwardingMessage$ takes as parameter the time-to-live number, and the string data of another message.

<	FW:numHops,msg	$	C	C	C	C	>

Figure 4-9: Multi-Hop Forwarding Message packet syntax

Acknowledge Message

The *AcknowledgeMessage* provides a simple way for a node to indicate that it has successfully received a transmission.

<	ACK:n,msg	$	C	C	C	C	>

Figure 4-10: Acknowledge Message packet syntax

4.2.2 Interface Module Messages

The messages exchanged among the autonomous beacon and assembly component nodes, or between the nodes and the interface module, differ in syntax from similar messages exchanged between the robot and the interface module. This is because messages between the robot and the interface module contain additional routing information about which of the interface module's three infrared channels the communication is routed through.

Send Query Message

The Send Query Message commands the inteface module to transmit a $QueryMessage$ with a specified address ID through one of the interface module's three infrared communication channels. The interface module is required to attempt to receive a message on that channel for 250ms in order to receive any response. If an *IDMessage* is received in response, the interface module constructs a *ReceivedIDMessage*, defined below, and transmits this to the robot through the module's serial port.

The *SendQueryMessage* takes as parameter the integer address ID number used for the Query, and a single character 'A', 'B', or 'C' indicating which of the module's three infrared channels should be used.

<	Q:c,n	$	C	C	C	C	>

Figure 4-11: Query Message packet syntax

Send Set Message

The *SendSetMessage* commands the inteface module to transmit a Set Message with a specified address ID and contents through one of the interface module's three infrared communication channels. The interface module is required to attempt to receive an ACK message on that channel for 250ms. If an *AcknowledgeMessage* is received in response, the interface module sends an *AcknowledgeReceivedMessage* to the robot through the module's serial port.

The *SendSetMessage* takes as parameter the integer address ID number used for the *SetMessage*, the data to be used as the contents of the *SetMessage*, and a single character 'A', 'B', or 'C' indicating which of the module's three infrared channels should be used.

<	Set:c,n,newData	$	C	C	C	C	>

Figure 4-12: SetMessage packet syntax

Received ID Message

The *ReceivedIDMessage* is communicated from the interface module to the robot through the module's serial port to indicate that an ID message was received on one of the module's infrared channels. The message takes as parameter an integer ID address of the sender of the ID message, the data string contents of the received ID message, and a single character 'A','B', or 'C' indicating the infrared channel on which the ID message was received.

<	ID:c,n,data	$	C	C	C	C	>

Figure 4-13: Received ID Message packet syntax

Send Set Transmitter Power Message

This message commands the interface module to send a *SetTransmitterPower* message through one of the interface module's three infrared communication channels.

<	POWER:c,n,power	$	C	C	C	C	>

Figure 4-14: Set Transmitter Power Message packet syntax

Send Acivate Laser Message

This message commands the interface module to send an *ActivateLaser* message through one of the interface module's three infrared communication channels.The message takes as parameters an integer ID address of the recipient of the *ActivateLaser* message, and a single character 'A','B', or 'C' indicating the module's infrared channel to be used for transmitting.

<	ON:c,n	$	C	C	C	C	>

Figure 4-15: Send Activate Laser Message packet syntax

Send Deactivate Laser Message

This message commands the interface module to send a *DeactivateLaser* message through one of the interface module's three infrared communication channels. The message takes as parameters an integer ID address of the recipient of the *DeactivateLaser* message, and a single character 'A','B', or 'C' indicating the module's infrared channel to be used for transmitting.

<	OFF:n	$	C	C	C	C	>

Figure 4-16: Send Deactivate Message packet syntax

Send Multi-hop Forwarding Message

The $SendMulti-hopForwardingMessage$ commands the interface module to send a $Multi-hopForwardingMessage$ through one of the interface module's three infrared communication channels. The message takes as parameters an integer time-to-live number, a message to forward, and a single character 'A','B', or 'C' indicating the module's infrared channel to be used for transmitting.

<	FW:c,numHops,msg	$	C	C	C	C	>

Figure 4-17: Send Multi-Hop Forwarding Message packet syntax

Received Acknowledge Message

The *ReceivedAcknowledge* Message is communicated from the interface module to the robot through the module's serial port to indicate that an $AcknowledgeMessage$ was received on one of the module's infrared channels. The message takes as parameter an integer ID address of the sender of the $AcknowledgeMessage$, and a single character 'A','B', or 'C' indicating the infrared channel on which the $AcknowledgeMessage$ was received.

<	ACK:c,n,msg	$	C	C	C	C	>

Figure 4-18: Received Acknowledge Message packet syntax

Firmware Info Request Message

The *FirmwareInfoRequestMessage* is used in special cases to obtain a string of information identifying the interface module. In particular, this is used by applications to automatically detect and connect to an appropriate serial port for the interface module. It takes no parameters, containing only the question mark character.

<	?	$	C	C	C	C	>

Figure 4-19: Firmware Information Request Message packet syntax

Firmware Info Message

The *FirmwareInfoMessage* is sent in response to a *FirmwareInfoRequestMessage*, and provides meta-information about the device firmware. It takes no parameters.

<	FIRMWARE:t	$	C	C	C	C	>

Figure 4-20: Firmware Information Message packet syntax

Take Scan Message

The *TakeScanMessage* commands the interface module to perform one scan of its optical sensor, and to transmit either a *LaserPositionMessage* or *LaserNotDetectedMessage*, both of which are defined below. The message takes no parameters.

<	SC	$	C	C	C	C	>

Figure 4-21: Take Scan Message packet syntax

Fast Scan Message

The *FastScanMessage* is similar to the *TakeScanMessage*, but commands the interface module to perform multiple scans and to transmit the results of those scans.

This allows for higher throughput that is obtaining by using communication to initiate each scan.

The *FastScanMessage* takes an integer parameter for the number of scans to be performed.

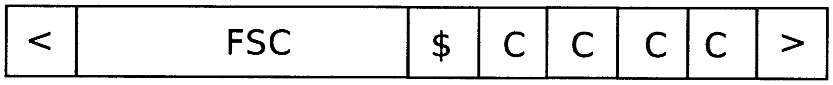


Figure 4-22: FastScan Message packet syntax

Laser Position Message

The *LaserPositionMessage* is sent from the interface module to the robot in response to a *TakeScanMessage* or a *FastScanMessage* in the event that pixel values exceeding a preset threshold were detected.

The message takes as parameter the integer pixel number of the laser position.

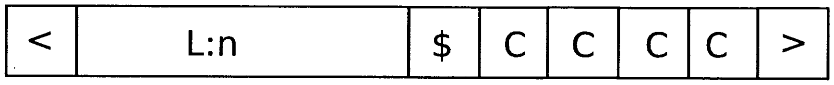


Figure 4-23: Laser Position Message packet syntax

Laser Not Detected Message

The *LaserNotDetectedMessage* is sent from the interface module to the robot in response to a *TakeScanMessage* or a *FastScanMessage* in the event that no pixel sampled exceeded the laser detection threshold.

The message takes no parameters.

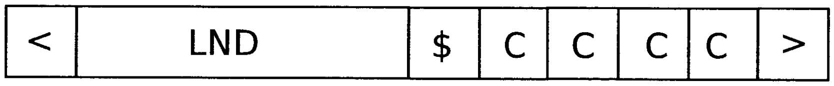


Figure 4-24: Laser Not Detected Message packet syntax

4.2.3 Embedded Control Program

Once the sensor routines, packet layer, and messaging have been implemented, the main function of the beacon, assembly component, or interface module reduces to a simple listen-respond loop. This can be viewed as a master-slave relationship between the robot and the interface module and a master-slave relationship between the interface module and the autonomous nodes. In both cases, the slave listens for messages from a master, and then performs a response to the message.

4.3 Application Software

Higher-level application software can be written by implementing the messaging protocol. This cleanly separates development of algorithms and control from the low-level implementation details of theh programmable matter system components. Chapter 5 describes an application layer written in the National Instruments *G* graphical programming language to provide closed-loop robot positioning, while Chapter 6 describes an application layer written in Java to perform object recognition and positioning.

Chapter 5

Intelligent Fasteners for Accurate Closed-Loop Linear Positioning of an Assembly Robot

Automated assembly of complex products presents many difficult design choices when planning for parts positioning and assembly strategies. As a motivating scenario, we consider the challenging task of installing fasteners inside an airplane wing. This task requires high precision within a large work envelope. Further, the work envelope is irregularly shaped, with a formidable ratio of depth to width, posing significant challenges for traditional articulated manufacturing robots.

In response to these challenges, a novel assembly system was developed that uses a pair of robots, one inside the airplane wing and one outside, that cooperate to accurately move within the wing. Specifically, the robot outside of the wing is equipped with powerful electromagnets and is actuated by an external positioning system; the inner robot is affixed to strong permanent magnets. Together, this allows the outer robot to exert magnetic forces on the inner robot, thus moving it. In this way, the two robots may move together, one on either side of the surface of the wing. Working as a pair, the robot system could cooperate on such tasks as inserting a bolt from one side and applying a nut on the other. Simple tasks such as these are currently impractical for robots in such a constrained space.

In order to position the robots accurately, a closed-loop control scheme is needed. This requires accurate measurements of the inner robot's position. We develop a solution for accurate, non-contact position sensing using intelligent fasteners. These fasteners are installed inside the wing beforehand and can be wirelessly controlled to emit a laser beam; the inner robot then determines its position relative to the laser beam. Then, using a wireless communication protocol, the robot can identify the fastener emitting the laser and obtain the fastener's world coordinates. With this information, the robot can determine its absolute position.

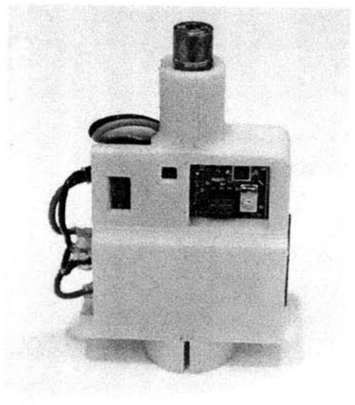


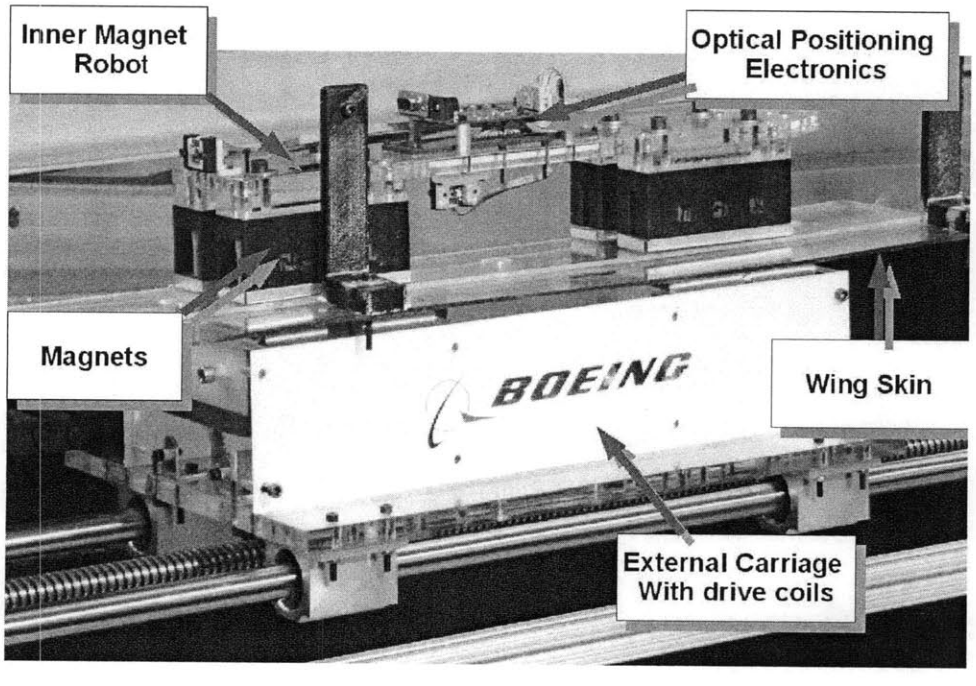
Figure 5-1: An intelligent fastener contains power, communication, and a laser beacon.

In this chapter, we describe an assembly system to evaluate magnetically-actuated robot motion within a mock airplane wing. First, the mechanical system, composed of the two magnetic robots, is detailed. Then, we describe a solution for sub-millimeter robot position sensing using intelligent fasteners and optical sensing. This sensor information allows us to control the robot with accurate closed-loop linear motion. Finally, we discuss modifications and revisions made to the system that were found necessary for practical experiments.

5.1 Magnetic Robot System

The robot system is comprised of two mechanical parts. The first, referred to as the inner robot, is a sled with four powerful permanent magnet Hallbach arrays. The inner robot is placed inside the mock airplane wingbox. The second, referred to as the outer robot, provides the driving magnetic field via two large electromagnetic coils. These use Lorentz forces to move the robot horizontally without generating normal force that would clamp the two robots to the wing. The two robots are separated by 6mm of acrylic, which functions as the surface of our mock wing. The outer robot is mounted on a lead screw gantry beneath the wingbox which allows the outer robot to move with precise linear motion. The wingbox and the robots are shown in Figure 5-2.

Control of the carriage's position and the current through its coils is provided by a National Instruments cRIO-9074 real-time I/O module. The cRIO contains a 400MHz real-time computer, and a 2M gate FPGA. Additionally, the cRIO serial port is used to interface with the infrared communication system.


Inner Magnet Robot
Optical Positioning Electronics
Magnets
BOEING
Wing Skin
External Carriage With drive coils
Figure 5-2: Wingbox with inner robot (above) and outer robot (below).

5.2 Positioning using Intelligent Fasteners

The Intelligent Fastener positioning system provides the accurate position measurements which are needed for the magnetic robot system to achieve controlled motion. Position measurements are obtained by using an optical sensor mounted on the inner robot to detect a laser beam emitted by an intelligent fastener. The robot can interact with the fasteners using wireless communication to identify fasteners, control their lasers, and query or record information in a fastener. The inner robot moves linearly over one or more beacons; at all times at least one beacon is beneath the robot's optical sensor, ensuring that position measurements may always be obtained.

5.2.1 Position Measurement

The first requirement of stable, closed-loop position control is obtaining accurate, low-latency sensor estimates of the inner robot's position. Position of the inner robot with respect to the wingbox is measured by a laser beam originating from an intelligent fastener that impinges on a linear photodiode array sensor mounted on the underside of the inner robot. The placement of the fasteners onto the surface of the wing beneath the robot is shown in Figure 5-3. More details about the intelligent fastener can be found in Section 3.2.2.

Determining the position of the inner robot relative to the wingbox requires two steps. In the first step, the inner robot uses infrared communication to issue a $QueryMessage$. An intelligent positioning fastener responds to this query, providing the robot with its identification number and its position in the wingbox coordinate frame. The robot then issues an $ActivateLaserMessage$ to the fastener to turn on its laser beacon. For more details on the messaging protocol, see Section 4.2.

In the second step, the inner robot uses its optical sensor to measure the position of the fastener's laser beam. The beam is considered *detected* if the optical sensor detects any pixel values exceeding 95% saturation. The position of the beam is determined as the center of the largest interval above the saturation threshold. This is intended to eliminate spurious peaks in the signal; however, in practice, the laser

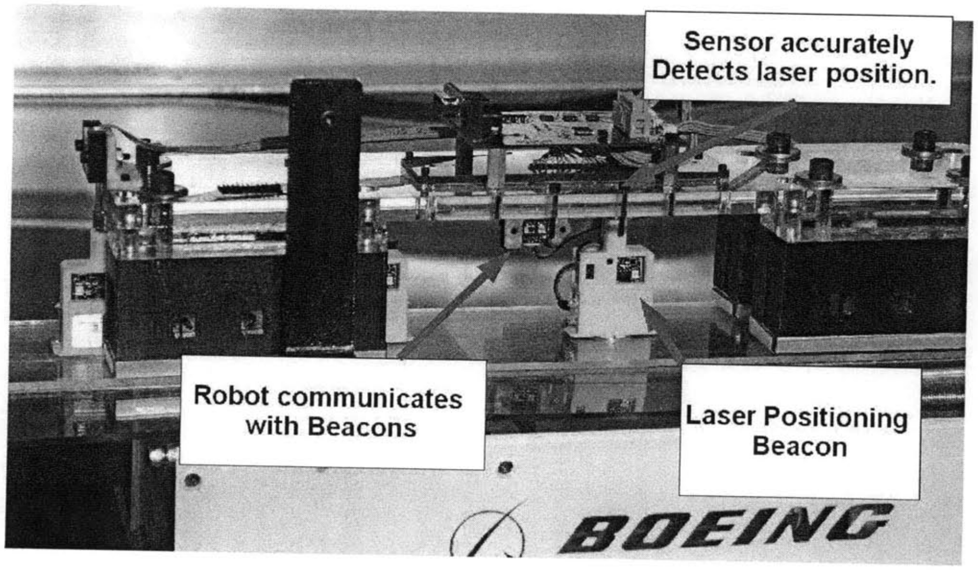

Sensor accurately Detects laser position.
Robot communicates with Beacons
Laser Positioning Beacon
BOEING

Figure 5-3: The inner robot communicates with intelligent fasteners mounted to the wing. An optical sensor mounted on the underside of the inner robot detects a fastener's laser beacon.

beam is reliably the only signal to exceed the threshold.

Using the pixel index, it is a simple matter to use the known $63.5\mu m$ pixel center-to-center spacing to compute the offset of the laser beam relative to the robot. This offset, along with the active fastener's position in the wingbox, allow the inner robot to determine its position. The length of the photodiode, at 10.2 cm., determines the range of motion possible using one beacon.

5.2.2 Closed Loop Control

Closed loop control requires measuring the robot's position, positioning the carriage beneath the robot, and driving current through the carriage's electromagnetic coils. The carriage can easily be controlled by PID to gently move to the robot's position. However, controlling the inner robot is more difficult due to its highly nonlinear dynamics: the robot is stationary until between four and five amps of current are applied to the coils. Then, the robot moves with nearly maximum velocity. A PID controller is used to command a stable current through the carriage's electromagnetic drive coils as the robot moves. A simple bang-bang control is used on top of that to set the coil current. While a thorough characterization of the robot's dynamics would allow a more sophisticated control scheme, with sufficient sensing simple bang-bang control is adequate for millimeter-scale positioning.

The control loop for the robot and carriage is depicted in Figure 5-4. Once a beacon has been identified and activated, the control loop uses bang-bang control with hysteresis to push the robot towards the target position. The carriage's coils can only exert force on the robot when located beneath the robot, so the control loop will not drive the coils unless the carriage is within some tolerance, typically under 1cm., of the robot's position.

Achieving stable closed-loop control requires position measurements to be accurate and low latency. Experimentally, 12 measurements per second was determined to be the minimum sampling rate to achieve 0.05" positioning without oscillation if the samples had millisecond latency. However, infrared communication introduces non-negligible latency. Closed-loop control was achieved while using 25 and 40 samples per second, with latency between 40mS and 20mS, respectively.

Experimental results demonstrating closed loop motion with 0.05 in. accuracy are presented in Section 7.2.

5.3 Systems Implementation

Implementing closed-loop control for the magnetic robot exposed several necessary modifications and possible improvements. Several of these were implemented, while others were not possible.

Several issues arose surrounding the linear motion of the robot. First, irregularities in the wingbox construction made the surface nonuniform where the robot moves. Irregularities include the seams between acrylic sheets, the sag of the acrylic between

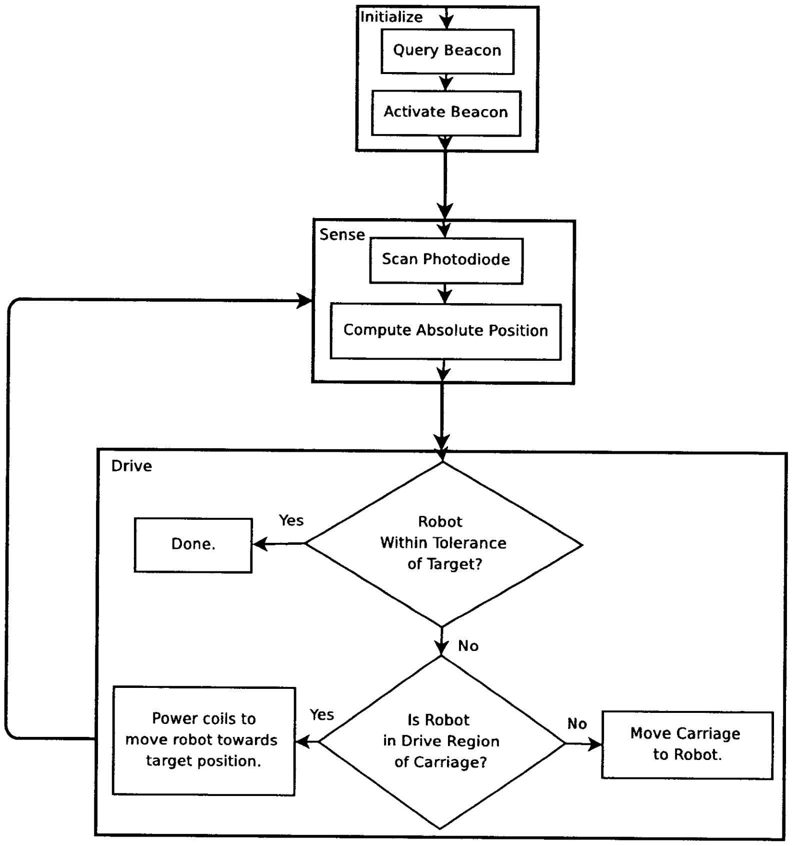

Initialize
Query Beacon
Activate Beacon
Sense
Scan Photodiode
Compute Absolute Position
Drive
Done.
Yes
Robot Within Tolerance of Target?
No
Power coils to move robot towards target position.
Yes
Is Robot in Drive Region of Carriage?
No
Move Carriage to Robot.

Figure 5-4: Robot and carriage control loop diagram.

supports, and irregular friction resulting from wear between the robot and the acrylic surface. These irregularities made the minimum amount of coil current needed to move the robot vary from between 4.4A and 5A. Higher currents result in higher velocity motion that is more difficult to control. These irregularities were considered inherent in the wingbox construction, and could not be corrected. Second, the wingbox was not well leveled, resulting in a sideways drift. This caused the robot's sensor to drift out of contact with the narrow laser beam. Two solutions were implemented to fix this: the lasers were focused to be ellipses wider in the sideways direction, and an acrylic rail was placed beneath the beacons to provide a smooth guide for linear motion.

Another significant issue arose regarding the latency of the sensor measurements.

While the initial specification of 10 position measurements per second proved approximately accurate (12 was determined to be the minimum), the system initially provided these measurements with approximately 90mS of latency. This latency was the result of the infrared communication rate of 2400bps. Such high latency was determined to produce oscillations in the motion control. Consequently, two modifications were implemented to reduce the position measurement latency. First, the message syntax was shortened to reduce the number of characters transmitted to a bare minimum. This was sufficient to obtain 40ms latency and adequate control. A second modification used a communication cable between the inner robot and the cRIO control module, enabling higher baud rates of up to 19200bps. This drastically reduced measurement latency to between 10mS and 20mS, producing much more stable of control. The impact of latency on positioning is presented in 7.2.

Chapter 6

Instrumented Objects for Grasping and Construction

In this chapter, we describe a robotic assembly system which uses optical communication between a robotic manipulator end-effector and a set of communication-enabled assembly materials to perform reliable part identification and grasping. As a motivating scenario, we consider building a cube from interlocking assembly materials.

First, the modular assembly materials are introduced in Section 6.1, followed by the design of the gripper end effector in Section 6.2. Then, Section 6.3 describes an algorithm that uses the components and gripper to perform reliable identification and gripping. A 3D construction application is presented in Section 6.4, and system implementation issues are presented in Section 6.5.

6.1 Modular Assembly Materials

Physical assemblies of Intelligent Assembly Components are possible by placing the Assembly Node PCB into a modular strut or connecting junction piece. Modular building materials were designed that allow complex 3D structures to be built, while still being practical for use with off-the-shelf robotic manipulators. Modular struts and junctions interlock both horizontally and vertically to allow scaffold-like structures to be built. The parts require only centimeter scale accuracy for placement, relying on the contoured design of the mating surfaces to fall into place precisely. Every part has a specially designed grasping point that can be passively aligned to a fully constrained position despite up to 2cm of misalignment. The parts are light weight and 3D-printed. Additionally, the modular components were produced in several colors, yielding a collection of heterogeneous building materials.

The modular strut, shown in Figure 6-1, can be used as a horizontal or vertical member. The Assembly Node PCB is placed in a specially contoured protrusion that serves as a grasp point for the strut. The strut is 18cm. long. With a rechargeable 3.7v 210mAh lithium polymer battery, the strut weighs 60 grams.

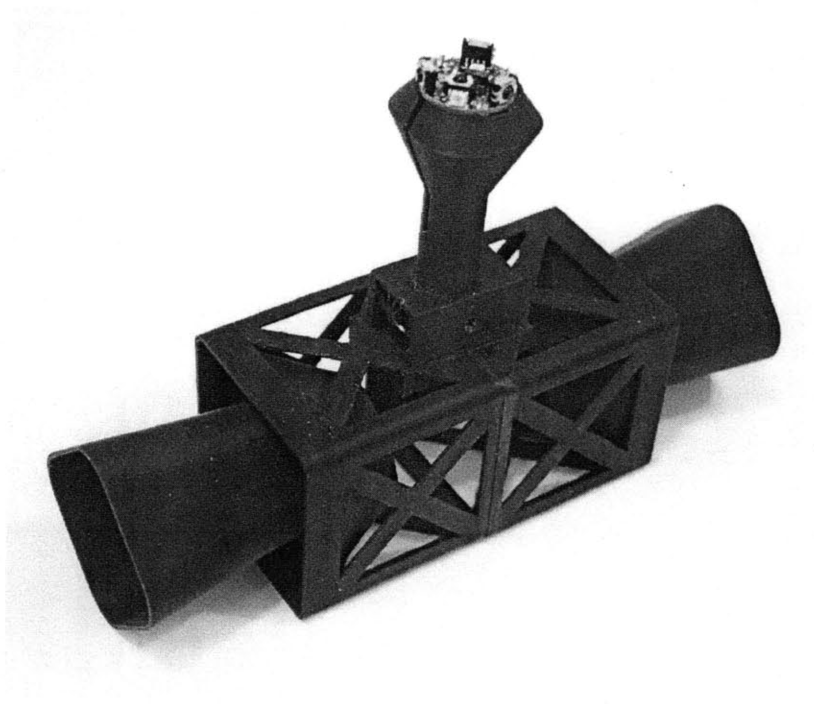


Figure 6-1: Intelligent Assembly Strut.

Struts are linked together with connecting junctions. The connecting junction, shown in Figure 6-2, is capable of connecting 6 struts in the North, South, East, West, Up, and Down directions. Using the connecting junction, 3D assemblies can be formed. Like the strut, the connecting junction contains the specially contoured grasp point containing an Assembly Node PCB and rechargeable lithium polymer battery. The connecting junction weighs 60 grams.

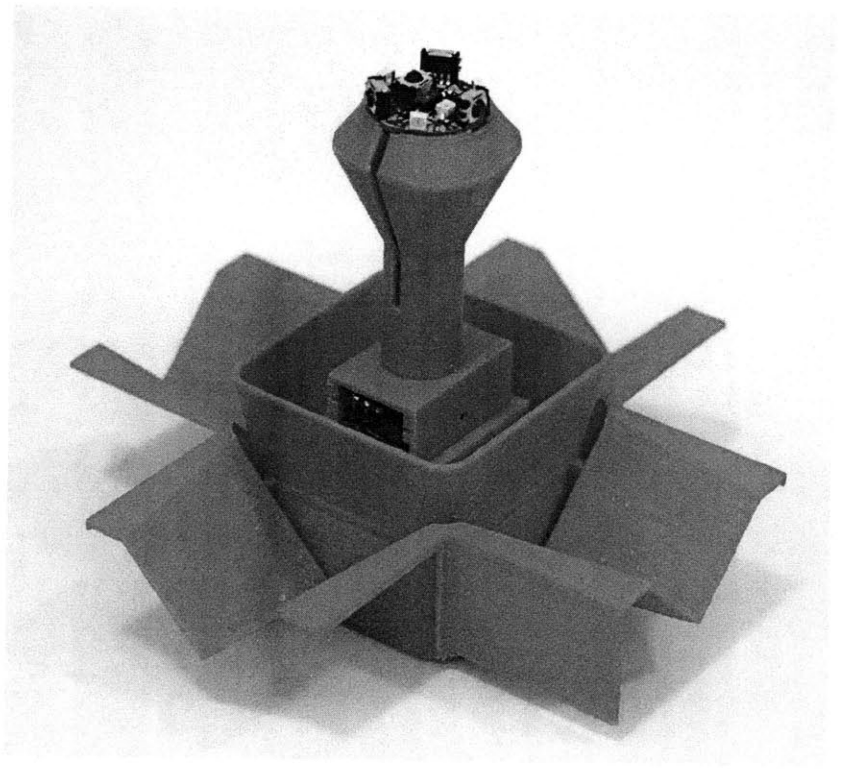


Figure 6-2: Modular Connecting Junction can connect six struts.

6.2 Instrumented Gripper

The instrumented gripper solves several problems faced by assembly robots. First, its special design allows it to reliably grasp parts despite centimeter-scale uncertainty in the part's position. It does this by passively aligning the grasp point into a unique orientation as the gripper closes. Second, the gripper allows the robot to identify individual parts. It accomplishes this by using an infrared receiver/transmitter PCB connected to an Interface Module to wirelessly communicate with nearby Intelligent Assembly Components. In effect, the gripper can "ask" the things in front of it if they are "graspable". Finally, the instrumented gripper solves the problem of precise positioning needed to reliably grasp by exploiting the field-of-view of its infrared communication: the gripper's field-of-view is exactly the region where a part can be located and still be passively grasped. Thus, establishing communication is equivalent to positioning the gripper for a successful grasp.

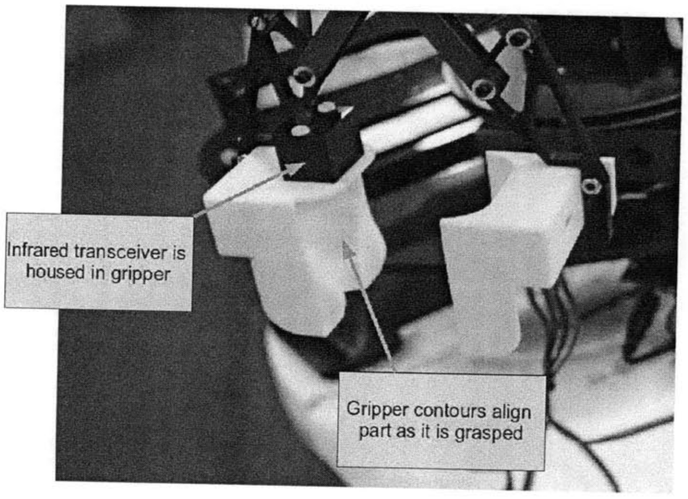

Infrared transceiver is housed in gripper
Gripper contours align part as it is grasped

Figure 6-3: Instrumented Gripper contains an infrared communication transceiver, and is contoured to align a grasped part as the gripper closes.

6.3 Precise sensing with Intelligent Components

Intelligent Assembly Components, used in conjunction with an Instrumented Gripper mounted on each robot, provides a solution to two challenges faced by the assembly robots: object recognition is provided by querying the building materials; and precise grasping is accomplished by a grasping algorithm that exploits the field-of-view of optical communication.

An assembly robot in relative proximity (60 cm.) to an intelligent assembly component can obtain all the sensory information it needs to identify the shape, color, and status of the component simply by asking. The robot may issue a $QueryMessage$ and use the existence of an $IDMessage$ response to determine if a component is present, and if so, if it is of the type the robot desires to perform its current task.

The intelligent assembly components make precise grasping of a component at an unknown location possible by using communication between the robot's gripper and the component as a sensing modality. The receiver/transmitter board placed within the instrumented gripper is oriented such that line-of-sight communication between the gripper and an assembly component is only possible when the gripper is held over the component. Specifically, the gripper is held 15cm. above the location of a potential component; the gripper will successfully grasp a component if it is within a 6cm. circle beneath the gripper. By choosing the placement of the receiver/transmitter board to provide a 6cm. field of view at a distance of 15cm, communication determines alignment of the gripper with a component.

6.3.1 Grasping Algorithm

A grasping algorithm defines the pattern of motion and communication needed to reliably find and grasp a component. For our experiments, it is assumed that the components' exact locations are not known, but that they are required to be within a linear region known as a "parts depot". This assumption was made due to the limited reach and accuracy of the robots' manipulators; linear search was more practical for experiments than attempting to move the manipulator in a raster pattern. We assume that a robot's knowledge of its position is sufficient for the robot to navigate to the parts depot. Within the parts depot, intelligent assembly struts and junctions may be placed. The components are not indexed into fixed positions, but may be at any point along a line.

The grasping algorithm is outlined in Algorithm 1.

The grasping algorithm begins once a robot arrives at the parts depot and orients its gripper above one end of the linear depot. The algorithm first locates a desired part to an approximate position estimate. Then, a fine-sensing pass is performed to accurately position the gripper over the component.

Once the robot has positioned its gripper over one end of the linear depot, the robot then begins moving its gripper over the depot, issuing $QueryMessages$ at every 5cm of linear motion. In this way, the field-of-view of successive communications overlap, allowing the entire area of the depot to be queried. If the robot desires a specific component, it may only address its $QueryMessages$ to the desired component's ID number. However, if the robot seeks any part with suitable properties, such

Algorithm 1 Locate-And-Grasp algorithm. *position_tolerance*, the field of view width *FOV* and *depot_length* are supplied constants.

```
1: $stepSize \leftarrow \frac{FOV}{2}$
2: $fineStepSize \leftarrow position\_tolerance$
3: $armPosition \leftarrow 0$
4: while $armPosition \leq depot\_length$ do
5:     send $Query$
6:     if desired $ID$ message received then
7:         while desired $ID$ message received do
8:             send $Query$
9:             $armPosition \leftarrow armPosition - fineStepSize$
10:        end while
11:        $a \leftarrow armPosition$
12:        while desired component responds to query do
13:            send $Query$
14:            $armPosition \leftarrow armPosition + fineStepSize$
15:        end while
16:        $b \leftarrow armPosition$
17:        $armPosition \leftarrow \frac{a+b}{2}$
18:        lower gripper and grasp component
19:        return TRUE
20:    else
21:        $armPosition \leftarrow armPosition + stepSize$
22:    end if
23: end while
24: return FALSE
```

as any assembly strut, the robot may issue *QueryMessages* using the *Broadcast_ID* number, and examine all responses for a match. When a match is found, a second phase of the algorithm begins.

When communication is established with a desired part, it can be inferred that the part is in some limited region beneath the gripper. The second phase of the algorithm performs linear search while performing finer motion between issuing *QueryMessages* to determine with greater precision the endpoints of the interval of successful communication. Then, the gripper moves to the center of this interval. In practice, the transmitting properties of the individual assembly components were found to be highly symmetric and repeatable, so moving to the center of the communication interval has proven a reliable means of centering the gripper over a component, with observed success rates well above 90%.

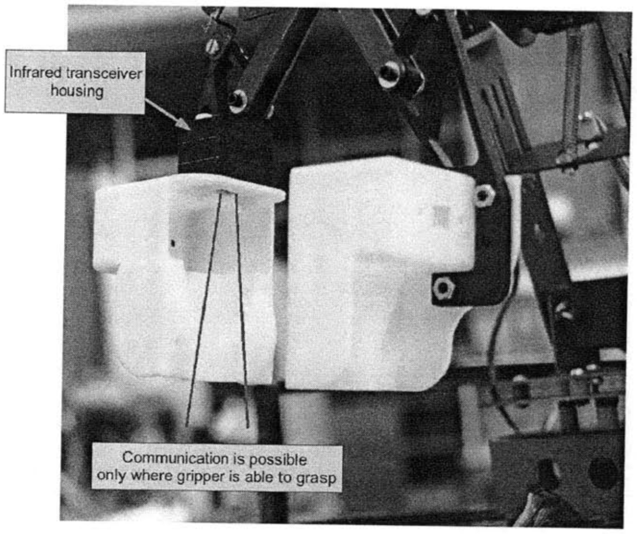

Infrared transceiver housing
Communication is possible only where gripper is able to grasp

Figure 6-4: The field of view of the gripper's infrared communication corresponds to the region of successful grasps.

6.4 3D Construction with Modular Components

At the time of writing, an ongoing application of the modular assembly materials and grasping algorithm is the construction of a 3D cube by a team of distributed assembly robots. The cube is constructed from 8 junctions and 12 struts from two colors of materials, depicted in Figure 6-5. Further, each robot is assigned a color, and is required to manipulate only materials of the same color.The ability to select and place materials of the correct color is representative of tasks such as selecting a bolt of the proper size and thread pitch, which a general modular assembly robot would certainly be faced with.

The implementation of our distributed robotic assembly system is derived from three design choices. First, the robots use a decentralized message-passing control scheme for task allocation and motion planning. This gives the system flexibility and scalability. Second, we assume that every robot knows its own position, and the position of all other robots, to a coarse scale. Here, "coarse scale" means that

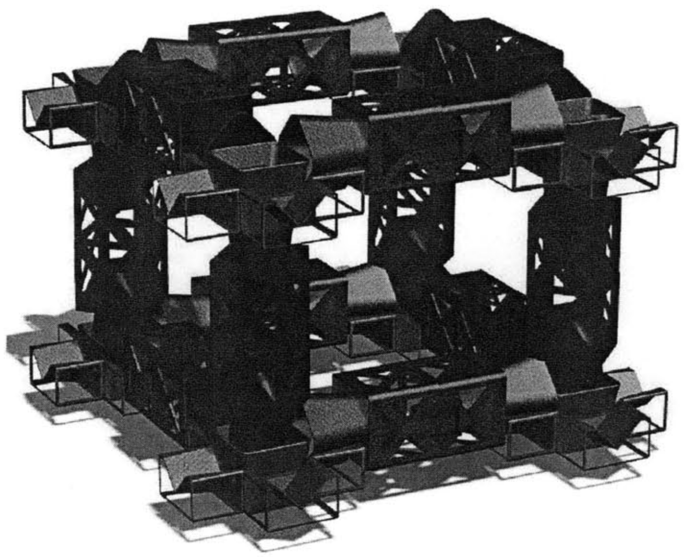


Figure 6-5: Modular construction materials can form 3D structures. This cube is constructed from 8 junctions, and 12 struts. The grasp points have been removed from the model for clarity.

the robot may be directed to "within arm's reach" accuracy of an object, but not with sufficient accuracy to find and grasp an object at a known location using open loop control. Finally, we assume that the location of parts within the workspace is constrained to within known regions, but that the specific parts therein and their precise locations are not known. We refer to these regions as "parts depots", and contrast them with the rigidly defined parts feeders, such as tape and reel machines, in conventional automated assembly.

6.4.1 Create Distributed Robotic Assembly Platform

Our robotic assembly platform is a group of iRobot Create robots, each equipped with a commercially available 5-DOF manipulator, mesh networking, and on-board laptop computer. These robots are used to perform distributed assembly of 3D node and truss structures using modular assembly materials. One such robot is depicted in Figure 6-6.

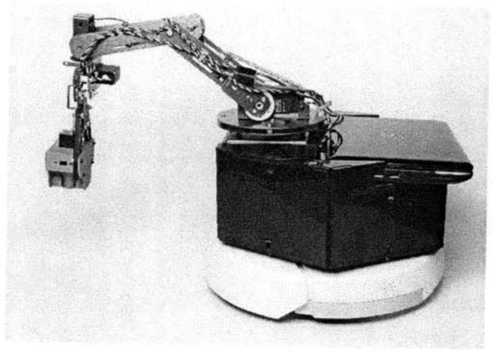


Figure 6-6: iRobot Create robot platform

6.5 Systems Implementation

In the course of system implementation and running experiments, several modifications and improvements were made to the system. Briefly, these included adjusting the field of view of the gripper's communication channel, using 1D search on a circular arc instead of linear search for components, automatically detecting and connecting to the serial devices of the system, and selecting more appropriate rechargeable batteries for the assembly components.

Initial experiments with the instrumented gripper revealed that the infrared transmitter was capable of transmitting through the 3D-printed plastic material of its housing, causing a very large effective field of view. This was remedied by placing the gripper's transmitter/receiver board within a black ABS plastic enclosure. This housing has an aperture through which communication is possible. By sizing the aperture, the positioning accuracy of the gripper can be controlled. In practice, we selected a positioning accuracy of 5 centimeters at 15 centimeters; that is, communication with an assembly component 15 centimeters below the gripper implies that the component is located in a 5 centimeter circle beneath the gripper.

Another refinement that was found necessary relates to the use of computer serial ports for communication between the robot and the interface module. Our robotic system uses three devices which interface via a serial port: the interface module, the Create robot base, and the LynxMotion manipulator servo controller. However, the laptops controlling the robots have only one serial port, but several USB ports. While

USB-to-serial adapters provide the missing link, the Ubuntu linux operating system on the laptops does not consistently assign identification numbers to these USB serial devices. Initially, this required specification of serial device numbers for each device every time the control software was started.

In response to the inconsistent serial device numbers, software was written to automatically detect which hardware device is connected to each USB serial channel. This was accomplished by creating a handshaking routine between the devices and the laptop. In particular, the laptop connects to each serial device in turn, attempting to establish communication using the protocol of the Create or the interface module. When communication is established, the port number is recorded in a lock file. In this way, the Create and interface module may be detected and automatically connected to. The lock file is used to establish the connection to the LynxMotion servo controller (which does not support a protocol conducive to handshaking) by process of elimination: the only device not listed in the lock file is connected to the LynxMotion board.

A final modification to the system involves a change to more appropriate, rechargeable batteries. The initial design of the intelligent assembly components used non-rechargeable lithium batteries, providing roughly an hour of use. While this was adequate for small tests, it was found that the mean time between failure when using a collection of components was too low to practically perform experiments. A rechargeable 3.7v 210mAh lithium polymer battery allows an assembly component to remain idle for up to 15 hours on a single charge, or transmitting continuously for over 3 hours. This has proven adequate for our experiments.

Chapter 7

Experimental Results

To evaluate the performance of the hardware and systems developed, experiments were performed to test such attributes as communication range, sensor latency, positioning accuracy, and grasping success rates. This section describes these experiments and presents the results.

As the core functionality, several aspects of the infrared communication system were empirically evaluated. These tests are presented in Section 7.1. Next, the performance of the optical positioning system is evaluated in Section 7.2 for accuracy and latency. The performance of the magnetically actuated robot using the optical sensor and intelligent beacons for closed-loop control is discussed in Section 7.3. Finally, the reliability of the instrumented gripper to identify and grasp an intelligent assembly component is evaluated in Section 7.4.

7.1 Communication Results

Infrared wireless communication links together the elements of the programmable matter system, and so the success of the system relies on reliable, well-quantified performance. Hundreds of tests were conducted to evaluate the effect of various system parameters on the reliability of communication. These parameters include the distance between axially-aligned transmitter and receiver; the angle between the transmitting infrared LED's optical axis and the line of sight to the receiver; the angle of incidence of the transmitted signal onto the receiver; and the duty cycle of the modulation waveform. Unless otherwise specified, all tests were performed with a modulation duty cycle of 50%.

7.1.1 Transmission Error Rates vs Distance

The effect of distance on transmission reliability was evaluated separately for the interface module's transmitter/receiver circuit board, and for the positioning beacon and assembly component node circuit boards. This was necessary because the interface module, not being limited by a small battery, uses a different maximum current through its transmitter LED. The tests were performed using a test fixture which held the transmitter fixed, and allowed the receiver to be moved to fixed distances between 0 and 10 feet from the transmitter. The jig ensured that the transmitter and receiver were kept parallel and axially aligned. The test measured the number of correctly transmitted characters by transmitting 1000 characters and counting the number of correct characters received. This process was repeated for each distance.

Figure 7-1 displays the results of these tests for the interface module's receiver/transmitter board. The data show that communication reliability is nearly error-less for unobstructed direct line of sight communication at ranges of up to 10ft. The experiment recorded over 15,000 consecutive error-free characters transmitted at ranges between 6in. and 9ft.

A similar test was performed with the transmitter used in the beacon node PCB and the assembly component node PCB. The results of this test are summarized in Figure 7-2. The data show that nearly errorless transmission is capable at up to 60cm.

These tests show that nearly error-free communication can be obtained during unobstructed, line-of-sight transmission at up to 10 feet in the case of the interface module, and up to 2 feet for the beacon and assembly component. The success rate drops off sharply after these distances.

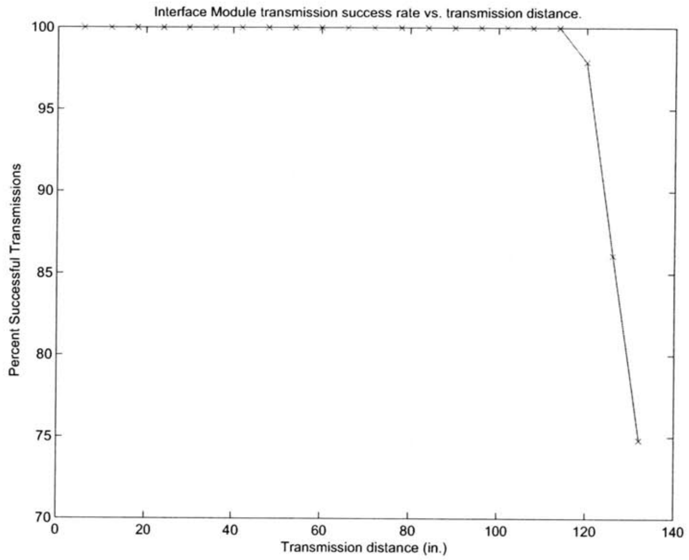


Figure 7-1: Interface Module transmission success rate versus distance. 1000 characters transmitted per distance.

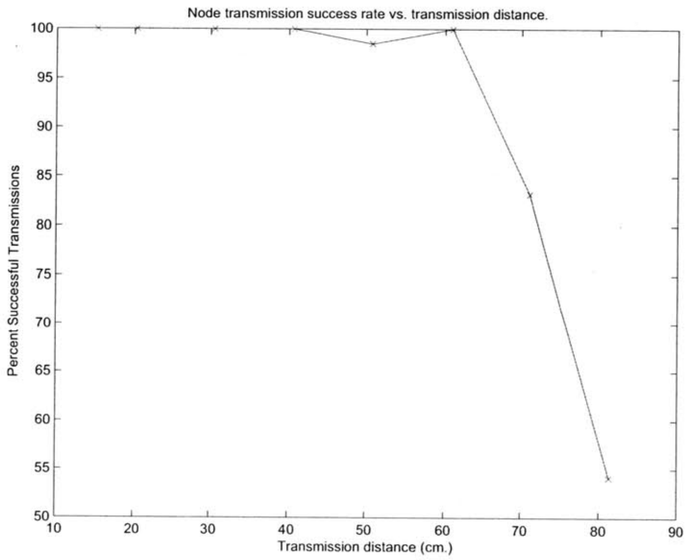


Figure 7-2: Beacon Node and Assembly Component Node transmission success rate versus distance.

7.1.2 Communication Error Rates vs Transmission Angle

Like the previous tests, this test uses a test fixture to control the distance and angle of the transmitter and receiver. Here, the receiver was maintained aligned with the transmitter, but the transmitter's optical axis was rotated away from the transmitter by a specific angle. This evaluates the width of the transmitted signal.

The first test in this category held the receiver at a distance of 240cm from the interface module's receiver/transmitter board, and varied the angle of the receiver/transmitter board. The results are presented in Figure 7-3. These results indicate that the transmitter field-of-view is reliably over 120 degrees. The test was performed at 240cm. because at shorter distances the field of view was consistently nearly 180 degrees.

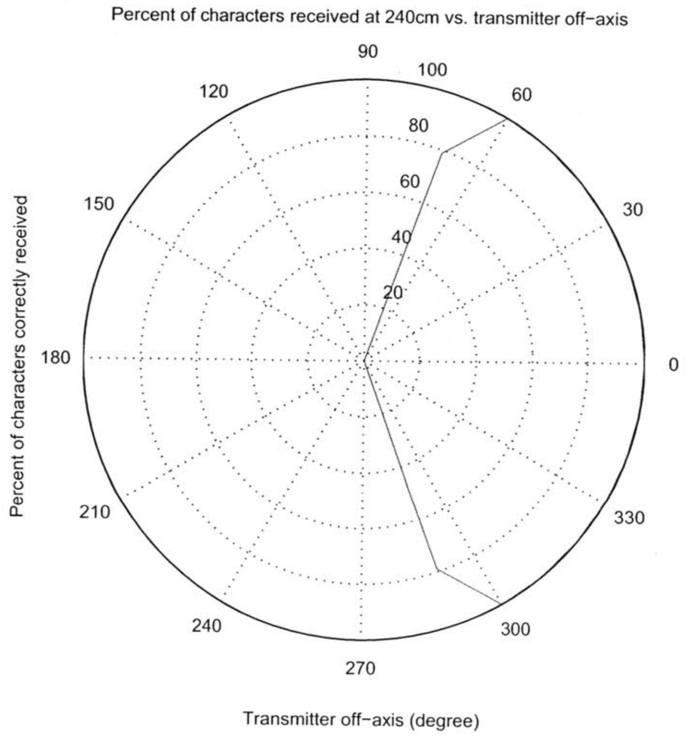


Figure 7-3: Interface Module transmission success rates at 240cm vs transmission angle.

7.1.3 Communication Error Rates vs Reception Angle

Another test was performed to evaluate the field of view of the demodulating receiver. Figure 7-4 shows a field of view of 140 degrees with nearly error free reception.

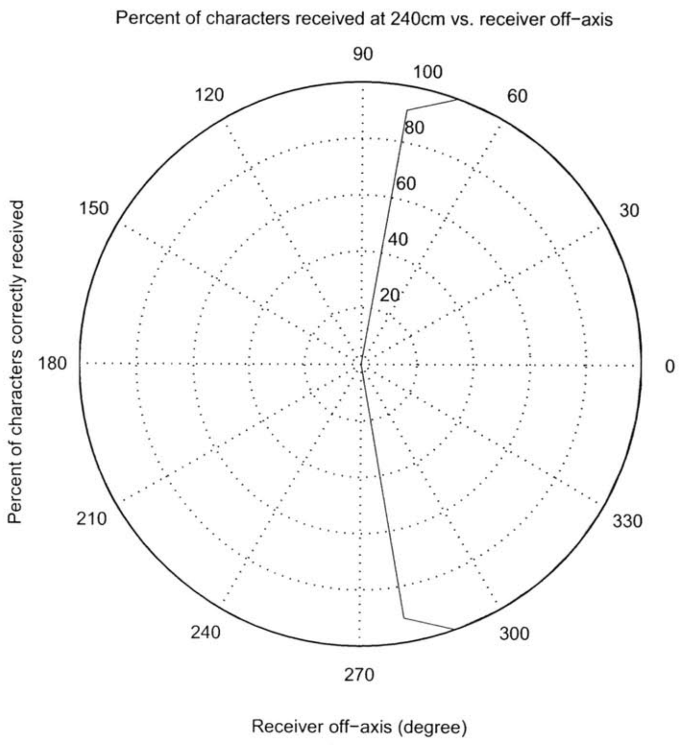


Figure 7-4: Transmission success rates at 240cm vs receiver angle.

7.1.4 Messaging Range vs Alignment Angle

The previous field of view tests characterize the individual components of the infrared communication system. This test measures the practical impact of the results by determining the maximum range at which the interface module can communicate with a beacon node or assembly component node, as a function of alignment angle. For this test, the interface module's receiver/transmitter board was held fixed, while a beacon node was rotated. 100 queries were issued, and the maximum distance for which at least 95% of the queries received error-free responses was recorded. Our communication protocol provides error detection, but does not attempt error correction.

Consequently, a character transmission success rate of at least 95% is recommended to maintain a high packet success rate.

Figure 7-5 presents the results. 60cm. of range can be achieved within ± 30 degrees, while 40cm. range is achievable for ± 60 degrees.

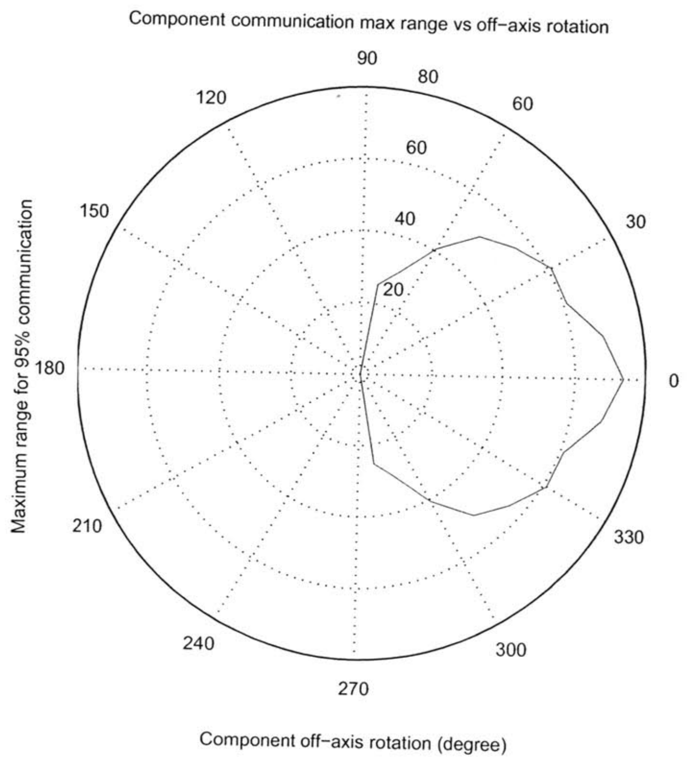


Figure 7-5: Maximum range for 95% query-response success vs. alignment angle.

7.1.5 Communication Range vs Transmit Power

All tests so far have performed infrared communication with a 38kHz modulation waveform of 50% duty cycle. The modulation duty cycle influences the average power applied to the transmitter LED, and so is a factor in transmission signal strength and battery usage.

While a higher duty cycle increases the infrared strength, the demodulating receiver appears to be most sensitive to a signal with shorter on periods than off. Consequently, 50% duty cycle was found to provide the longest range. However, it

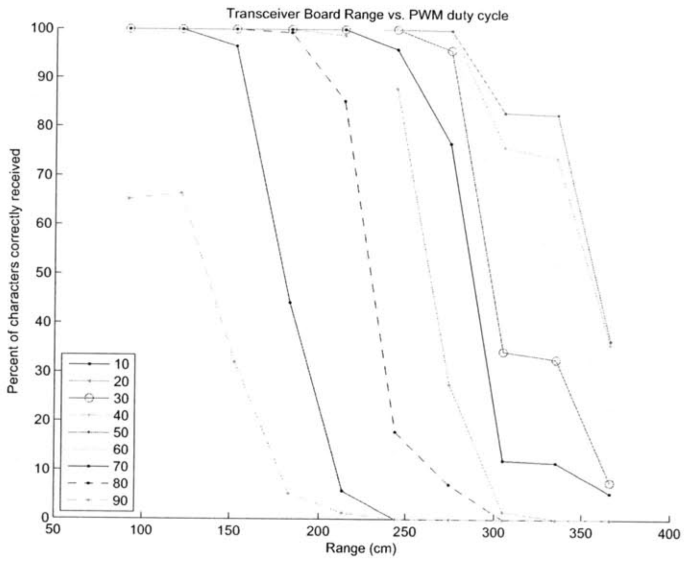


Figure 7-6:

is interesting to note that lower PWM values generally performed better than corresponding higher PWM values, for example 40% achieves longer range than 60%.

7.2 Optical Positioning Precision and Accuracy

This section evaluates the optical position sensing of the interface module and the TAOS TSL 1410r photo diode array. Measurements and comparisons were made using a linear variable differential transformer (LVDT), an accurate tool for measuring linear displacement.

7.2.1 Stationary Precision

The stationary precision of the optical sensor was measured by recording the detected position of the laser beam for 3500 samples. The precision of the sensor is compared to the LVDT after calibrating both sensor readings to be zero-mean. Figure 7-8 displays the results. The ration of the variance of the optical measurements to the variance of the LVDT measurements is 1.7:1.

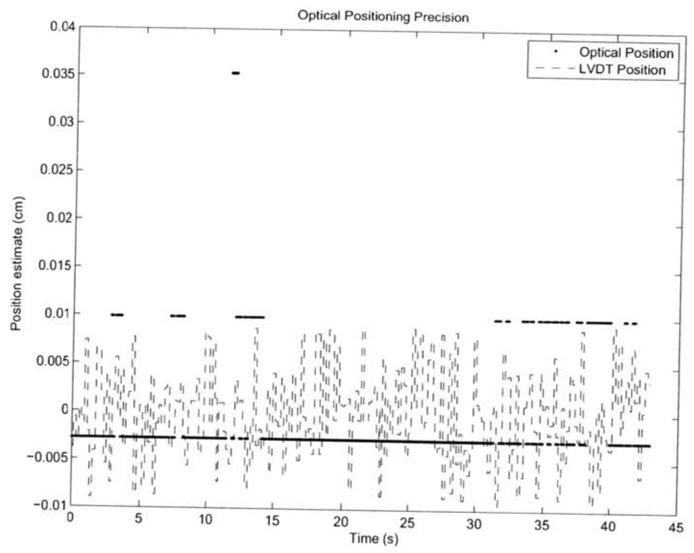


Figure 7-7: Optical Positioning Precision

7.2.2 Motion Accuracy

A separate test was conducted to measure the accuracy of the optical sensor position across a range of motion. For this test, an LVDT sensor with a 5cm range of motion was attached to the robot and used as ground truth. Prior to the test, 250 sample values from both the optical sensor and the LVDT were used to determine the mean zero position for each sensor for calibration. After calibration, the robot was moved 4.5cm (to stay within the 5cm range of the LVDT), and the mean absolute error between the optical position and the LVDT was recorded to be 0.254mm.

7.3 Magnetic Robot Control using Laser Beacons

These experiments evaluate closed-loop control of the magnetically actuated robot using laser beacons for optical position. Communication latency was found to be a significant factor in the stability of the control loop, and so these tests demonstrate results for both the standard 240bps infrared communication rate, as well as for 19200bps using a serial cable.

The first experiment was performed using the standard infrared communication at

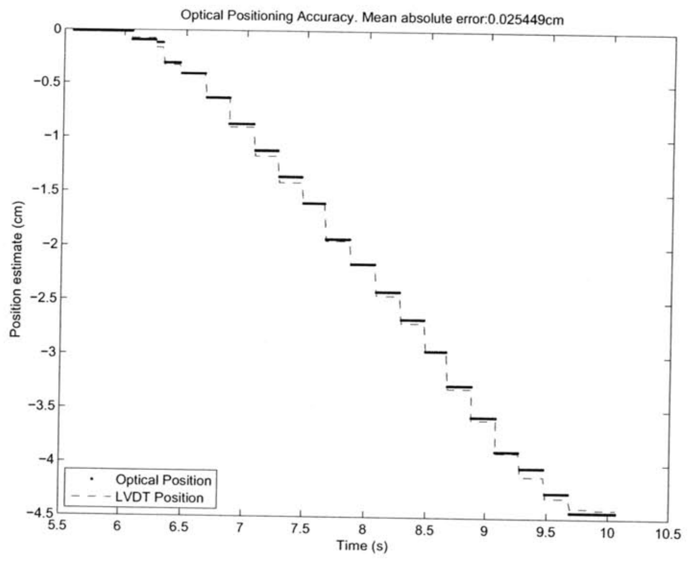


Figure 7-8: Optical Positioning Motion Accuracy

2400bps. The *FastScan* messaging functionality was used to increase the throughput of the optical sensor data, however, the messages communicating the laser position still required between 40ms and 50ms of transmission time. This, in addition to the 8ms of time to scan the photodiode, resulted in position measurements with up to 58ms of latency. Figure 7-9 shows that closed-loop control was accomplished, but that the robot frequently oscillated about its target position. The robot was commanded to several different positions, with consistent behavior. During this test, the target position tolerance was set to 0.05".

The oscillations exhibited in Figure 7-9 can be remedied by specifying looser position tolerances or by reducing the latency of position measurements. Figure 7-10 shows that oscillations can be avoided with a position tolerance of 0.1". A first attempt to reduce the latency of the position measurements removed the checksum information from the packets. This shortened each communication by 22ms, to obtain a latency of 36ms. Figure 7-11 shows that this allowed stable closed-loop control with 0.05" position tolerance.

Another attempt to reduce position measurement latency was made by increasing the communication baud rate from 2400bps to 19200bps. This prevented us from

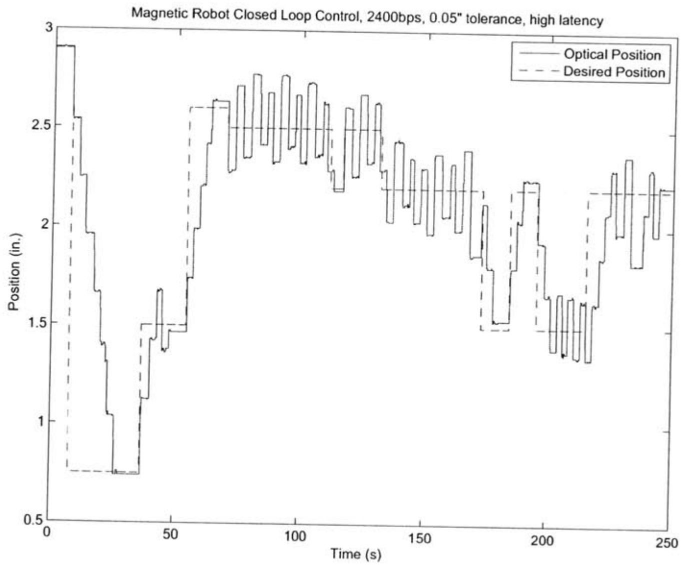


Figure 7-9: High latency in position data causes stable oscillation with 0.05" position tolerance.

using infrared communication for this test. Using the shortened, checksum-less message at this data rate resulted in 12ms of latency. Figure 7-12 shows that with the decreased latency, the 0.05" position tolerance is easily achievable. Additionally, a larger number of samples per second are obtained using the increased baud rate, which directly results in more iterations of the control loop and a smoother motion profile.

A final experiment was performed to characterize the effect of irregularities in the wingbox construction on accurate motion. This experiment used identical sensor configurations as used in Figure 7-12, but the robot was positioned over a seam in the wingbox construction. The minimum current required to move the robot was increased from 4.4A to 4.8A. Consequently, the robot's velocity increased, causing overshooting and oscillation. This is demonstrated in Figure 7-13.

The experimental results indicate that communication latency is the primary limiting factor in using the optical positioning system to achieve accurate closed-loop control of a magnetically actuated robot. The reference tolerance of 0.05" was achievable with both 2400bps and 19200bps, although the higher baud rate provided smoother

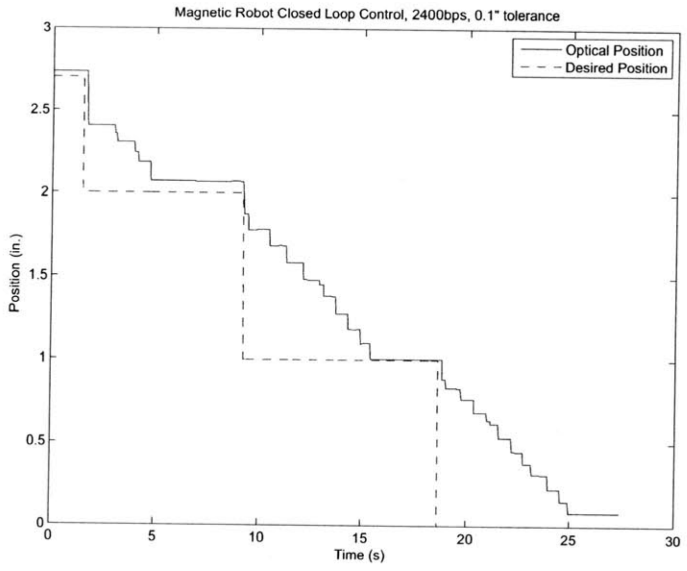


Figure 7-10: Looser tolerances eliminates oscillations. At 2400bps, full-length messages achieve 0.1" position tolerances.

and more reliable positioning.

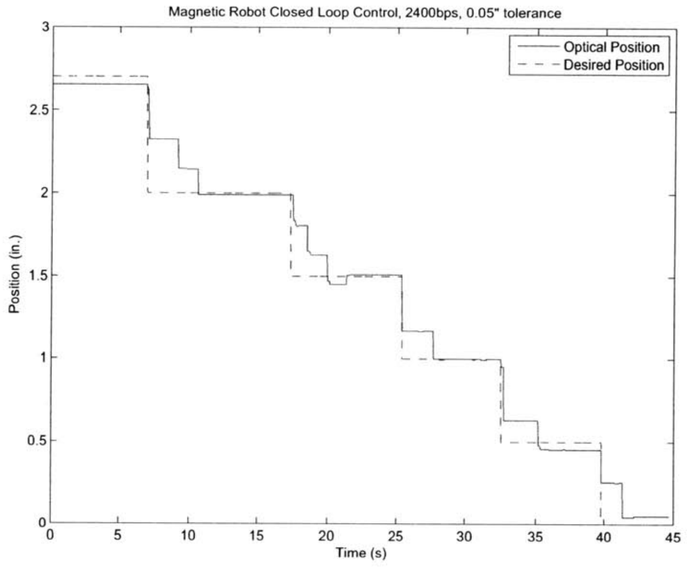


Figure 7-11: Closed loop control at 2400bps is stable for lower latency, checksum-less communication, with 0.05" position tolerance.

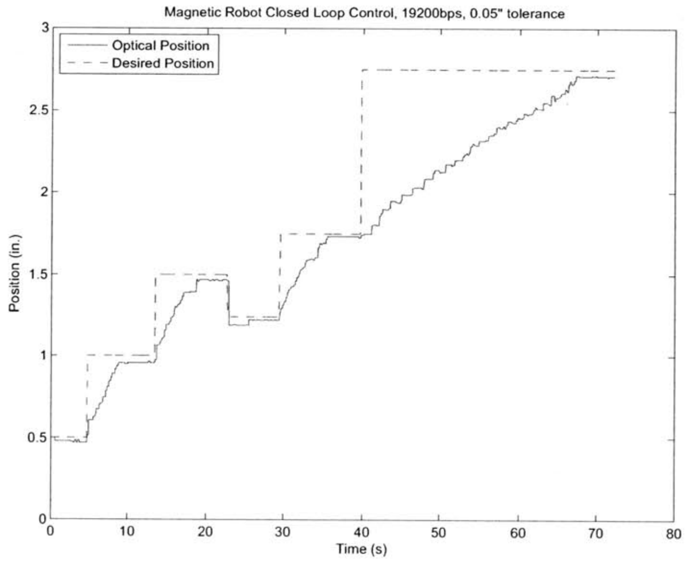


Figure 7-12: A higher baud rate of 19200bps reduces latency, providing smooth stable closed-loop control.

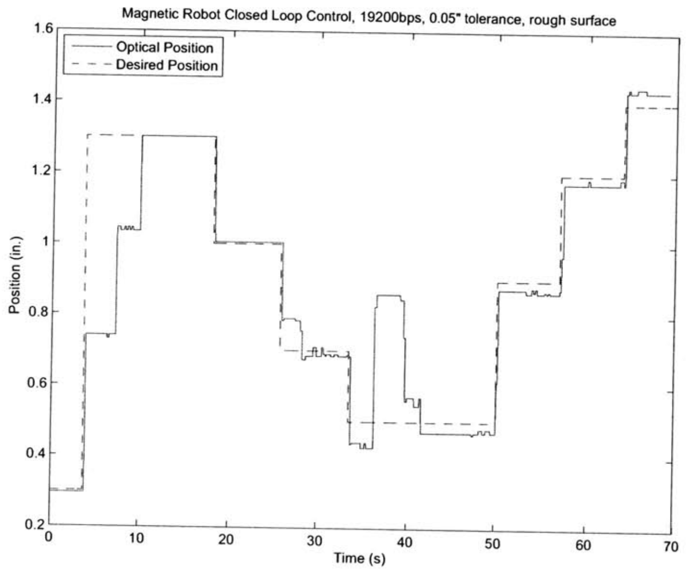


Figure 7-13: Rough surface requires higher drive current, producing overshooting.

7.4 Instrumented Gripper Reliability

The reliability of the instrumented gripper was quantified through over 1400 identification trials and 165 end-to-end tests of the Locate-and-Grasp algorithm.

7.4.1 Material identification testing

To test the gripper's ability to identify materials, an assembly material was placed within the field of view of the gripper at a distance between 15 and 30cm. A single identification trial consists of the gripper issuing a *Query* message to the component, and then listening for an ID message response. In the event that a message is received, but the message does not match its checksum, the gripper will issue one additional *Query* message. A success is recorded if an error-free ID message is received by the gripper after the first or second transmission.

7.4.2 $Locate - and - Grasp$ algorithm testing

The grasp testing evaluates the use of infrared communication to perform a line search for a desired component, align the gripper above the component, lower the manipulator and perform a successful grasp. A grasp is considered successful when the gripper closes completely around the assembly component.

For our experiments, it is assumed that the material's exact location is not known, but that it is required to be along an 4cm wide arc-shaped depot of fixed radius, rather than in a linear depot. This assumption was made due to the limited reach and accuracy of the robot's manipulator; sweeping the gripper in an arc by rotating the base of the manipulator was more practical for experiments than attempting to move the manipulator over a linear region. We assume that the assembly robot's knowledge of its position is sufficient for the robot to navigate to the parts depot. Within the parts depot, intelligent assembly struts and junctions may be placed. The components are not indexed into fixed positions, but their center point must be within the depot.

The reliability of the instrumented gripper was evaluated by performing a sequence of 165 select-and-grasp operations. A select-and-grasp operation consists of a radial search (sweeping the manipulator 180 degrees around its base axis) in search of a blue fastener, followed by completely enclosing the fastener within the closed gripper. A blue fastener is placed along the arc, at a position unknown to the robot. A successful select-and-grasp determines the fastener's position and securely grasps it.

Of the 165 select-and-grasp operations performed, 164 resulted in successful grasps, while one attempt swept over the fastener but failed to detect its presence (a "pass-over"). This failure mode results from a transmission bit error in the message transmitted from the gripper to the component: the component does not respond to corrupted messages, and so the gripper does not detect the component's presence.

Table 7.1 summarizes the results of the identification and grasping algorithm tests. In both sets of experiments, fewer than 1% of trials were failures. Failures result when communication is not established between the gripper and an assembly component within the gripper's field of view. Currently, the robot will make one attempt to resend a *Query* message if a corrupted response is received; however, the assembly components will not acknowledge a corrupted request. The introduction of a Negative-Acknowledge (NACK) message or an error-correction scheme for these cases may further improve the reliability of establishing communication.

Type	Attempted	Successful	Success Rate	Failure Modality
Identify Component	1447	1446	99.9%	random bit error
Identify, Align, and Grasp	165	164	99.3%	failed to identify part

Table 7.1: Summary of Experimental Identification and Grasping Results.

Chapter 8

Conclusions

The process of designing, building, programming, testing, and integrating the elements of the programmable matter assembly system, uncovered many problems and unexpected difficulties, and a few pleasant surprises. Solutions to some of these problems became core parts of the system. In some cases, problems that could not be satisfactorily resolved provided insight into previously unappreciated aspects of designing such systems. This chapter hopes to convey those lessons learned which may be widely applicable for designing programmable matter robotic systems.

8.1 Software Development Conclusions

Low-level programming respects few abstraction barriers. Attempts were made to test each sub-section of the embedded programs regularly throughout the software design cycle. However, as the embedded programs grew to use nearly all of the Atmega8's program space, these unit tests needed to be removed. Consequently, separate programs were run on the hardware to perform unit testing of different subsets of the software. Additionally, the Splint static code checking tool was used to enforce coding assumptions and provide stricter error checking than is provided by the AVR-GCC C compiler. Despite these precautions, several software problems were encountered. A common theme among these problems was the shared use of the AVR's hardware peripherals among multiple, conceptually independent software sub-sections. While

C is inherently an "unsafe" programming language, these problems were addressed by abstraction conventions that each hardware peripheral would be managed by a distinct software module. Splint annotations were employed in the embedded code to help enforce these conventions, but in many cases programmer discipline was the only actual enforcement. These experiences suggest that software development for embedded, programmable matter applications would benefit significantly from more capable embedded processors that could offer richer hardware abstractions, or even virtual machine functionality.

A related, but more specific lesson learned is the high importance of designing a communication abstraction that is robust and modular enough to readily support modifications to the communication protocol. More than once, as new functionality was added to the system, the communication needs or available hardware resources changed. Fortunately, the communication infrastructure, which separated packet-level syntax, communication timing, checksum construction, and message parsing into separate modules, was prepared for these changes.

8.2 Hardware Implementation Conclusions

A robotics professor once said wryly, "Robotics is the science of the battery and the connector", and so these hardware conclusions offer some advice relevant to the manufacturing of a distributed, programmable matter system.

A first observation is the importance of designing for manufacturing. In the end, over 150 individual PCBs were assembled. Some of the first versions required extensive, delicate assembly, and used fine-pitch connectors prone to gradual failure from the fatigue of ordinary use. Learning from these observations, power-on self-tests were incorporated into all modules. Interestingly, more than one PCB failed its first self-test on arrival from the fabrication and assembly facility.

A second observation does, in fact, center on the battery and the connector. A rule of thumb was discovered that states batteries should not need to be unplugged from their systems to be recharged, and that the runtime of one battery charge should be greater than or equal to the average interval between coffee breaks (to allow robots and researchers to maintain charge synchronization).

Appendix A

Electronics Schematics and PCB Layout

This appendix presents the schematics and PCB layout for each of the four electronics modules developed. For more information on each, please see Chapter 3.

A.1 Intelligent Beacon Node Schematic

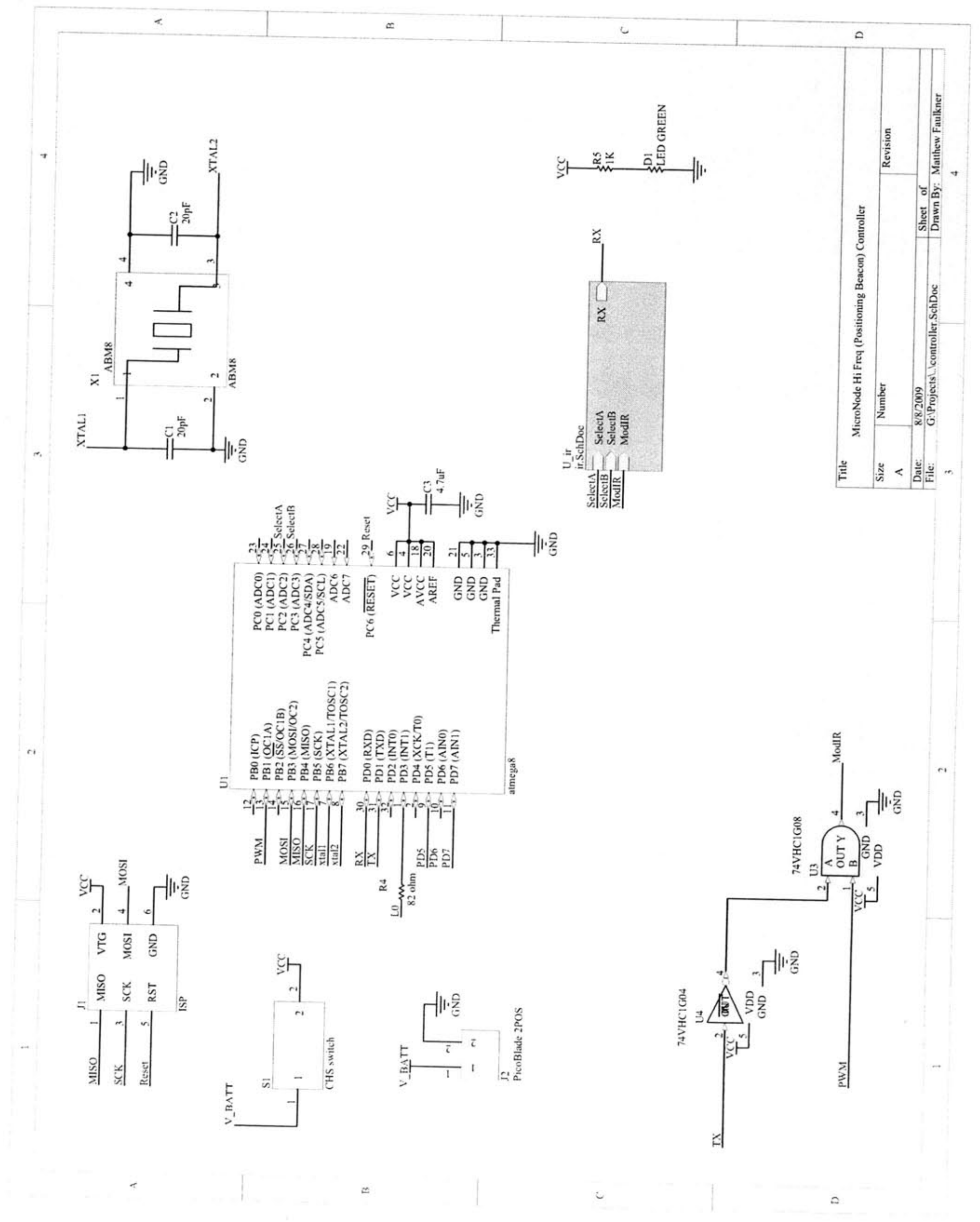

MicroNode Hi Freq (Positioning Beacon) Controller
Date: 8/8/2009
File: G:\Projects\..\controller.SchDoc
Drawn By: Matthew Faulkner
atmega8
ABM8
74VHC1G08
74VHC1G04
CHS switch
PicoBlade 2POS
LED GREEN
ir.SchDoc

Figure A-1: Intelligent Positioning Beacon Schematic, Sheet 1.

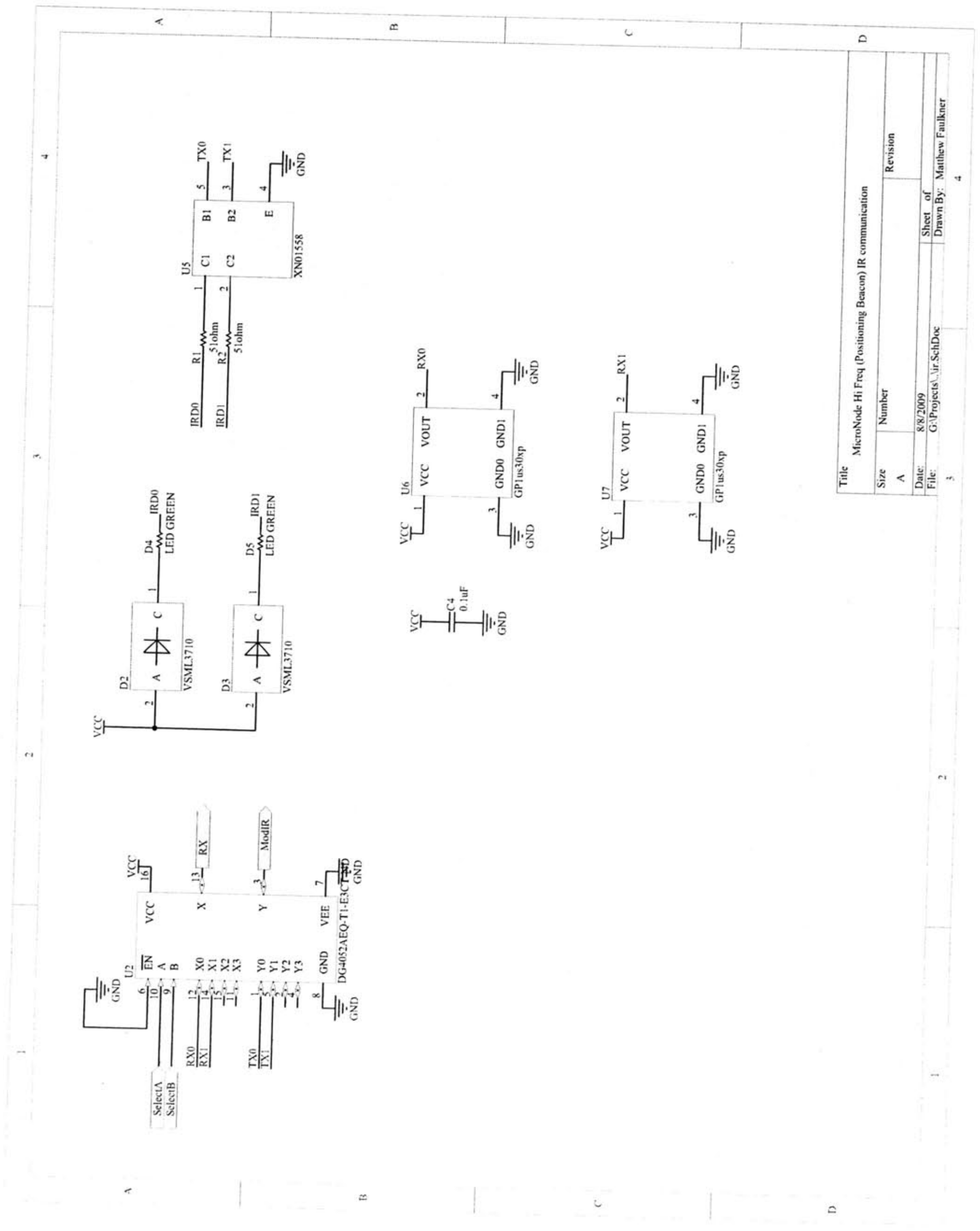


Figure A-2: Intelligent Positioning Beacon Schematic, Sheet 1.

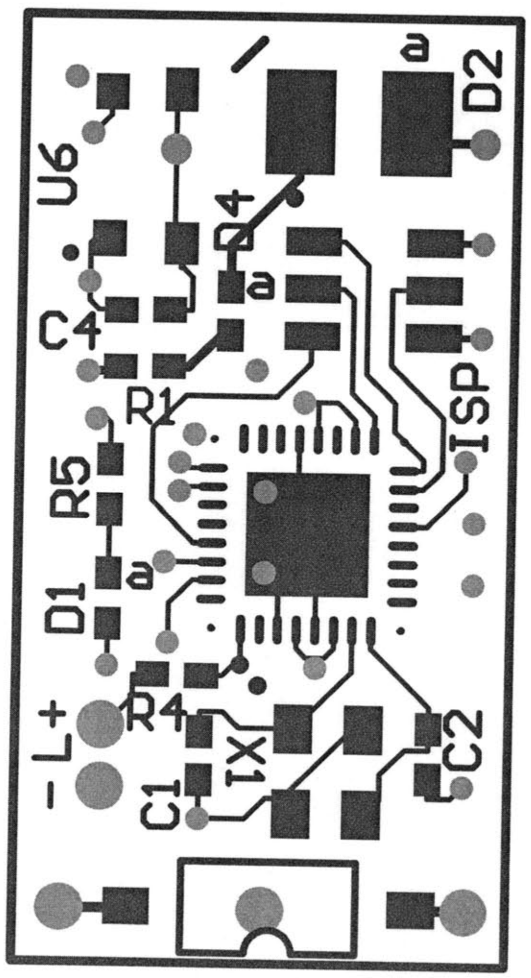


Figure A-3: Intelligent Postioning Beacon PCB layout, top.

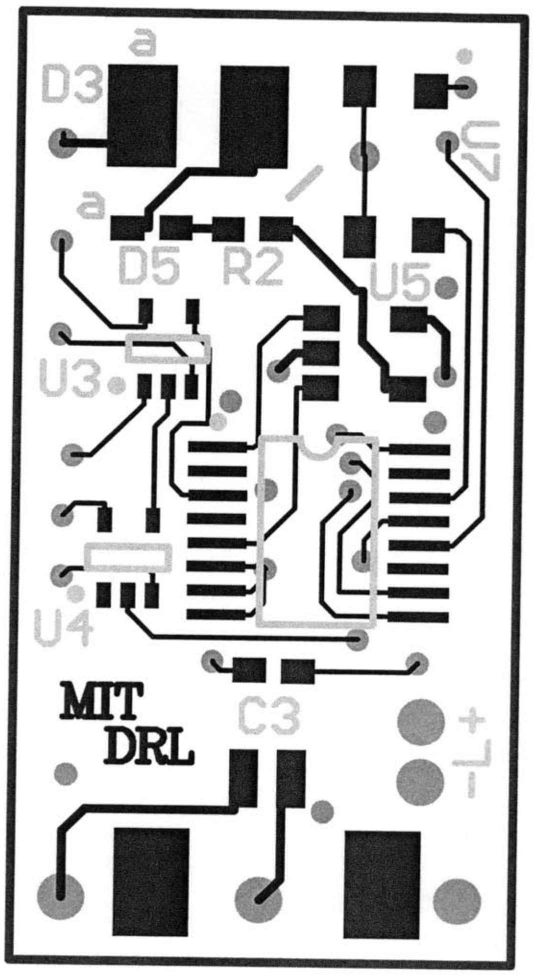


Figure A-4: Intelligent Positioning Beacon PCB layout, bottom.

A.2 Intelligent Assembly Node Schematic

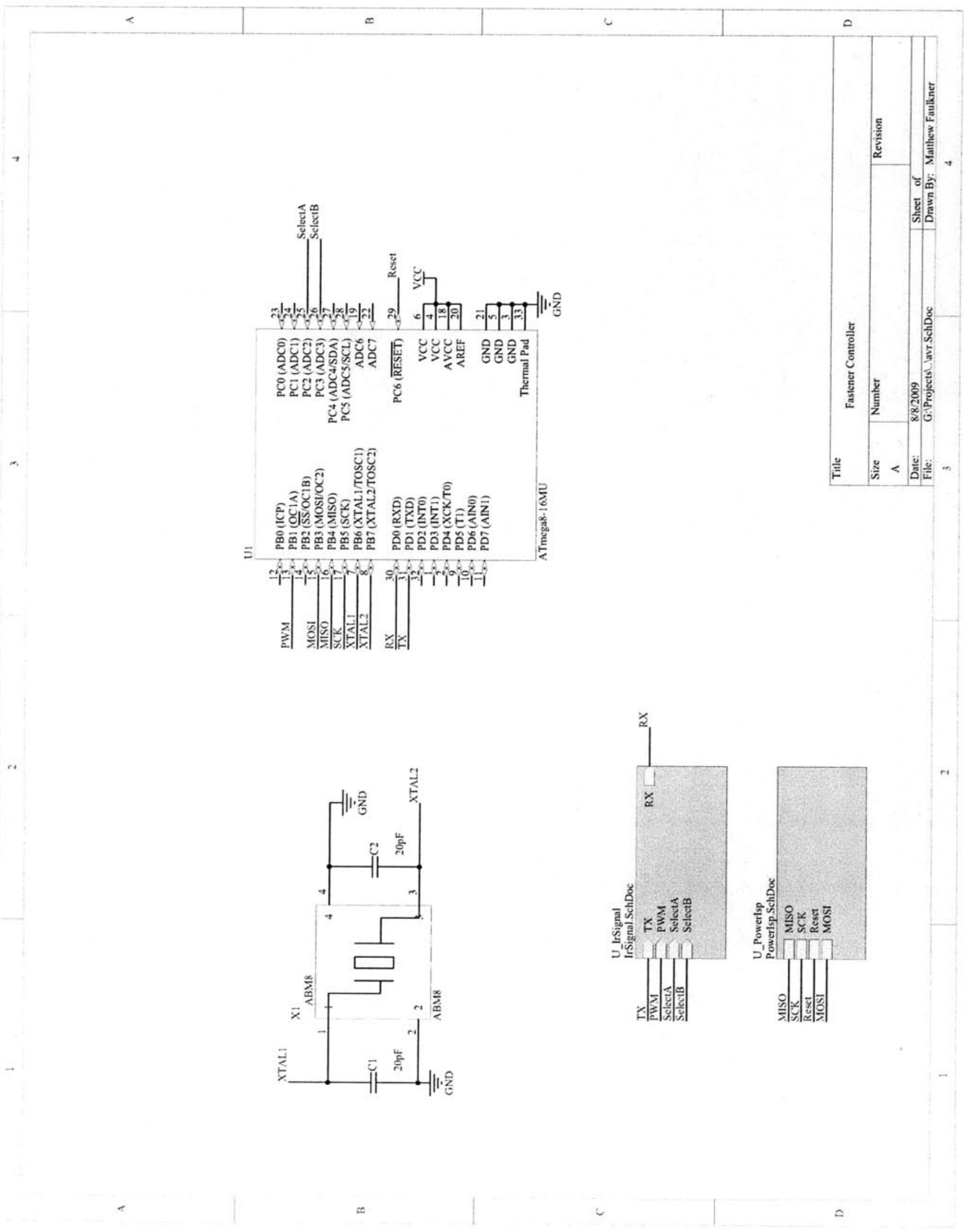


Figure A-5: Intelligent Assmbly Node Schematic, Sheet 1

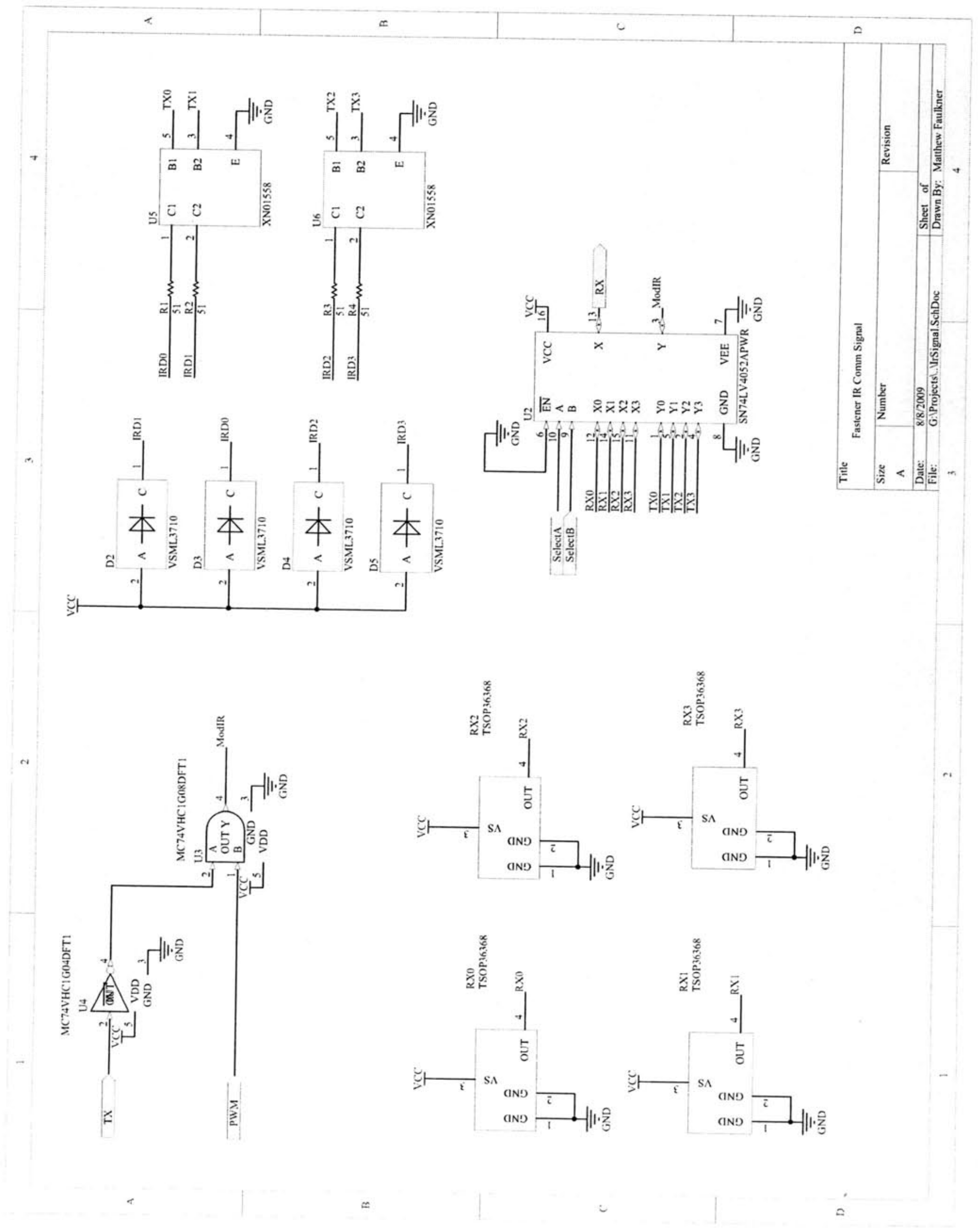


Figure A-6: Intelligent Assmbly Node Schematic, Sheet 2

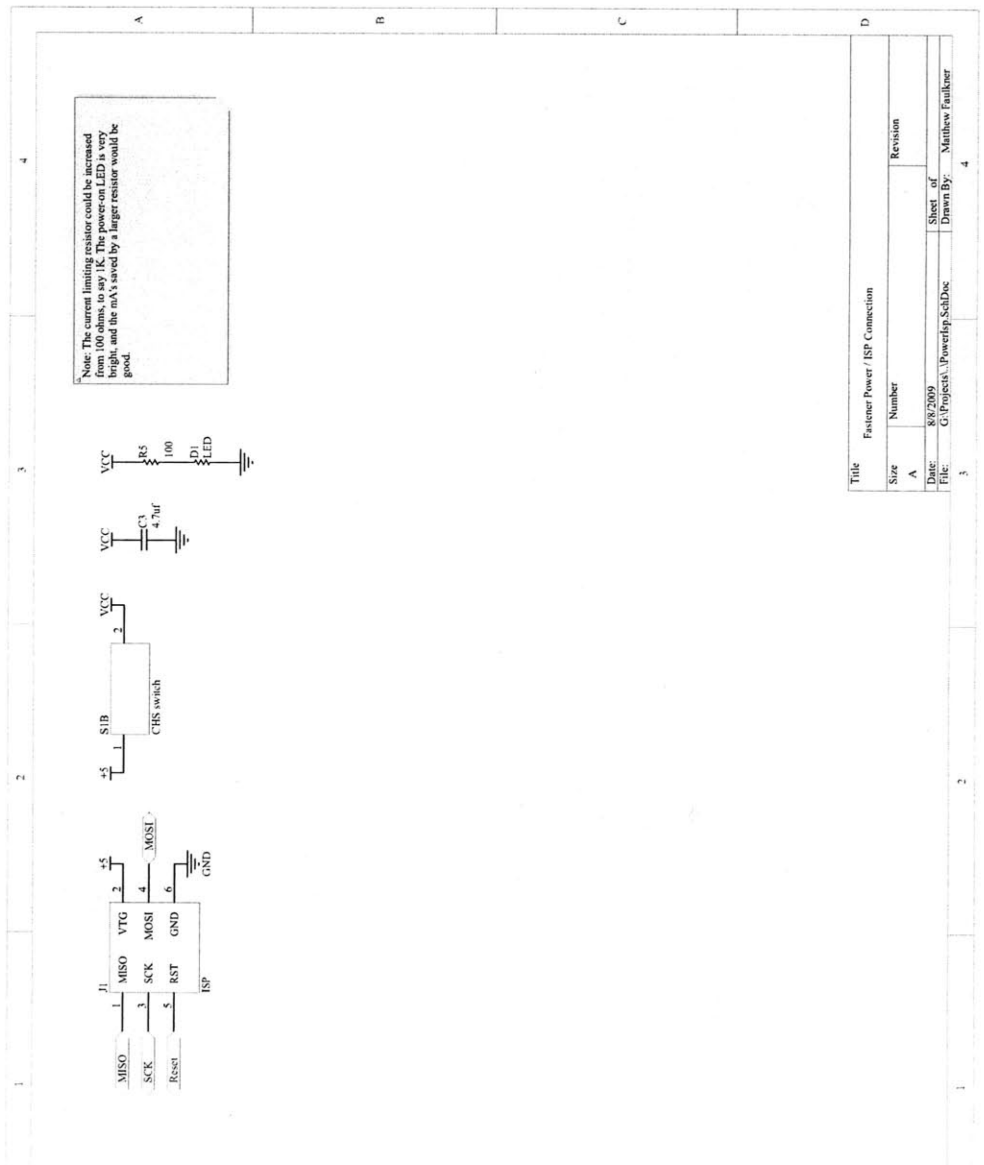

MISO
SCK
Reset
J1
MISO
SCK
RST
VTG
MOSI
GND
ISP
+5
MOSI
GND
+5
S1B
CHS switch
VCC
VCC
C3
4.7uf
VCC
R5
100
D1
LED
Note: The current limiting resistor could be increased from 100 ohms, to say 1K. The power-on LED is very bright, and the mA's saved by a larger resistor would be good.
Title
Fastener Power / ISP Connection
Size
A
Number
Revision
Date: 8/8/2009
Sheet of
File: G:\Projects\..\PowerIsp.SchDoc
Drawn By: Matthew Faulkner

Figure A-7: Intelligent Assmbly Node Schematic, Sheet 3

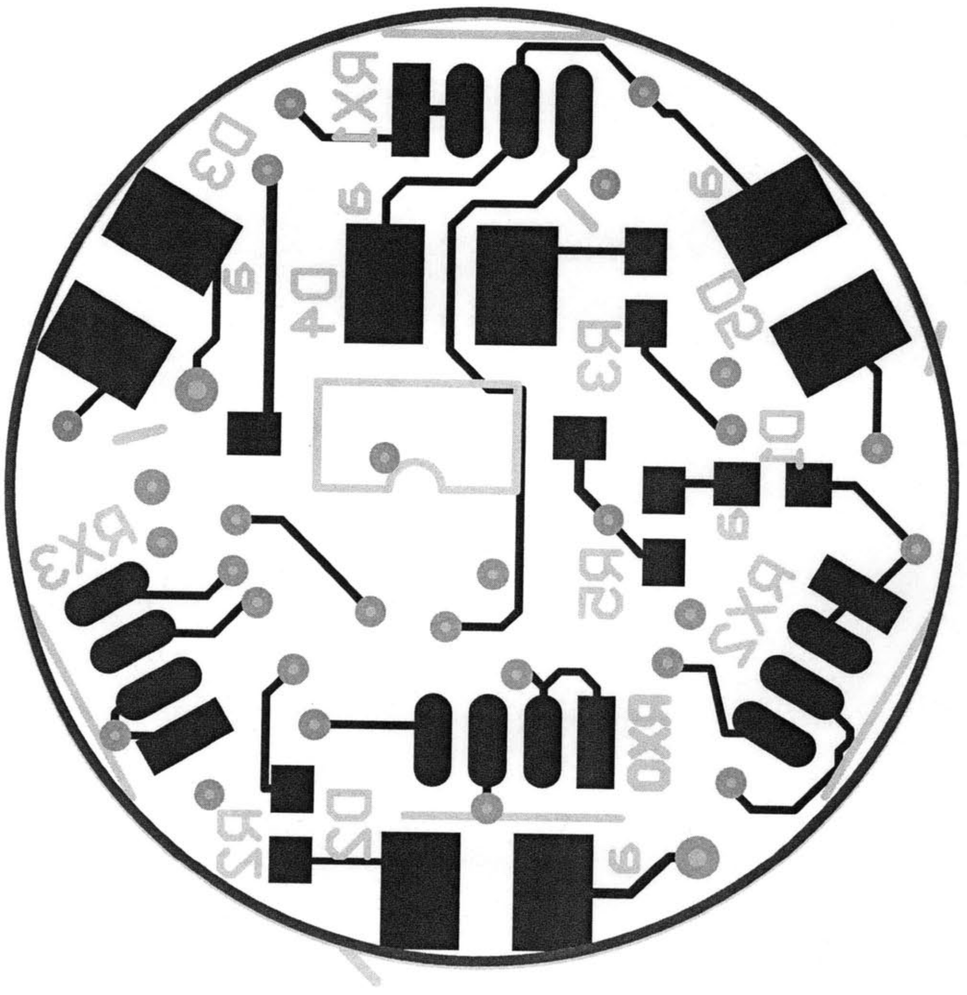


Figure A-8: Intelligent Assmbly Node PCB layout, top

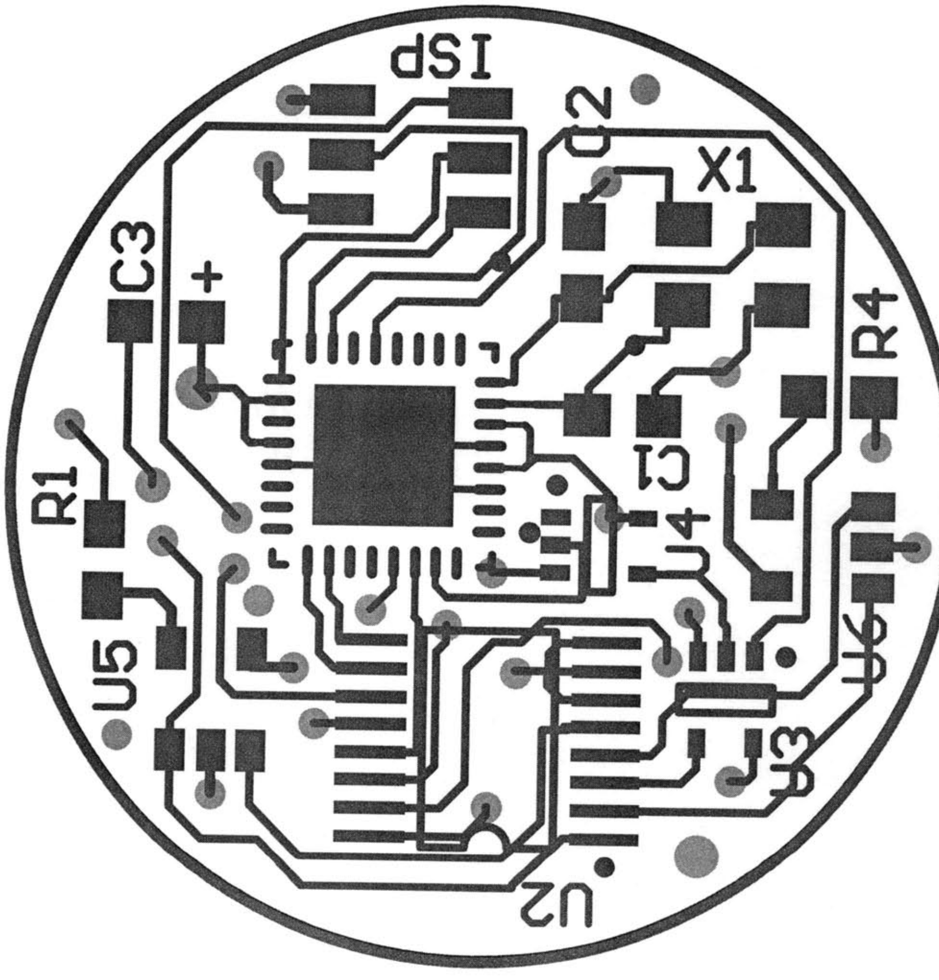


Figure A-9: Intelligent Assmbly Node layout, bottom

A.3 Interface Module Schematic

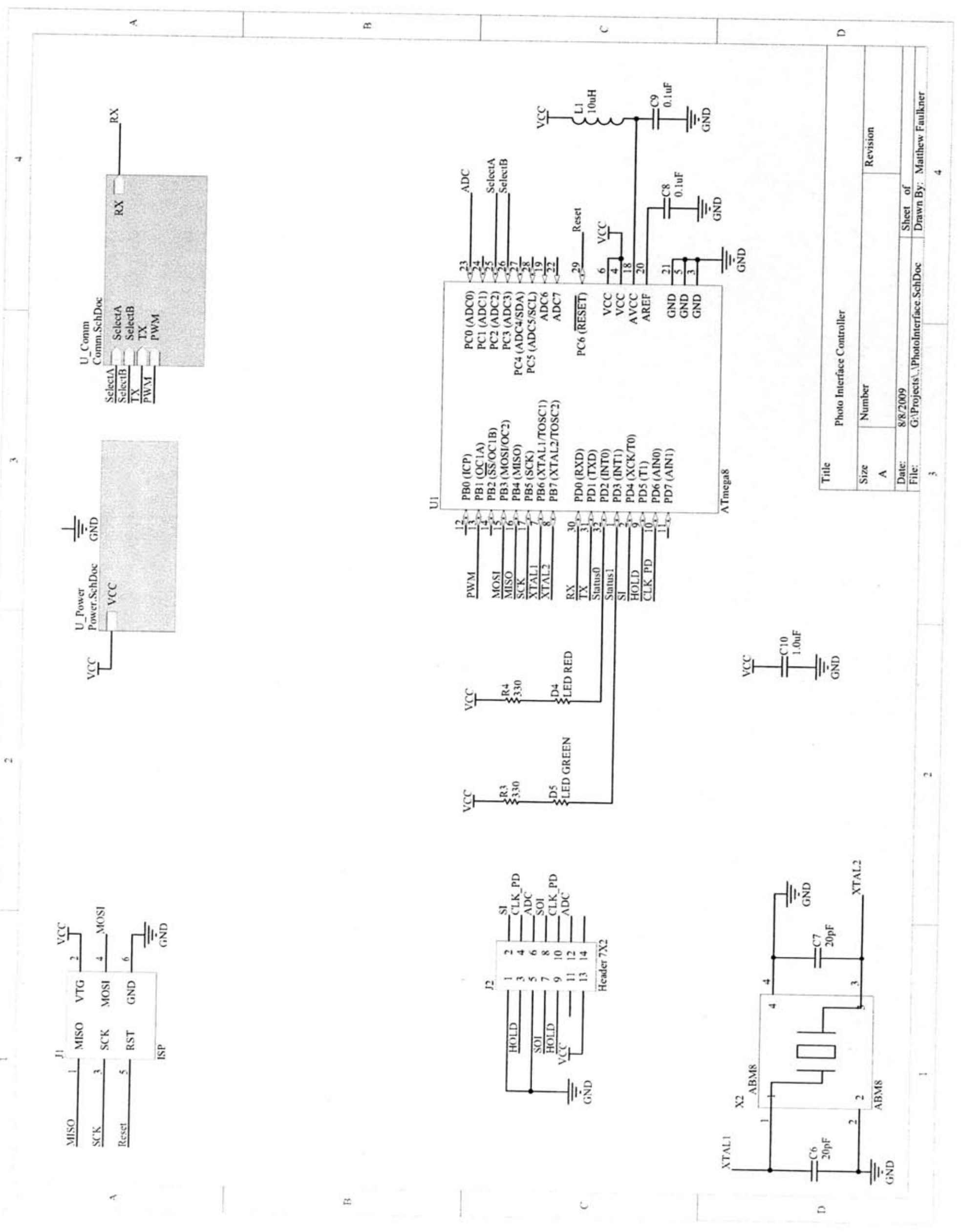


Photo Interface Controller
ATmega8
U_Comm
Comm.SchDoc
U_Power
Power.SchDoc
ISP
Header 7X2
ABM8
8/8/2009
G:\Projects\..\PhotoInterface.SchDoc
Drawn By: Matthew Faulkner

Figure A-10: Interface Module Schematic, Sheet 1

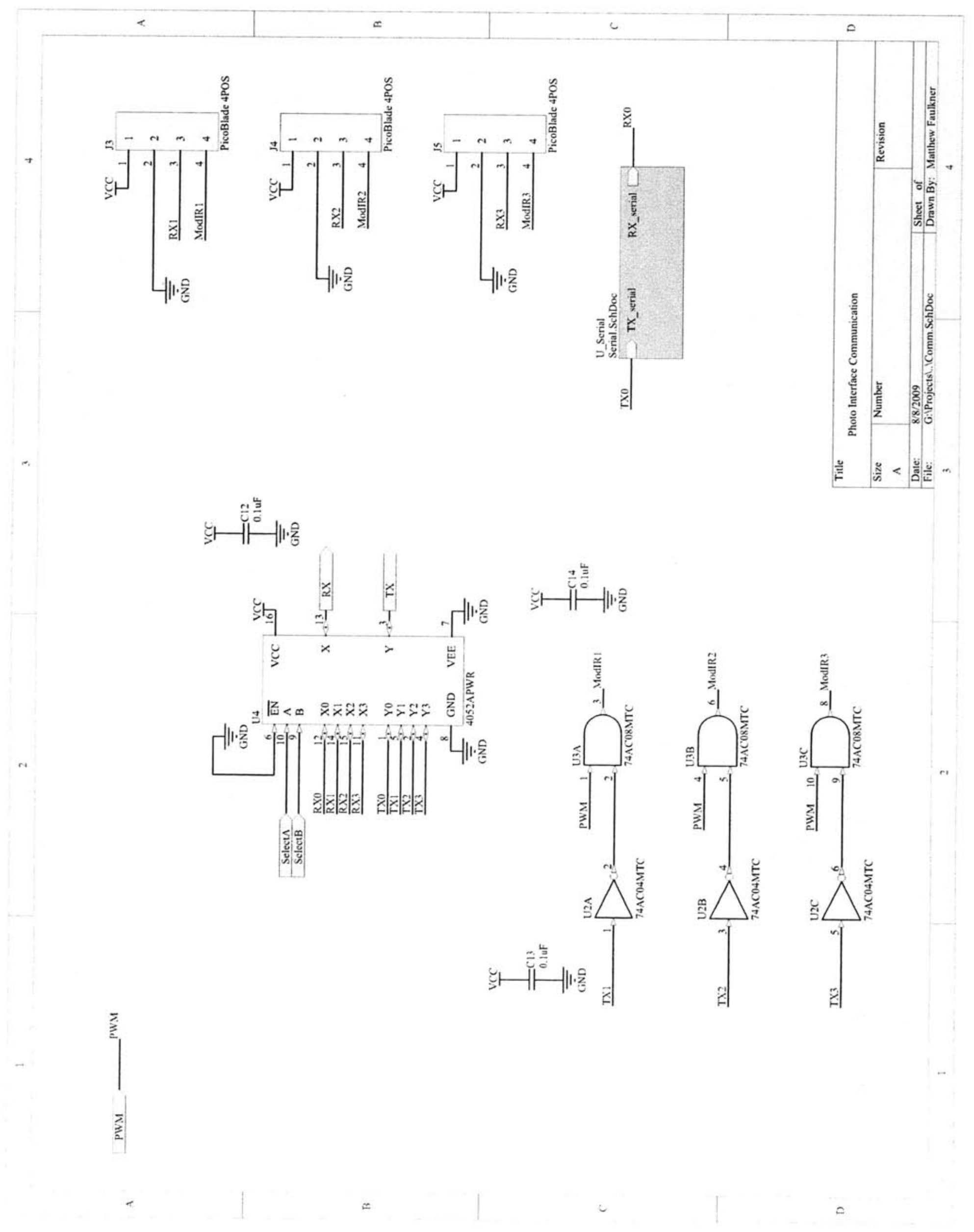


Figure A-11: Interface Module Schematic, Sheet 2

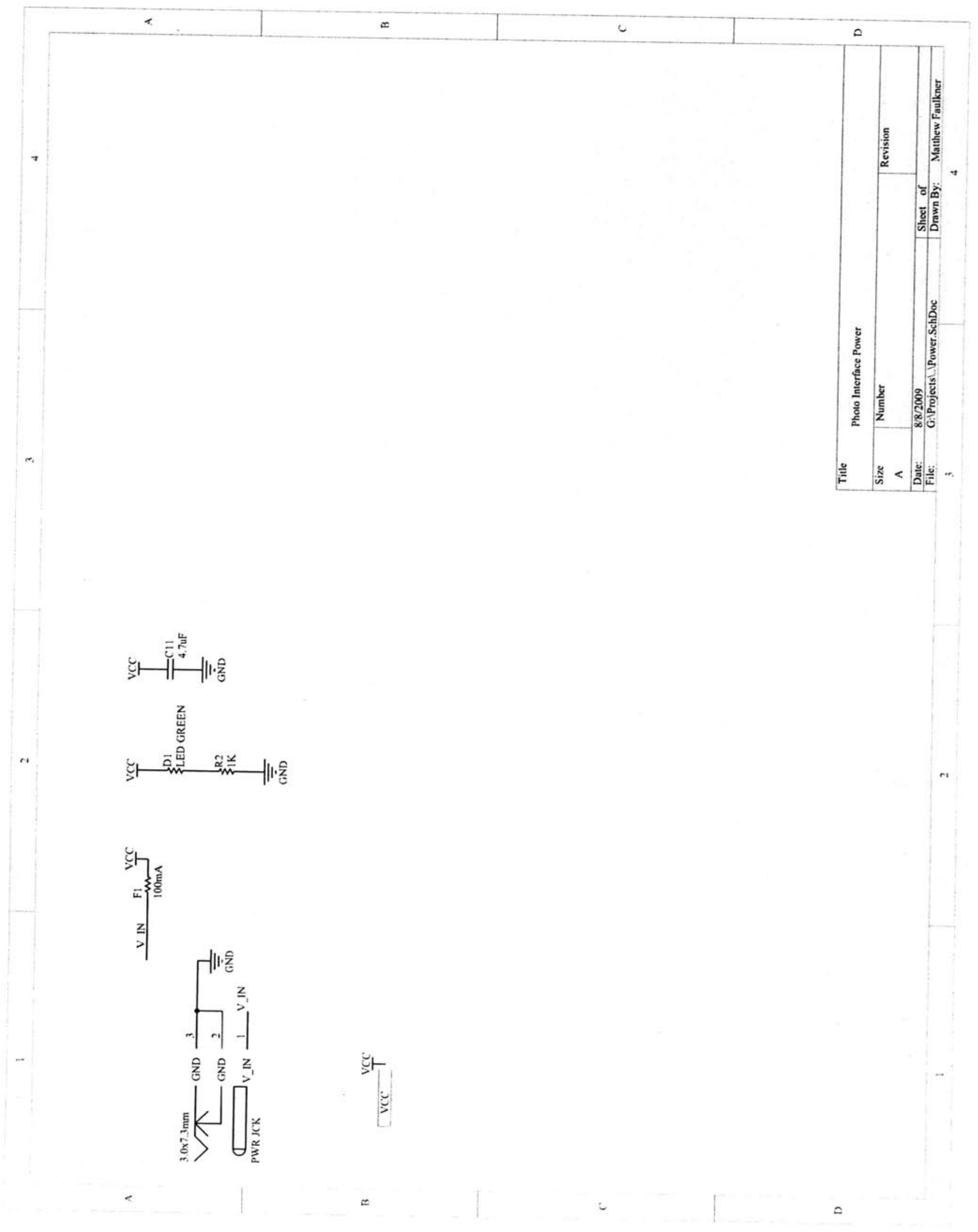


Figure A-12: Interface Module Schematic, Sheet 3

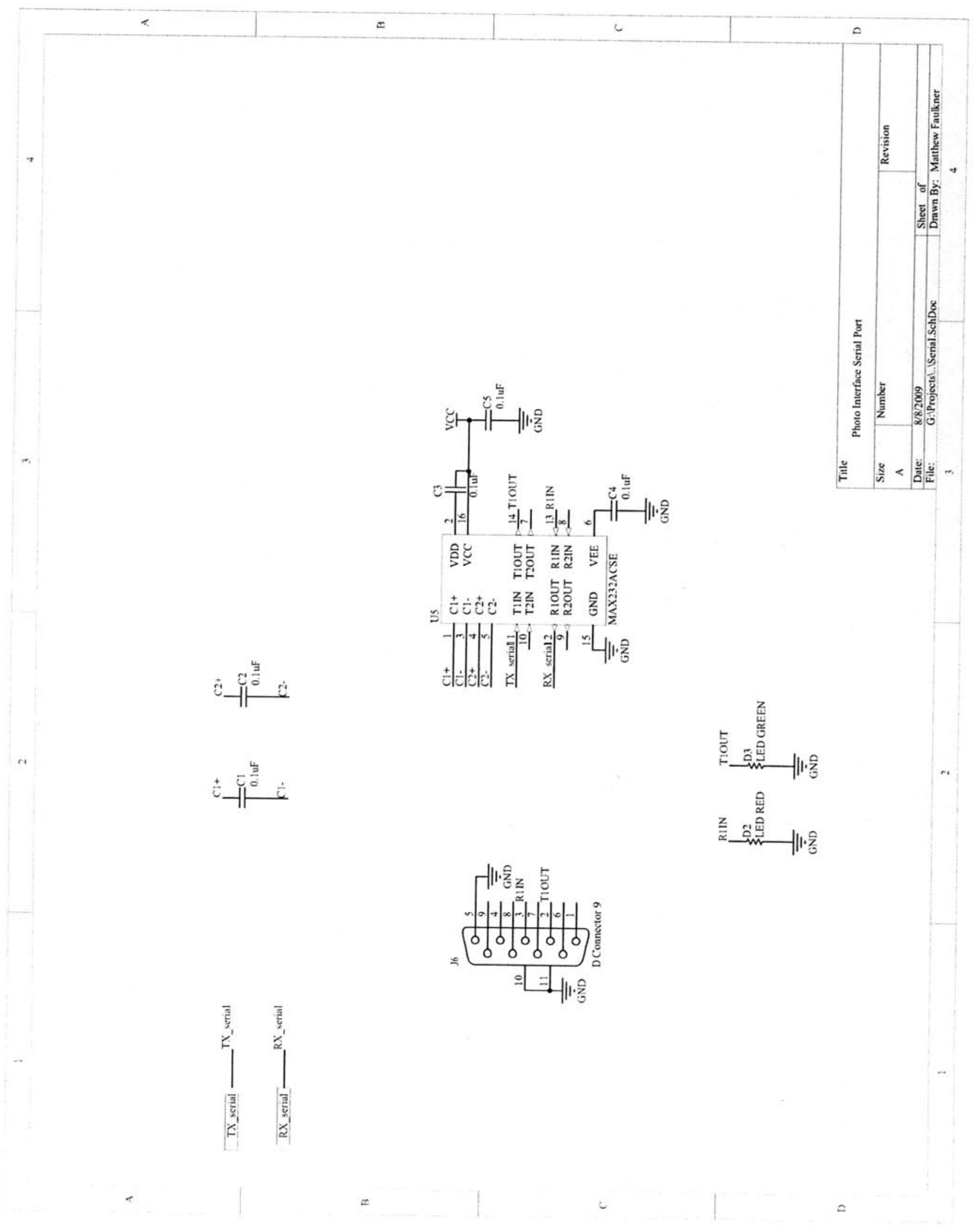


Figure A-13: Interface Module Schematic, Sheet 4

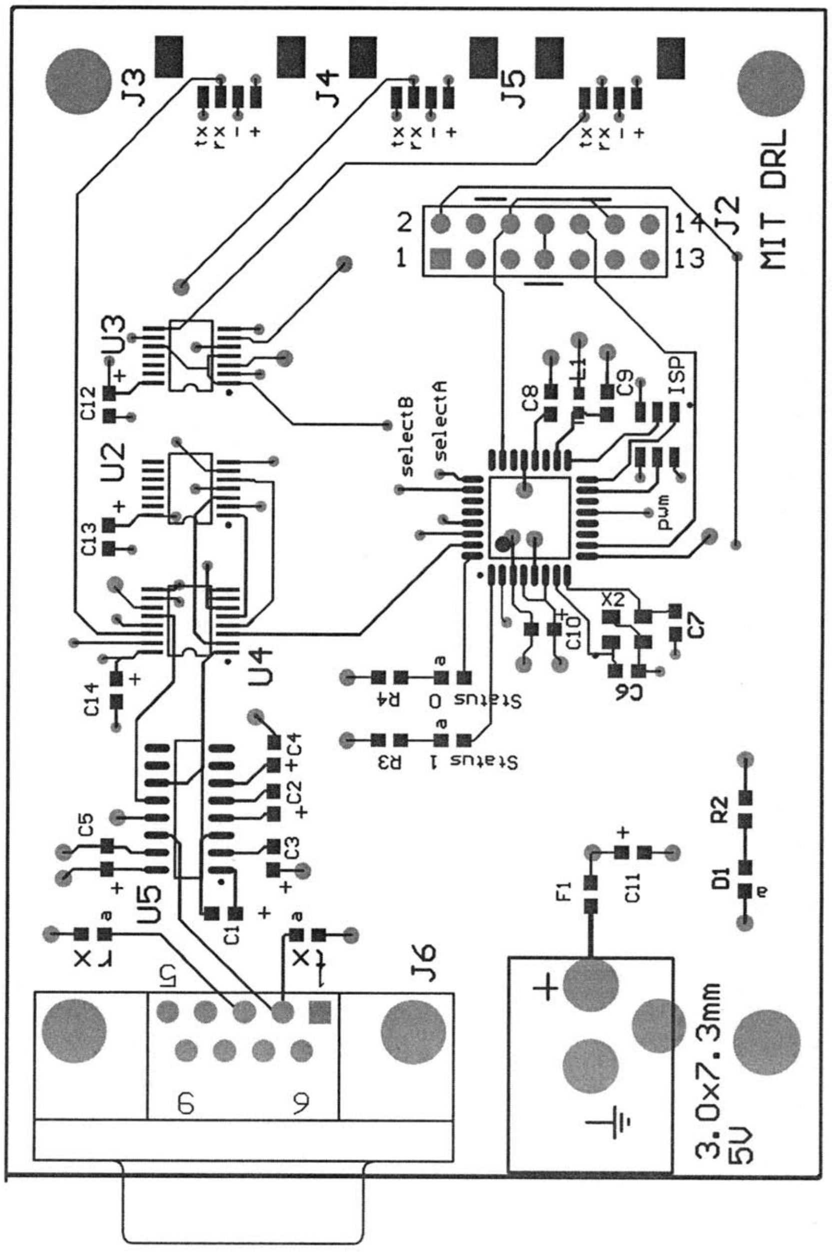


Figure A-14: Interface Module PCB layout, top.

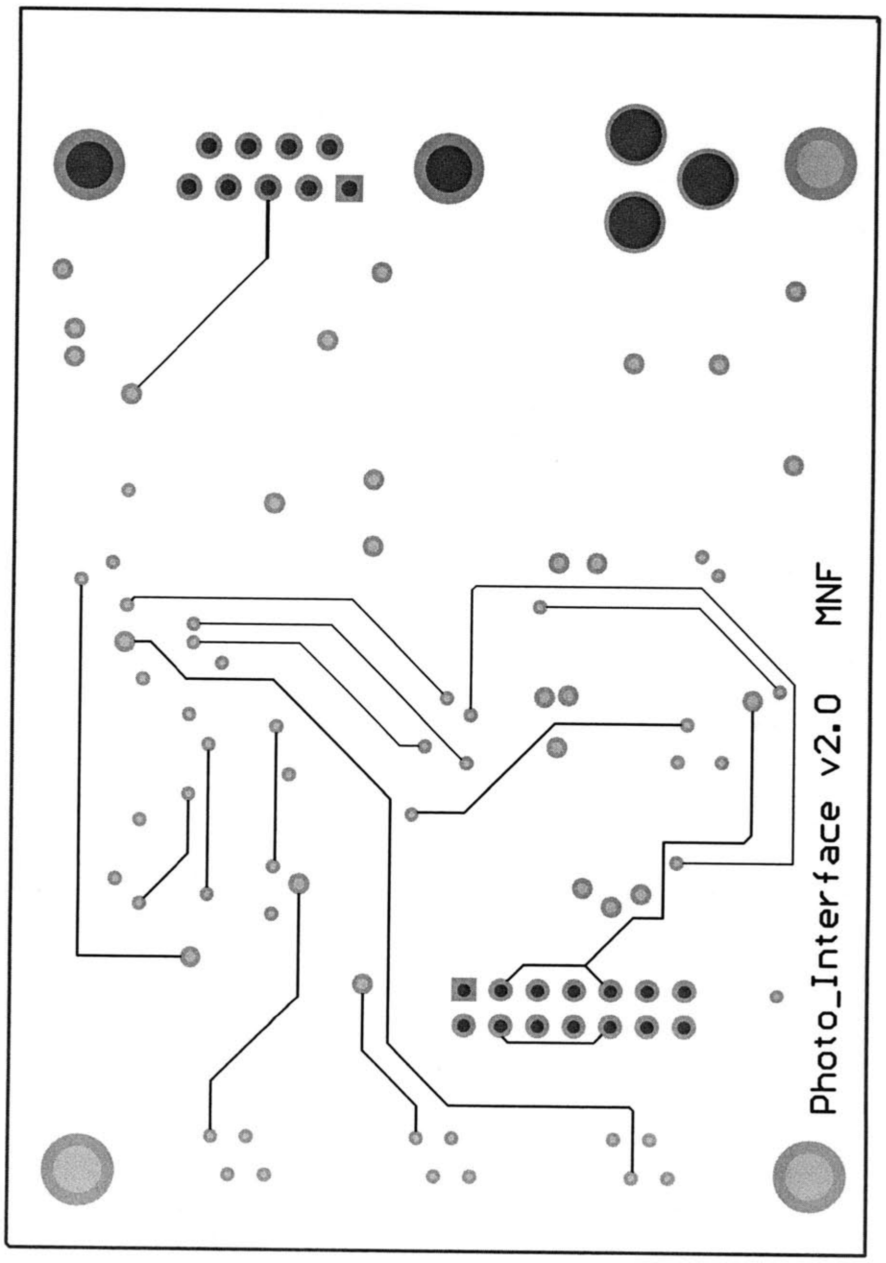


Figure A-15: Interface Module PCB layout, bottom

A.4 Receiver-Transmitter Board Schematic

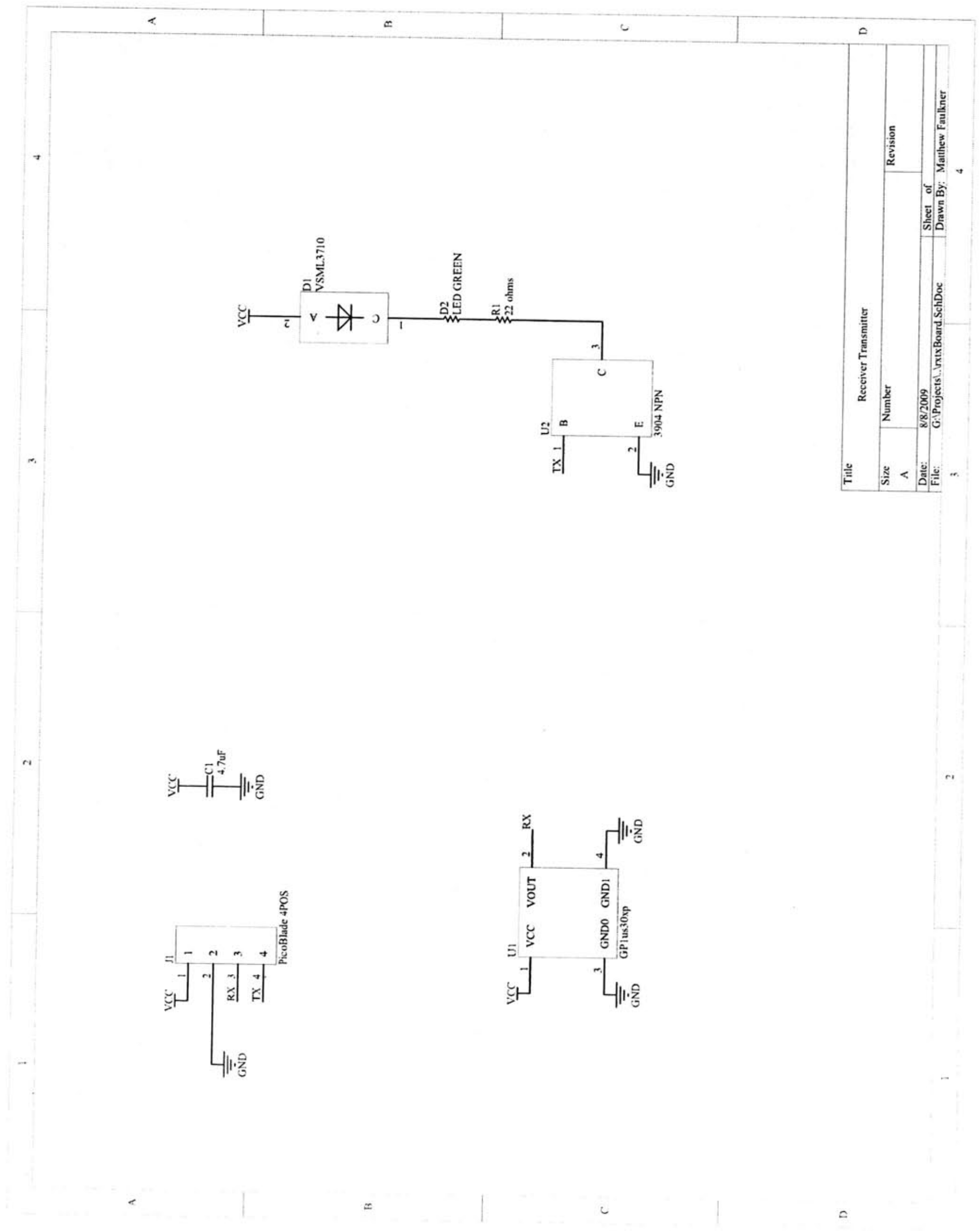


Figure A-16: Receiver Transmitter

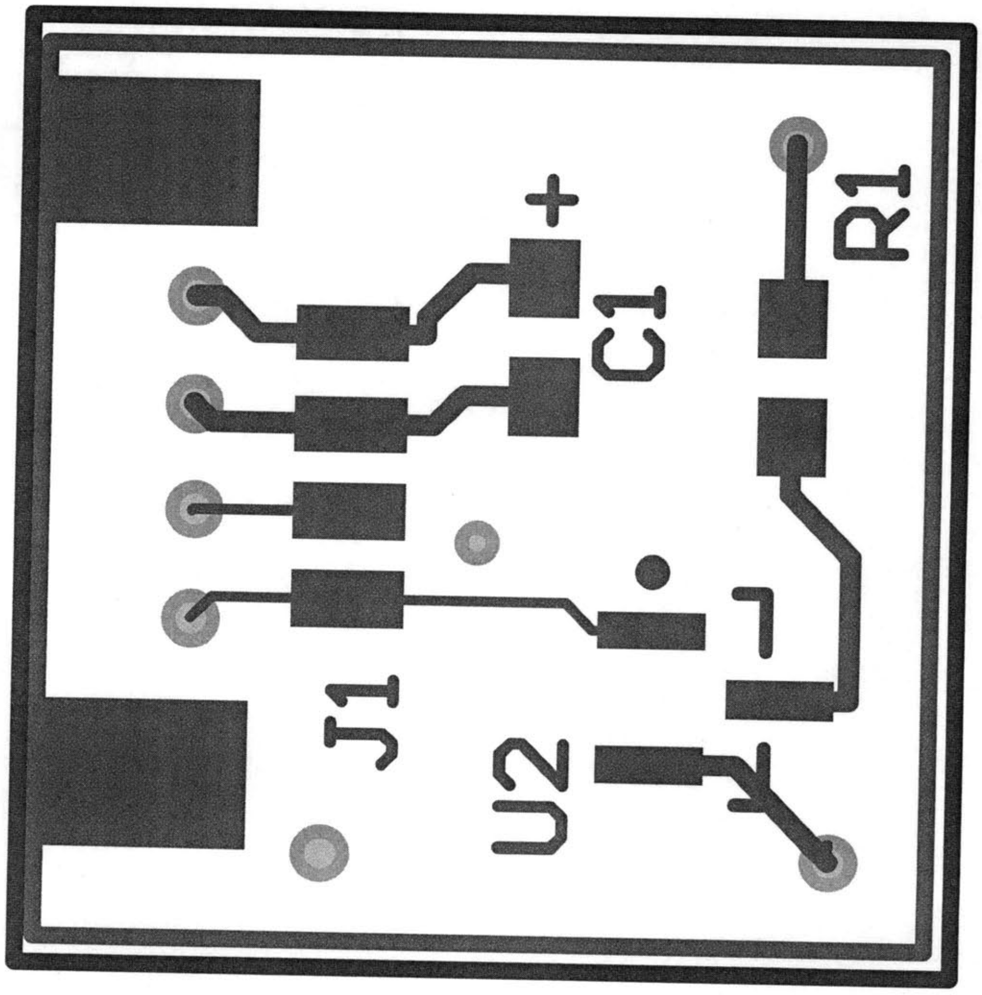


Figure A-17: Receiver Transmitter PCB layout, top.

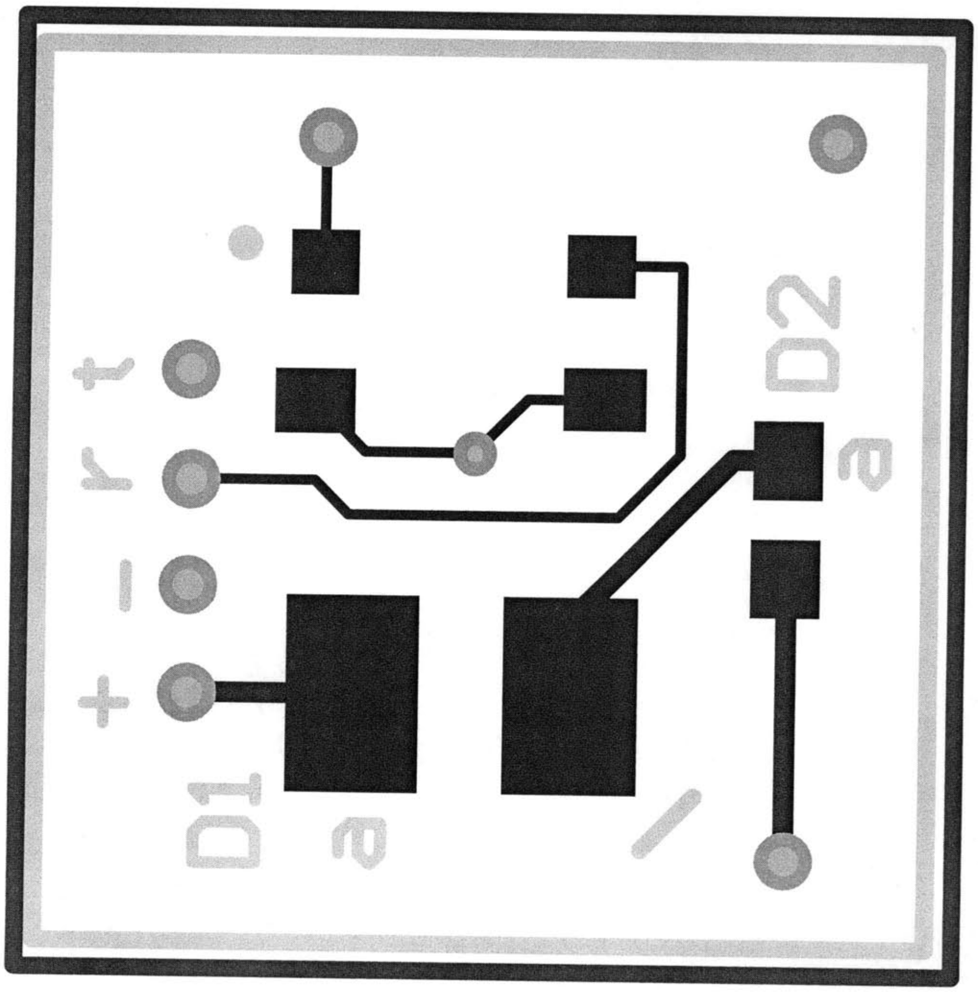


Figure A-18: Receiver Transmitter PCB layout, bottom

Appendix B

Embedded C Source Code

This appendix presents the programs written for each of the three electronic modules containing a microcontroller.

B.1 Intelligent Beacon Firmware

B.1.1 beacon.c

```
#include <avr/interrupt.h>
#include <string.h>
#include <util/delay.h>
#include "serialCommManager.h"
#include "beaconState.h"
#include "beaconMessage.h"
#include "usart.h"
#include "laser.h"

#define BAUDRATE 2400

/*
  These give more meaningful names to the serial channels than those
  given in serialCommManager.
*/
```

```
#define BEACON_IR_0 0 //side with AVR chip
#define BEACON_IR_1 1 //side with battery connector

static char msg[MAX_MESSAGE_LENGTH];

int main(void){

    cli();

    // initializations
    init_SerialCommManager(BAUDRATE);
    init_BeaconState();
    init_Laser();

    sei();

    // proof of life
    transmitMessage("Beacon.", 0);
    transmitMessage("Beacon.", 1);

    for(int i = 0; i <3; ++i)
    {
        //toggle laser
        setLaser(1);
        _delay_ms(250);
        setLaser(0);
        _delay_ms(250);
    }

    setLaser(1);

    for(;;){
        //alternate listening on the two channels.
        if( receiveMessage(msg, BEACON_IR_0) == 1)
        {
            handleReceivedMessage(msg, BEACON_IR_0);
```

```
                }
                if( receiveMessage(msg, BEACON_IR_1) == 1)
                {
                        handleReceivedMessage(msg, BEACON_IR_1);
                }
        }
}
```

B.1.2 beaconState.c

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "beaconState.h"

char beaconString[BEACON_STRING_FIELD_LENGTH+1]; //extra 1 for null terminating character


/*
  Internal beacon state.
*/
struct{
        char *const stringPtr;
        int identificationNumber;
} beaconState =
{
        .stringPtr = &beaconString[0],
        .identificationNumber = 6
};

void init_BeaconState(void){
        setBeaconString("default string");
        //seed the random number generator uniquely for each part.
        srand(beaconState.identificationNumber);
}

/* ---- Accessors ------ -*/

void setBeaconString(/*in */const char *src)
{
        strncpy(beaconState.stringPtr, src, BEACON_STRING_FIELD_LENGTH);
}

void getBeaconString(char * dest)
{
        strncpy(dest, beaconState.stringPtr, BEACON_STRING_FIELD_LENGTH);
}

int getBeaconIdentificationNumber()
{
        return beaconState.identificationNumber;
}

void setBeaconIdentificationNumber(int n)
{
        beaconState.identificationNumber = n;
}
```

B.1.3 beaconMessage.c

```
/**
 * beaconMessage.c
 * Message handling routines for beacon.
 *
 * Q:n - return the beacon's internal state String
 * SET:n,contents - set the beacon's internal state String
 * ON:n - activate laser
 * OFF:n - deactivate laser
 * POWER:n,m
 *
 * n is the address ID number. 255 is the broadcast ID
 *
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <util/delay_basic.h>
#include "beaconMessage.h"
#include "beaconState.h"
#include "serialCommManager.h"
#include "usart.h"
#include "utils.h"
#include "timer1.h" // for IR duty cycle control
#include "laser.h"

#define BROADCAST_ID 255

static char tempBeaconString[BEACON_STRING_FIELD_LENGTH + 1];
static char tempMessageString[MAX_MESSAGE_LENGTH+1];

/* --- Forward declarations ----- */
static void handleQuery(const char * msg, int channel);
static void handleSetMessage(const char * msg, int channel);
static void handleActivate(const char * msg, int channel);
static void handleDeactivate(const char * msg, int channel);
static void handleForward(const char * msg, int channel);

void handleReceivedMessage(const char* msg, int channel)
{
        if( stringStartsWith(msg, "Q:") )
        {
                handleQuery(msg, channel);
        }
        else if ( stringStartsWith(msg, "SET:") )
        {
                handleSetMessage(msg, channel);
        }
        else if (stringStartsWith(msg, "ON:"))
        {
                handleActivate(msg, channel);
        }
        else if (stringStartsWith(msg, "OFF:"))
        {
                handleDeactivate(msg, channel);
        }
        else if (stringStartsWith(msg, "FW"))
        {
                handleForward(msg, channel);
        }
        else
        {
                /* invalid message. Could broadcast a nack. */
        }
}

/* ———— Private Functions ———— */

/**
 *
 * msg - the received ID request message
 * channel - the channel msg was received through
 *
 */
static void handleQuery(const char * msg, int channel)
{
        int toField;
        int n = sscanf(msg, "Q:%d", &toField);
        if(n==1)
        {
                int id = getBeaconIdentificationNumber();
                if((toField == id) || (toField == BROADCAST_ID))
                {
                        getBeaconString(tempBeaconString);
                        snprintf(tempMessageString, MAX_MESSAGE_LENGTH, "ID,%d:%s",
                                getBeaconIdentificationNumber(),tempBeaconString);
                        transmitMessage(tempMessageString, channel);
                }
                else
                {
                        /* addressed to someone else. */
                }
        }
        else
        {
                /* message does not match expected syntax */
        }
}

/**
 *
 * Sets the internal beaconState string. The first
 * BEACON_STRING_FIELD_LENGTH characters of the msg contents are
 * copied, or up to the first whitespace.
 *
 * If the interface board is used to relay messages from laptop to
 * beacons, check that the timeout and max message length of the
 * interface board are long enough for the message contents.
 *
 * msg The received SET message
 * channel The channel on which the message was received.
 */
static void handleSetMessage(const char * msg, int channel)
{
        int toField;
        int n = sscanf(msg, "SET:%d,%s", &toField, tempBeaconString);
        /* check that both toField and string parsed from message: */
        if(n == 2)
        {
                int id = getBeaconIdentificationNumber();

                if((toField == id) || (toField == BROADCAST_ID))
                {
                        setBeaconString(tempBeaconString);
                        transmitMessage("ACK", channel);
                }
                else
                {
                        /* addressed to someone else. */
                }
        }
        else
        {
                /* message does not match expected syntax */
        }
}

static void handleActivate(const char * msg, int channel)
{
        int toField;
        int n = sscanf(msg, "ON:%d", &toField);
        if(n==1)
        {
                int id = getBeaconIdentificationNumber();
                if((toField == id) || (toField == BROADCAST_ID))
                {
                        setLaser(1);
                }
                else
                {
                        /*addressed to someone else. */
                }
        }
        else
        {
                /* message does not match expected syntax */
        }
}

static void handleDeactivate(const char * msg, int channel)
{
        int toField;
        int n = sscanf(msg, "OFF:%d", &toField);
        if(n==1)
        {
                int id = getBeaconIdentificationNumber();

                if((toField == id) || (toField == BROADCAST_ID))
                {
                        setLaser(0);
                }
                else
                {
                        /* addressed to someone else. */
                }
        }
        else
        {
            /* message does not match expected syntax */
        }
}

/**
 *
 *
 */
static void handleForward(const char * msg, int channel)
{
        int timeToLive = 0;
        char msgToForward[MAX_MESSAGE_LENGTH+1];
        int n = sscanf(msg, "FW,%d#%s", &timeToLive, msgToForward);
        if(n==2)
        {
                handleReceivedMessage(msgToForward, channel);
                if(timeToLive > 0)
                {
                        timeToLive--;
                        snprintf(tempMessageString, MAX_MESSAGE_LENGTH,
                                "FW,%d#%s", timeToLive, msgToForward);
                        /*
                          Delays for a random time between 0 and 1
                          second. Since the delay interval is not
                          known at compile-time, the standard
                          _dealy_ms function cannot be used. Instead,
                          the basic _delay_loop_2 is used, which
                          executes 4 CPU cycles per iteration. At
                          14.7456MHZ, each iteration lasts for .271
                          millionths of a second, so at 65536
                          iterations, the delay loop can take 0.018s,
                          or about 1/56 of a second.
                        */
                        uint16_t randomDelay = rand();
                        int i;
                        for(i = 0; i < 56; ++i)
            {
                /*
                  can't use _delay_ms if the time
                  interval isn't known at
                  compile-time.
                */
                _delay_loop_2(randomDelay);
            }

            transmitMessage(tempMessageString, 0);
            transmitMessage(tempMessageString, 1);
        }
        else
        {
            /* message dies here. Don't forward. */
        }
    }
}
```

B.1.4 laser.c

```
#include "laser.h"
#include "util_defines.h"
#include <avr/io.h>

#define LASER PD3

void init_Laser()
{
	SETBIT(DDRD, LASER); //configure PD3 as a digital output
	setLaser(0); //laser is off by default
}

void setLaser(int i)
{
	if(i == 0)
	{
		/* turn laser off */
		CLEARBIT(PORTD, LASER);
	}
	else
	{
		/* turn laser on */
		SETBIT(PORTD, LASER);
	}
}
```

B.2 Intelligent Assembly Component Firmware

B.2.1 fastener.c

```
#include <avr/interrupt.h>
#include <avr/io.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <util/delay.h>
#include "serialCommManager.h"
#include "fastenerState.h"
#include "fastenerMessage.h"
#include "usart.h"

#define BAUDRATE 2400

static char msg[MAX_MESSAGE_LENGTH];

int main(void)
{
        cli();
        /* ------ initializations ------------- */

        init_SerialCommManager(BAUDRATE);
        init_FastenerState();

        sei();

        /* proof of life */
        transmitMessage("Fastener.", 0);
        transmitMessage("Fastener.", 1);
        transmitMessage("Fastener.", 2);
        transmitMessage("Fastener.", 3);
```

```
    for(;;){
        if( receiveMessage(msg, 0) == 1)
        {
            /*
              listen on center receiver
            */
            handleReceivedMessage(msg, CHANNEL_SERIAL_PORT);
        }
    }
}
```

B.2.2 fastenerMessage.c

```
/**
 * fastenerMessage.c
 * Message handling routines for fastener.
 *
 * Q:n - return the beacon's internal state String
 * SET:n,contents - set the beacon's internal state String
 * POWER:n,m
 *
 * n is the address ID number. 255 is the broadcast ID
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "fastenerMessage.h"
#include "fastenerState.h"
#include "serialCommManager.h"
#include "usart.h"
#include "utils.h"
#include "timer1.h" // for IR duty cycle control

#define BROADCAST_ID 255

// Plus 1 for the null terminating character:
static char tempFastenerString[FASTENER_STRING_FIELD_LENGTH + 1];
static char tempMessageString[MAX_MESSAGE_LENGTH+1];

/* —– Forward declarations ——- */
static void handleQuery(const char * msg, int channel);
static void handleSetMessage(const char * msg, int channel);
static void handlePowerMessage(const char * msg, int channel);

void handleReceivedMessage(const char* msg, int channel)
{
        if( stringStartsWith(msg, "Q:") )
        {
                handleQuery(msg, channel);
        }
        else if (stringStartsWith(msg, "SET:"))
        {
                handleSetMessage(msg, channel);
        }
        else if (stringStartsWith(msg, "POWER"))
        {
                handlePowerMessage(msg, channel);
        }
        else
        {
                /* unsupported message. */
        }
}

/* ———— Private Functions ——— */

/**
 *
 * msg - the received ID request message
 * channel - the channel msg was received through
 *
 */
static void handleQuery(const char * msg, int channel)
{
        int toField;
        int n = sscanf(msg, "Q:%d", &toField);
        if(n==1)
        {
                int id = getFastenerIdentificationNumber();
                if( (toField == id) || (toField == BROADCAST_ID))
                {
                        getFastenerString(tempFastenerString);
                snprintf(tempMessageString, MAX_MESSAGE_LENGTH, "ID,%d:%s",
                        getFastenerIdentificationNumber(),tempFastenerString);
                transmitMessage(tempMessageString, 2); //transmit on top channel
            }
            else
            {
                /* addressed to someone else. */
            }
        }
        else
        {
            /* message does not match expected syntax */
        }
}

/**
 * Sets the internal beaconState string. The first
 * BEACON_STRING_FIELD_LENGTH characters of the msg contents are
 * copied, or up to the first whitespace.
 *
 * If the interface board is used to relay messages from laptop to
 * beacons, check that the timeout and max message length of the
 * interface board are long enough for the message contents.
 *
 * msg - The received SET message
 *
 * channel - The channel on which the SET message was received.
 */
static void handleSetMessage(const char * msg, int channel)
{
        int toField;
        int n = sscanf(msg, "SET:%d,%s", &toField, tempFastenerString);
        /*Check that both toField and contents string parsed from message.*/
        if(n == 2)
        {
            if( (toField == getFastenerIdentificationNumber())
                || (toField == BROADCAST_ID))
            {
                setFastenerString(tempFastenerString);
                transmitMessage("ACK", channel);
            }
            else
            {
                /*addressed to someone else.*/
            }
        }
        else
        {
            /*message does not match expected syntax*/
        }
}

/**
 * Set the duty cycle for IR transmission.
 */
static void handlePowerMessage(const char * msg, int channel)
{
        int dutyCycle = 50;
        char dutyCycleString[10];
        int toField;
        int n = sscanf(msg, "POWER:%d,%s", &toField, dutyCycleString);
        if(n==2)
        {
            if( (toField == getFastenerIdentificationNumber())
                || (toField == BROADCAST_ID))
            {
                dutyCycle = atoi(dutyCycleString);
                setModulationDutyCycle(dutyCycle);
            }
            else
            {
                /*addressed to someone else.*/
        }
    }
    else
    {
        /*message does not match expected syntax*/
    }
}
```

B.2.3 fastenerState.c

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "fastenerState.h"


/* extra 1 for null terminating character: */
char fastenerString[FASTENER_STRING_FIELD_LENGTH+1];


/*
  Internal Fastener State
*/
struct{
        char *const stringPtr;
        int identificationNumber;
} fastenerState =
{
        .stringPtr = &fastenerString[0],
        .identificationNumber = 7
};


/*
  Record nonvolatile state
*/
void init_FastenerState(void)
{
        setFastenerString("BLUE");
}

/* ——— Accessors ————        */

void setFastenerString(/*in*/const char *src)
{
        strncpy(fastenerState.stringPtr, src, FASTENER_STRING_FIELD_LENGTH);
}
```

```
void getFastenerString(char * dest)
{
        strncpy(dest, fastenerState.stringPtr, FASTENER_STRING_FIELD_LENGTH);
}

void setFastenerIdentificationNumber(int n)
{
        fastenerState.identificationNumber = n;
}

int getFastenerIdentificationNumber()
{
        return fastenerState.identificationNumber;
}
```

B.3 Robotic Interface Module Firmware

Due to limitations on device memory, two separate firmwares were developed for the robotic interface module. The first firmware is used with the instrumented gripper for assembly tasks. The second firmware is used with the optical sensor for robotic positioning.

B.3.1 Firmware for use with Instrumented Gripper

gripper.c

```
/* gripper.c
 *
 */
#include <avr/interrupt.h>
#include "serialCommManager.h"
#include "usart.h"
#include "gripperMessage.h"
#include "util_defines.h"
#include "status.h"
#include <util/delay.h>

#define BAUDRATE 2400

static char msg[MAX_MESSAGE_LENGTH];

int main(void)
{
        cli();

        /* ----- initializations ------ */
        init_SerialCommManager(BAUDRATE);
        init_StatusLEDs();

        sei();
```

```
    _delay_ms(50);

    (void)transmitMessage("#Gripper#", CHANNEL_SERIAL_PORT);
    (void)transmitMessage("#Gripper#", CHANNEL_IR_A);
    (void)transmitMessage("#Gripper#", CHANNEL_IR_B);
    (void)transmitMessage("#Gripper#", CHANNEL_IR_C);

    _delay_ms(250);
    setStatus0(1);
    _delay_ms(250);
    setStatus0(0);

    _delay_ms(250);

    setStatus0(1);
    _delay_ms(250);
    setStatus0(0);

    for(;;)
    {
        if( receiveMessage(msg, CHANNEL_SERIAL_PORT) == 1)Ĝ
        {
            handleReceivedMessage(msg, CHANNEL_SERIAL_PORT);
        }
    }
}
```

gripperMessage.c

```
/**
 * gripperMessage.c
 * Message handling routines for Gripper interface board.
 *
 * -Messages supported:
 *
 * Q:n,c - requests a QUERY to be forwarded on
 * channel c, c = "A","B" or "C"
 *
 * SET:n,c,contents - requests a SET,contents message to be forwarded
 * on channel c, c = "A","B" or "C"
 *
 * ? - request information about the firmware.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "gripperMessage.h"
#include "serialCommManager.h"
#include "usart.h"
#include "utils.h"
#include "timer1.h"

static char tempMessage[MAX_MESSAGE_LENGTH];

/* --- Forward declarations ---- */
static void handleQuery(/*in*/ const char * msg, int channel);
static void handleSetMessage(/*in*/ const char * msg, int channel);
static void handleFirmwareInfo(/*in*/ const char * msg, int channel);
static void handlePowerMessage(const char * msg, int channel);

/*!
  param[in] msg Pointer to received message null terminated string.
  param[in] channel The channel on which the message was received.
*/
void handleReceivedMessage(/*in*/ const char* msg, int channel)
{
        if (stringStartsWith(msg, "SET:"))
        {
                handleSetMessage(msg, channel);
        }
        else if( stringStartsWith(msg, "Q:") )
        {
                handleQuery(msg, channel);
        }
        else if ( stringStartsWith(msg, "?") )
        {
                handleFirmwareInfo(msg, channel);
        }
        else if ( stringStartsWith(msg, "POWER") )
        {
                handlePowerMessage(msg, channel);
        }
        else
        {
                transmitMessage("Unrecognized message:", CHANNEL_SERIAL_PORT);
                transmitMessage(msg, CHANNEL_SERIAL_PORT);
        }
}

/* -- Message Handlers ------ */

/*!
 *
 * param[in] msg The received ID request message.
 * param[in] channel Tthe channel msg was received through.
 *
 */
static void handleQuery(/*in*/ const char * msg, int channel)
```

```
{
    char forwardingChannelName;
    int forwardingChannelNumber = 0;
    int toField;
    int n = sscanf(msg, "Q:%d,%c", &toField, &forwardingChannelName);
    if (n==2)
    {
        switch (forwardingChannelName)
        {
        case 'a': case 'A':
            forwardingChannelNumber = CHANNEL_IR_A;
            break;
        case 'b': case 'B':
            forwardingChannelNumber = CHANNEL_IR_B;
            break;
        case 'c': case 'C':
            forwardingChannelNumber = CHANNEL_IR_C;
            break;
        default:
            transmitMessage("invalid channel", channel);
            break;
        }
    }
    else
    {
        transmitMessage("invalid query syntax", channel);
    }
    sprintf(tempMessage, "Q:%d", toField);

    transmitMessage(tempMessage, forwardingChannelNumber);

    if( receiveMessage(tempMessage, forwardingChannelNumber) == 1){
        transmitMessage(&tempMessage[3], channel);
    }else{
        transmitMessage("#timed out#", channel);
    }
}

static void handleSetMessage(/*in*/ const char * msg, int channel)
{
        char forwardingChannelName;
        int forwardingChannelNumber = 0;
        char contents[40];
        contents[0] = '\0';
        int toField;

        if(strlen(msg) > 40) return;

        int n = sscanf(msg, "SET:%d,%c,%s", &toField, &forwardingChannelName, contents);
        if (n==3)
        {
                switch (forwardingChannelName)
                {
                case 'a': case 'A':
                        forwardingChannelNumber = CHANNEL_IR_A;
                        break;
                case 'b': case 'B':
                        forwardingChannelNumber = CHANNEL_IR_B;
                        break;
                case 'c': case 'C':
                        forwardingChannelNumber = CHANNEL_IR_C;
                        break;
                default:
                        transmitMessage("invalid channel", channel);
                        break;
                }
        }
        sprintf(tempMessage, "SET:%d,%s", toField, contents);
        transmitMessage(tempMessage, forwardingChannelNumber);
}

static void handleFirmwareInfo(/*in*/ const char * msg, int channel)
```

```
{
        transmitMessage("FIRMWARE:Gripper.", channel);
}

/**
 * Set the duty cycle for IR transmission.
 */
static void handlePowerMessage(const char * msg, int channel)
{
        int dutyCycle = 50;
        char dutyCycleString[10];
        int n = sscanf(msg, "POWER:%s", dutyCycleString);
        if(n==1)
        {
                dutyCycle = atoi(dutyCycleString);
                setModulationDutyCycle(dutyCycle);
        }
        else
        {
                /*message does not match expected syntax*/
        }
}
```

B.3.2 Firmware for use with Optical Sensor

photo.c

```
/**
 * photo.c
 * Firmware for photo-diode interface board, used with laser beacons
 * Make sure F_CPU is set to 14745600
 *
 * Target device: atmega8
 */

#include <avr/interrupt.h>
#include <avr/io.h>
#include <stdlib.h>
#include <stdio.h>
#include <util/delay.h>
#include "serialCommManager.h"
#include "usart.h"
#include "photoMessage.h"
#include "util_defines.h"
#include "status.h"
#include "taos1410.h"

/*
  2400 is the maximum baud rate that the IR receivers can do reliably.
  Communication is the limiting factor in sensing latency, so if the photo interface PCB
  will not be performing wireless communication, baud can be cranked up to 19200 or higher.
 */
#define BAUDRATE 2400

char msg[MAX_MESSAGE_LENGTH];

int main(void)
{
  cli();
  // ----------- initializations --------------- //
  init_SerialCommManager(BAUDRATE);
  init_StatusLEDs();
  init_taos1410();

  sei();

  transmitMessage("#Photo#", CHANNEL_SERIAL_PORT);

  _delay_ms(500);
  setStatus0(1);
  _delay_ms(500);
  setStatus0(0);

  setStatus1(1);
  _delay_ms(500);
  setStatus1(0);

  for(;;)
  {
    if( receiveMessage(msg, CHANNEL_SERIAL_PORT) == 1)
    {
      handleReceivedMessage(msg, CHANNEL_SERIAL_PORT);
    }
  }
}
```

photoMessage.c

```
/**
 * photoMessage.c
 *
 * Message handling routines for photo-diode interface board.
 * Supports the following messages:
 *
 * SCAN - perform one scan of the photo diode's 1280 pixels.
 *
 * FASTSCAN,n - perform n scans in succession. n is a string of up to 4 characters.
 *
 * ACTIVATE,c - send an ACTIVATE message on channel c (c = 'A', 'B', or 'C')
 *
 * DEACTIVATE,c - send a DEACTIVATE message on channel c (c = 'A', 'B', or 'C')
 *
 * QUERY,c - send a QUERY message on channel c (c = 'A', 'B', or 'C')
 *
 * SET,c,contents - send a SET,contents message on channel c (c = 'A', 'B', or 'C')
 *
 * FORWARD,n,c#msg - forwards msg with n hops on channel c
 *
 * ? - request information about the firmware.
 *
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "photoMessage.h"
#include "serialCommManager.h"
#include "usart.h"
#include "utils.h"
#include "util_defines.h"
#include "taos1410.h"
#include "status.h"
```

```
// ----- Forward declarations -------- //
static void handleScan(const char * msg, int channel);

static void handleFastScan(const char * msg, int channel);

static void handleActivate(const char * msg, int channel);

static void handleDeactivate(const char * msg, int channel);

static void handleQuery(const char * msg, int channel);

static void handleSetMessage(const char * msg, int channel);

static void handleFirmwareInfo(const char * msg, int channel);

static void handleForward(const char * msg, int channel);

char tempMessage[MAX_MESSAGE_LENGTH];

/* ------- Public Functions -------- */

/*!
 *
 * Parses the received message, and responds.
 *
 * param[in] msg The received message.
 * param[in] channel The channel msg was received through.
 *
 */
void handleReceivedMessage(const char* msg, int channel)
{
        if( stringStartsWith(msg, "SC") )
        {
                handleScan(msg, channel);
        }
        else if( stringStartsWith(msg, "FASTSCAN") )
        {
                handleFastScan(msg, channel);
        }
        else if( stringStartsWith(msg, "ON:") )
        {
                handleActivate(msg, channel);
        }
        else if( stringStartsWith(msg, "OFF:") )
        {
                handleDeactivate(msg, channel);
        }
        else if( stringStartsWith(msg, "Q:") )
        {
                handleQuery(msg, channel);
        }
        else if( stringStartsWith(msg, "SET:") )
        {
                handleSetMessage(msg, channel);
        }
        else if ( stringStartsWith(msg, "?") )
        {
                handleFirmwareInfo(msg, channel);
        }
        else if ( stringStartsWith(msg, "FW"))
        {
                handleForward(msg, channel);
        }
        else{
                transmitMessage("unrecognized message:", CHANNEL_SERIAL_PORT);
                transmitMessage(msg, CHANNEL_SERIAL_PORT);
        }
}


/* ----- Private Functions ------- */
```

```
/*!
 *
 * param[in] msg The received ID request message.
 * param[in] channel Tthe channel msg was received through.
 *
 */
static void handleQuery(/*in*/ const char * msg, int channel)
{
        char forwardingChannelName;
        int forwardingChannelNumber = 0;
        int toField;
        int n = sscanf(msg, "Q:%d,%c", &toField, &forwardingChannelName);
        if (n==2)
        {
                switch (forwardingChannelName)
                {
                case 'a': case 'A':
                        forwardingChannelNumber = CHANNEL_IR_A;
                        break;
                case 'b': case 'B':
                        forwardingChannelNumber = CHANNEL_IR_B;
                        break;
                case 'c': case 'C':
                        forwardingChannelNumber = CHANNEL_IR_C;
                        break;
                default:
                        transmitMessage("invalid channel", channel);
                        break;
                }
        }
        else
        {
                transmitMessage("invalid query syntax", channel);
        }
        sprintf(tempMessage, "Q:%d", toField);
        transmitMessage(tempMessage, forwardingChannelNumber);
        if( receiveMessage(tempMessage, forwardingChannelNumber) == 1)
        {
                transmitMessage(&tempMessage[3], channel);
        }
        else
        {
                transmitMessage("#timed out#", channel);
        }
}

/*!
 *
 * param[in] msg The received ID request message.
 * param[in] channel The channel msg was received through.
 *
 */
static void handleScan(const char * msg, int channel)
{
        for(;;)
        {
                setStatus0(1);
                /*
                  Just using startPhotoDiodeScan instead of scanPhotoDiode
                  assumes that:
                  0.) High throughput is more important than latency.
                  1.) the photo diode state is valid for any calls made
                  by the messaging
                  2.) the photoDiode scan will complete during mesage transmission.
                */

                scanPhotoDiode();
                int laserDetected = laserWasDetected();
                int laserCoordinate = getMaxPixelIndex();
```

```
            if( laserDetected == 1 )
            {
                sprintf(tempMessage, "<P%d>", laserCoordinate);
                transmitRawStringNoPacket(tempMessage, CHANNEL_SERIAL_PORT);
            }else{
                transmitRawStringNoPacket("<LND>", CHANNEL_SERIAL_PORT);
            }
            setStatus0(0);
        }
}

/*!
 *
 * param[in] msg The received fastscan request message.
 * param[in] channel The channel msg was received through.
 *
 */
static void handleFastScan(const char * msg, int channel)
{
        int numScans = 0;
        char numScansString[4];
        int n = sscanf(msg, "FASTSCAN,%s", numScansString);
        if(n==1)
        {
                numScans = atoi(numScansString);
                int i;
                for(i = 0; i < numScans; ++i)
                {
                        handleScan(msg, channel);
                }
        }
        else
        {
                transmitMessage("Invalid FASTSCAN", channel);
        }
}
```

```
/*!
 *
 * param[in] msg The received activate request message.
 * param[in] channel The channel msg was received through.
 *
 */
static void handleActivate(const char * msg, int channel)
{
        char forwardingChannelName;
        int forwardingChannelNumber = 0;
        int toField;
        int n = sscanf(msg, "ON:%d,%c", &toField, &forwardingChannelName);
        if (n==2)
        {
                switch (forwardingChannelName)
                {
                case 'a': case 'A':
                        forwardingChannelNumber = CHANNEL_IR_A;
                        break;
                case 'b': case 'B':
                        forwardingChannelNumber = CHANNEL_IR_B;
                        break;
                case 'c': case 'C':
                        forwardingChannelNumber = CHANNEL_IR_C;
                        break;
                default:
                        transmitMessage("Invalid channel.", channel);
                        break;
                }

                //should only do this if case a,b, or c holds.
                sprintf(tempMessage, "ON:%d,", toField);
                transmitMessage(tempMessage, forwardingChannelNumber);
        }
        else
        {
                transmitMessage("Specify a channel.", channel);
        }
}

/*!
 *
 * param[in] msg The received deactivate request message.
 * param[in] channel The channel msg was received through.
 *
 */
static void handleDeactivate(const char * msg, int channel)
{
        char forwardingChannelName;
        int forwardingChannelNumber = 0;
        int toField;

        int n = sscanf(msg, "OFF:%d,%c", &toField, &forwardingChannelName);
        if (n==2)
        {
                switch (forwardingChannelName)
                {
                case 'a': case 'A':
                        forwardingChannelNumber = CHANNEL_IR_A;
                        break;
                case 'b': case 'B':
                        forwardingChannelNumber = CHANNEL_IR_B;
                        break;
                case 'c': case 'C':
                        forwardingChannelNumber = CHANNEL_IR_C;
                        break;
                default:
                        transmitMessage("Invalid channel.", channel);
                        break;
                }
                sprintf(tempMessage, "OFF:%d", toField);
        transmitMessage(tempMessage, forwardingChannelNumber);
    }
    else
    {
        transmitMessage("Specify a channel.", channel);
    }
}


static void handleSetMessage(/*in*/ const char * msg, int channel)
{
    char forwardingChannelName;
    int forwardingChannelNumber = 0;
    char contents[40];
    contents[0] = '\0';
    int toField;

    if(strlen(msg) > 40) return;

    int n = sscanf(msg, "SET:%d,%c,%s", &toField, &forwardingChannelName, contents);
    if (n==3)
    {
        switch (forwardingChannelName)
        {
        case 'a': case 'A':
            forwardingChannelNumber = CHANNEL_IR_A;
            break;
        case 'b': case 'B':
            forwardingChannelNumber = CHANNEL_IR_B;
            break;
        case 'c': case 'C':
            forwardingChannelNumber = CHANNEL_IR_C;
            break;
        default:
            transmitMessage("invalid channel", channel);
            break;
            }
        }
        sprintf(tempMessage, "SET:%d,%s", toField, contents);
        transmitMessage(tempMessage, forwardingChannelNumber);
}

static void handleFirmwareInfo(/*in*/ const char * msg, int channel)
{
        transmitMessage("FIRMWARE:Photo.", channel);
}

static void handleForward(/*in*/ const char * msg, int channel)
{
        int timeToLive;
        char forwardingChannelName;
        int forwardingChannelNumber = 0;
        char messageToForward[MAX_MESSAGE_LENGTH+1];

        int n = sscanf(msg, "FW,%d,%c#%s", &timeToLive, &forwardingChannelName, messageToForward);
        if(n==3)
        {
                sprintf(tempMessage, "FW,%d#%s", timeToLive, messageToForward);
                switch (forwardingChannelName)
                {
                case 'a': case 'A':
                        forwardingChannelNumber = CHANNEL_IR_A;
                        break;
                case 'b': case 'B':
                        forwardingChannelNumber = CHANNEL_IR_B;
                        break;
                case 'c': case 'C':
                        forwardingChannelNumber = CHANNEL_IR_C;
                        break;
                default:
                        transmitMessage("invalid channel", channel);
                        break;
            }
            transmitMessage(tempMessage, forwardingChannelNumber);
        }
        else
        {
            //didn't parse correctly.
            transmitMessage("FW didn't parse", channel);
        }
}
```

taos1410.c

```
/* taos1410.c
 *
 * Timer0 is used to produce a clock waveform to initiate sampling and
 * clock out data from a TAOS 1410r linear photodiode. The 1410r has
 * 1280 pixels. Rising and falling edges of the control waveforms are
 * determined by timer0 overflow interrupts.
 *
 * Sampling of the pixel values is performed by starting a
 * "single-shot" ADC conversion; the analog-to-digital conversion
 * complete interrupt performs the detection of the laser beam.
 *
 */

#include "taos1410.h"
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdlib.h>
#include <stdio.h>
#include "util_defines.h"

#define STATUS_0 PD2

#define STATUS_1 PD3

#define PHOTODIODE_SI PD4

#define PHOTODIODE_HOLD PD5

#define PHOTODIODE_CLK PD6

#define ADC_INPUT PC0

/* --- Constants ---- */
```

```
#define TIMER_ZERO_START 215 //for full resolution pixel sampling

#define NUM_INTEGRATION_CLOCKS 1

#define LASER_THRESHOLD 220

static struct{
        volatile int pdClkCount;
        volatile int sampling;          // "is-sampling" flag
        volatile int sampleIndex;

        volatile char maxSampleValue;
        volatile int maxSampleIndex;

        volatile char minSampleValue;
        volatile int minSampleIndex;

        volatile char sensingComplete;
} taosState;

/*  ———— Forward Declarations ————-     */

static void init_taosState();

static void init_timer0();

static void init_adc();

/*  ————    Public Functions ———— */

void init_taos1410()
{
        DDRD |= _BV(STATUS_0);
        DDRD |= _BV(STATUS_1);
        DDRD |= _BV(PHOTODIODE_SI);
        DDRD |= _BV(PHOTODIODE_HOLD);
        DDRD |= _BV(PHOTODIODE_CLK);

        init_taosState();
        init_timer0();
        init_adc();
}

/**
 *
 * Performs one scan of all 1280 sensor pixels and
 * detects the laser beam. Blocks until complete
 *
 *
 */
void scanPhotoDiode()
{
        init_taosState();
        //timer0 overflow interrupt enable:
        SETBIT(TIMSK, TOIE0);
        while(taosState.sensingComplete == 0)
        {
                //wait
        }
}

/**
 * Initiates a scan of all 1280 sensor pixels.
 * Non-blocking, user must then poll to see when complete.
 */
void startPhotoDiodeScan()
{
        init_taosState();
        //timer0 overflow interrupt enable:
        SETBIT(TIMSK, TOIE0);
}
```

```
/**
 * Returns 1 when sensing is complete.
 *
 */
int isPhotoDiodeScanComplete()
{
        return taosState.sensingComplete;
}

int getMaxPixelValue()
{
        return taosState.maxSampleValue;
}

int getMaxPixelIndex()
{
        return taosState.maxSampleIndex;
}

int laserWasDetected()
{
        if( taosState.maxSampleValue > LASER_THRESHOLD){
                return 1;
        }
        return 0;
}

/* ----- Private Functions -------- */

/**
 *
 *
 */
static void init_taosState()
{
        taosState.pdClkCount = 0;
        taosState.sampling = 0;
        taosState.sampleIndex = 0;
        taosState.maxSampleValue = 0;
        taosState.minSampleValue = 255;
        taosState.maxSampleIndex = 0;
        taosState.minSampleIndex = 0;
        taosState.sensingComplete = 0;
}

/**
 *
 *
 */
static void init_timer0()
{
        SETBIT(TCCR0, CS00);      //prescale: 1
        TCNT0 = TIMER_ZERO_START; //initial value
}

/**
 */
void init_adc(){
        /*
        Analog to Digital Converter Configured for Single Conversion
        Mode

        select prescale between system clock and ADC clock. The ADC
        clock must be between 50kHz and 200kHz for full accuracy. I
        think lower clock frequencies are more accurate, but since it
        takes 13 ADC clock cycles per sample (and 25 for the first
        sample!), a high clock frequency may be needed for adequate
        sample rate.

        Assuming F_CPU = 14.7456MHz, then 14745600/200kHz = 73.7
                so ballpark prescalers:
```

```
F_CPU/4 = 3.6864MHz  PS2 = 0   PS1 = 1     PS0 = 0
F_CPU/8 = 1.843MHz PS2 = 0     PS1 = 1     PS0 = 1
F_CPU/16 = 921.6KHz PS2 = 1    PS1 = 0     PS0 = 0
F_CPU/32 = 460.8KhZ PS2 = 1    PS1 = 0     PS0 = 1
F_CPU/64 = 230.4kHz PS2 = 1    PS1 = 1     PS0 = 0
F_CPU/128 = 115.2kHz  PS2 = 1  PS1 = 1     PS0 = 1

*/

//prescale 2
/*
  CLEARBIT(ADCSRA, ADPS2);
  CLEARBIT(ADCSRA, ADPS1);
  CLEARBIT(ADCSRA, ADPS0);
*/

//prescale 4
CLEARBIT(ADCSRA, ADPS2);
SETBIT(ADCSRA, ADPS1);
CLEARBIT(ADCSRA, ADPS0);

/*
//prescale 8
CLEARBIT(ADCSRA, ADPS2);
SETBIT(ADCSRA, ADPS1);
SETBIT(ADCSRA, ADPS0);
*/

/*
//prescale 16
SETBIT(ADCSRA, ADPS2);
CLEARBIT(ADCSRA, ADPS1);
CLEARBIT(ADCSRA, ADPS0);
*/

/*
```

```
    //prescale:32
    SETBIT(ADCSRA, ADPS2);
    CLEARBIT(ADCSRA, ADPS1);
    SETBIT(ADCSRA, ADPS0);
    */

    /*
    // prescale:64
    SETBIT(ADCSRA, ADPS2);
    SETBIT(ADCSRA, ADPS1);
    CLEARBIT(ADCSRA, ADPS0);
    */

    // Vref = AVCC with external capacitor at AREF pin:
    SETBIT(ADMUX, REFS0);

    //left-adjust for 8-bit precision by reading only ADCH, ignoring the two LSBs in ADCL
    SETBIT(ADMUX, ADLAR);

    //select analog channel for ADC_INPUT on pin ADC0

    //select single-conversion mode, as opposed to Free Running mode
    //CLEARBIT(ADCSRA, ADFR);

    //enable the ADC complete interrupt
    SETBIT(ADCSRA, ADIE);

    //finally, enable the ADC!
    SETBIT(ADCSRA, ADEN);

    SETBIT(ADCSRA, ADSC);
    while( BITVAL(ADCSRA, ADSC) == 1){
        //wait for first conversion to finish, since
        //the first conversion performs initialization of the ADC
    }
}
```

```
/**
 *
 * clocks data out of linear photodiode and initiates sampling.
 * Must call initializeSamplingState prior to enabling this ISR
 *
 */
ISR(TIMER0_OVF_vect)
{
    TCNT0 = TIMER_ZERO_START;
    //falling clock edge:
    if( BITVAL(PORTD, PHOTODIODE_CLK) == 1 )
    {
        CLEARBIT(PORTD, PHOTODIODE_SI);
        CLEARBIT(PORTD, PHOTODIODE_HOLD);
        CLEARBIT(PORTD, PHOTODIODE_CLK);

        //sampled all 1280 pixels?:
        if(taosState.sampleIndex == 1280){
            //done.
            taosState.sampling = 0;
            //clock should be low when not in use
            CLEARBIT(TIMSK, TOIE0);
        }
    }
    else
    {
        //rising clock edge
        taosState.pdClkCount++;
        if( taosState.pdClkCount == 1 )
        {
            SETBIT(PORTD, PHOTODIODE_SI);
        }
        //see datasheet for why the extra 18:
        if ( taosState.pdClkCount == NUM_INTEGRATION_CLOCKS + 18)
        {
            SETBIT(PORTD, PHOTODIODE_HOLD);
        }

        if( taosState.pdClkCount == 1281){
            SETBIT(PORTD, PHOTODIODE_SI);
            taosState.sampling = 1;
            //assume sampling index has already been reset to zero
        }

        //SI and Hold are clocked in on rising edge, so CLK
        // must be set only after SI and HOLD have been set
        SETBIT(PORTD, PHOTODIODE_CLK);

        //start a sample?
        if(taosState.sampling == 1){
            taosState.sampleIndex++;
            //only sample even pixels.
            if( (taosState.sampleIndex % 2) == 0)
            {
                SETBIT(PORTD, STATUS_0); //indicate sample beginning
                SETBIT(ADCSRA, ADSC); //start an ADC conversion
            }
            else
            {
                TCNT0 = 254;
            }
        }
    }
}

/**
 *
 * ADC conversion complete interrupt service handler
 * Processes each new pixel value.
 *
 * A potential issue is that if an interval of pixels saturate to 255,
 * then the coordinate of the 'max' is the first pixel. However, the
 * center of the interval makes more sense.
 *
 */
ISR(ADC_vect){
        CLEARBIT(PORTD, STATUS_0);
        if( taosState.sampleIndex > 20)
        {
                if(ADCH > taosState.maxSampleValue){
                        taosState.maxSampleValue = ADCH;
                        taosState.maxSampleIndex = taosState.sampleIndex;
                }
                if(ADCH < taosState.minSampleValue){
                        taosState.minSampleValue = ADCH;
                        taosState.minSampleIndex = taosState.sampleIndex;
                }
        }
        if(taosState.sampleIndex >= 1280)
        {
                taosState.sensingComplete = 1;
        }
}
```

status.c

```
/*
  status.c
*/

#include <avr/io.h>
#include "status.h"
#include "util_defines.h"

void setStatus0(int i)
{
        if(i==0)
        {
                SETBIT(PORTD, STATUS_0);
        }
        else
        {
                CLEARBIT(PORTD, STATUS_0);
        }
}

void setStatus1(int i)
{
        if(i==0)
        {
                SETBIT(PORTD, STATUS_1);
        }
        else
        {
                CLEARBIT(PORTD, STATUS_1);
        }
}

void init_StatusLEDs()
{
    DDRD |= _BV(STATUS_0);
    DDRD |= _BV(STATUS_1);
    /*status LEDS off by default:*/
    setStatus0(0);
    setStatus1(0);
}
```

B.4 Shared Communication Utilities

B.4.1 checksum.c

```
#include "checksum.h"
#include <stdint.h>

uint16_t crcUpdate (uint16_t crc, uint8_t data){
        int i;
        crc = crc ^ ((uint16_t)data << 8);
        for (i=0; i<8; i++)
        {
                if (crc & 0x8000)
                        crc = (crc << 1) ^ 0x1021;
                else
                        crc <<= 1;
        }
        return crc;
}
```

B.4.2 serialCommManager.c

```
/**
 * serialCommManager.c
 *
 * Abstraction for usart, communication multiplexer, and IR modulation.
 * This may end up being specific to each pcb, since they use different numbers of comm channels.
 */

#include "serialCommManager.h"

#include <avr/io.h>
#include <util/crc16.h>
#include <string.h>
#include <stdio.h>
#include <util/delay.h>
#include <stdint.h>

#include "usart.h"
#include "timer1.h"
#include "util_defines.h"
#include "checksum.h"

/* for setting selectA and selectB for the rx/tx multiplexer */
#define SELECT_A PC2
#define SELECT_B PC3

struct{
	int packetCount;
} serialCommState =
{
	.packetCount = 0
};
```

```
static void setSerialCommChannel(int c);
static void constructPacketString(/*out*/char* dest, const char * msg);

/*    ———- Public Functions —————-        */

void init_SerialCommManager(int baudrate)
{
        //configure mux select pins as output:
        DDRC |= _BV(SELECT_A);
        DDRC |= _BV(SELECT_B);
        //initialize comm hardware:
        init_usart(baudrate);
        initializeModulateIR();
}

/**
 * Blocks until the message has been transmitted
 *
 * returns 1 if transmitted, 0 otherwise
 */
int transmitMessage(/*in*/ const char * msgPtr, int channel)
{
        setSerialCommChannel(channel);
        if( isTransmitBufferEmpty() == 0 )
        {
                return 0;
        }
        else
        {
                char packet[MAX_PACKET_LENGTH];
                constructPacketString(packet, msgPtr);
                enqueueStringForTransmit(packet);
                transmitNextString();
                return 1;
        }
}
```

```
/**
 * An abstraction violation, used for debugging (and for an optimized,
 * low-latency control loop for laser positioning...)
 */
int transmitRawStringNoPacket(const char * msgPtr, int channel)
{
        setSerialCommChannel(channel);
        if(isTransmitBufferEmpty() == 0)
        {
                return 0;
        }
        else
        {
                enqueueStringForTransmit(msgPtr);
                transmitNextString();
                return 1;
        }
}

/**
 * Blocks until a message is received or the receiver times out
 *
 */
int receiveMessage(/*in*/ char * dest, int channel)
{
        setSerialCommChannel(channel);
        listenForMessage();
        if( isReceiveBufferFull() == 1 ){
                dequeueReceivedString(dest);
                return 1;
        }
        return 0;
}

void setIRModulationDutyCycle(int dutyCycle){
        setModulationDutyCycle(dutyCycle); //timer1 function
}

/* ———        Private Functions  ————      */

/**
 *
 * Sets selectA and selectB of rx/tx mux
 */
static void setSerialCommChannel(int c)
{
        switch(c)
        {
        case CHANNEL_SERIAL_PORT:
                CLEARBIT(PORTC, SELECT_A);
                CLEARBIT(PORTC, SELECT_B);
                break;
        case CHANNEL_IR_A:
                SETBIT(PORTC, SELECT_A);
                CLEARBIT(PORTC, SELECT_B);
                break;
        case CHANNEL_IR_B:
                CLEARBIT(PORTC, SELECT_A);
                SETBIT(PORTC, SELECT_B);
                break;
        case CHANNEL_IR_C:
                SETBIT(PORTC, SELECT_A);
                SETBIT(PORTC, SELECT_B);
                break;
        }
}

/**
 * dest must be of size MAX_PACKET_LENGTH to be safe.
 *
 */
```

```
static void constructPacketString(char* dest, const char * msg)
{
        //compute crc checksum
        uint16_t checksumCRC = 0xFFFF; //seed value
        uint8_t msgLength = strlen(msg);
        uint8_t i;
        for(i=0; i < msgLength; i++){
                checksumCRC = crcUpdate(checksumCRC, (unsigned char)msg[i] );
        }
        (void)snprintf(dest,MAX_PACKET_LENGTH, "<%s$%04X>", msg, checksumCRC);
}
```

B.4.3 status.c

```
/*
  status.c
*/

#include <avr/io.h>
#include "status.h"
#include "util_defines.h"

void setStatus0(int i)
{
        if(i==0)
        {
                SETBIT(PORTD, STATUS_0);
        }
        else
        {
                CLEARBIT(PORTD, STATUS_0);
        }
}

void setStatus1(int i)
{
        if(i==0)
        {
                SETBIT(PORTD, STATUS_1);
        }
        else
        {
                CLEARBIT(PORTD, STATUS_1);
        }
}

void init_StatusLEDs()
{
    DDRD |= _BV(STATUS_0);
    DDRD |= _BV(STATUS_1);
    /*status LEDS off by default:*/
    setStatus0(0);
    setStatus1(0);
}
```

B.4.4 timer1.c

```
/**
 * Timer1 is used for generating a 38kHz carrier frequency for IR
 * communication
 *
 * Configuration for Atmega8 Timer1 for 38kHz PWM output on pin OC1A
 * for driving an IR LED.
 *
 *  In phase-correct PWM mode, the timer counts from 0 up to a "TOP"
 *  value, and then counts back down. A THRESHOLD value is set so that
 *  the output is set, cleared, or toggled when crossed.
 *
 * ----------------------------- "TOP"
 *         /\
 *        /  \
 *       /    \           /
 *      /      \         /
 * ---   - - - -----   - - -  "THRESHOLD"
 *   /|         |    /|
 *  / |         |\  / |
 * /  |         | \/  |
 *  / |---------|  \/  |--------- PWM output
 *
 *
 * Here, "TOP" is the 16-bit ICR1 register, THRESHOLD is the 16-bit
 * OCR1 register, aand the pwm is output on the OC1A pin.
 *
 * The PWM frequency is chosen using F_PWM = F_CPU / (2*PRESCALE*TOP)
 *
 * Note: I initially selected Timer1 for generating modulated IR
 * because it both has an output compare unit (e.g. OC1A) that can be
 * set to toggle automatically, and because it has a threshold which
 * can be used to set the duty cycle of the PWM. In contrast, 8-bit
 * Timer2 has an output compare unit but no threshold, while 8-bit
 * Timer0 has neither. A 16-bit timer is not needed to generate the
 * 38KHz waveform, but the ability to set the duty cycle was
 * considered important, e.g. for regulating the average current
 * through the transmit IR LEDs. Thus, Timer1 met my needs, but if
 * duty cycle does not need to be varied, Timer2 could suffice.
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include "timer1.h"
#include "util_defines.h"
#include <util/delay.h>

#define F_PWM 38000
#define TOP_PWM 192
#define THRESHOLD_PWM 80

/* ------ Public Functions -------- */

/**
 * set up registers for phase-correct PWM mode 8, prescale 1, output
 * on OC1A. See AVR131
 */
void initializeModulateIR(void)
{
        /*set up OC1A as an output, DDRB, PB1*/
        DDRB |= _BV(PB1); //pin PB1 (aka OC1A) set as output
        ICR1 = TOP_PWM; //16-bit register
        OCR1A = THRESHOLD_PWM; //16-bit register

        /*
          bit      7      6      5    4     3     2     1     0
          TCCR1A: COM1A1 COM1A0 COM1B1 COM1B0 FOC1A FOC1B WGM11 WGM10
          TCCR1b: ICNC1 ICES1 - WGM13 WGM12 CS12 CS11 CS10

          the COM1A1:0 and COM1B1:0 control the output compare pins OC1A
          and OC1B respectively. If one or both of the COM1A1:0 bits
```

```
      are written to one, the OC1A output OVERRIDES the normal port
      operation of the IO pin it is connected to.
     */
    TCCR1A = _BV(COM1A1);
    TCCR1B = _BV(WGM13) | _BV(CS10);

    TCNT1 = 0; //reset timer count
}

/**
 * Enable PWM output on OC1A pin
 */
void enableModulatedOutput(void)
{
    SETBIT(TCCR1A, COM1A1);
}

void disableModulatedOutput(void)
{
    CLEARBIT(TCCR1A, COM1A1);
}

/*!
 *
 * Control IR modulation duty cycle.
 *  (Power levels 10%-40% are most useful.)
 *
 * param[in] i The percent duty cycle. Must be a mutltiple of 10.
*/
void setModulationDutyCycle(int i)
{
    disableModulatedOutput();
    /* assuming timer counts up to TOP_PWM = 192,
    10% increments are 19 counts
    */
    switch(i)
```

```
        {
        case 10:
                OCR1A = 19;
                break;
        case 20:
                OCR1A = 2*19;
                break;
        case 30:
                OCR1A = 3*19;
                break;
        case 40:
                OCR1A = 4*19;
                break;
        case 50:
                OCR1A = 5*19;
                break;
        case 60:
                OCR1A = 6*19;
                break;
        case 70:
                OCR1A = 7*19;
                break;
        case 80:
                OCR1A = 8*19;
                break;
        case 90:
                OCR1A = 9*19;
                break;
        case 100:
                OCR1A = 10*19;
                break;
        }
        enableModulatedOutput();
}
```

B.4.5 timer2.c

```
/**
 * timer2.c
 * Timeout timer.
 *
 *       Generates timer events every 1ms.
 *
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "util_defines.h"
#include "timer2.h"

static struct{
        /*number of overflow events so far:*/
        volatile int numEvents;

        /*number of overflows for timeout to occur:*/
        volatile int eventThreshold;

        /*flag:*/
        volatile int timedOut;

        /*value to set the timer to when reset/overflow:*/
        volatile int baseCount;
}timerState;

/**
 * t is timeout interval in ms.
 *
 */
void timer2_init_TimeoutTimer(int t)
{
        /*configure timer for 1ms overflows:*/
        TCCR2 = _BV(CS22) | _BV(CS21) ; //prescale 256
        timerState.baseCount = 200;

        timer2_resetTimeoutTimer();
        timerState.eventThreshold = t;
}

void timer2_resetTimeoutTimer(void)
{
        timerState.numEvents = 0;
        timerState.timedOut = 0;
        TCNT2 = timerState.baseCount;
        /*enable timer overflow interrupt:*/
        SETBIT(TIMSK, TOIE2);
}

int timer2_timeoutOccurred(void)
{
        return timerState.timedOut;
}

/**
 * Timer 2 overflow interrupt handler. This syntax is
 *      dependent on avr-gcc; beware of other compilers!
 *
 */
ISR(TIMER2_OVF_vect)
{
        timerState.numEvents++;
        if(timerState.numEvents >= timerState.eventThreshold){
                timerState.timedOut= 1;
                CLEARBIT(TIMSK, TOIE2);
        }
}
```

B.4.6 usart.c

```
/**
 * usart.c
 * Defines routines for interrupt-driven UART receiving and transmitting of packets.
 *
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <util/crc16.h>
#include <string.h>
#include "usart.h"
#include "timer1.h"
#include "timer2.h"
#include "util_defines.h"
#include "checksum.h"

#define TIMEOUT_INTERVAL_MS 50

/*
  Receiver State Machine
*/
typedef enum {WAIT_FOR_START_BYTE, BUFFER_DATA, CHECKSUM} RxState;

/*add 1 for the null terminating character*/
static volatile char receiveBuffer[MAX_PACKET_LENGTH + 1];
static volatile char transmitBuffer[MAX_PACKET_LENGTH + 1];

/*anything touched by interrupt handlers needs to be declared volatile.*/
static struct{
        volatile char *const receiveBufferPtr;
        volatile char *const transmitBufferPtr;
        volatile int receiveBufferFull;
        volatile int transmitBufferEmpty;

        volatile int transmitLength;
        volatile int receiveLength;

        volatile RxState rxState;
} usartState =
{
        .receiveBufferPtr = &receiveBuffer[0],
        .transmitBufferPtr = &transmitBuffer[0],
        .receiveBufferFull = 0,
        .transmitBufferEmpty = 1,
        .transmitLength = 0,
        .receiveLength = 0,
        .rxState = WAIT_FOR_START_BYTE
};

/*
  Forward declarations
*/
static void enableTransmitInterrupt(void);
static void disableTransmitInterrupt(void);
static void disableReceiveInterrupt(void);
static void enableReceiveInterrupt(void);

/* ——— Public Functions ——— */

/**
 *
 * Load lower 8-bits of baudPrescale into the low byte of the UBRR register
 * Load upper 8-bits of the baudPrescale into the high byte of the UBRR register
 */
void init_usart(unsigned int baudrate)
{
    unsigned long baudPrescale = (F_CPU / (16L * baudrate)) − 1;

    UBRRH = (unsigned char)(baudPrescale >> 8);
    UBRRL = (unsigned char)baudPrescale;

    8−bit, no parity, 1 stop bit, no handshaking:

    /*
      This is cryptic, so here's a translation: URSEL bit = 1: "write to
      UCSRC, not UBRRH which shares the same address. (only for
      atmega8, not atmega168) UMSEL bit = 0: asynchronous, 1
      synchronous UPM1, UPM0 both = 0: no parity USBS bit = 0: one
      stop bit UCSZ2 = 0, UCSZ1 = 1, UCSZ0 = 1: eight-bit character
      size
    */

    UCSRC |= (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1);
    /*override normal port operation with rx and tx:*/
    UCSRB |= (1 << RXEN) | (1 << TXEN);
}

static void resetReceiverStateMachine(void)
{
    usartState.rxState = WAIT_FOR_START_BYTE;
    usartState.receiveBufferFull = 0;
    usartState.receiveLength = 0;
}

int isReceiveBufferFull(void)/*modifies nothing*/
{
    return usartState.receiveBufferFull;
}


int isTransmitBufferEmpty(void)/*modifies nothing*/
{
        return usartState.transmitBufferEmpty;
}

/**
 * Loads a string into the transmit buffer.
 * Externally, transmit must be enabled to actually
 * send the contents of the buffer.
 */
void enqueueStringForTransmit(/*in*/ char *const ptr)
{
        int stringLength = (int)strlen(ptr);
        //cast discards volatile qualifier:
        strncpy((char *const)usartState.transmitBufferPtr, ptr, MAX_PACKET_LENGTH);
        usartState.transmitBufferEmpty = 0;

        if( stringLength <= MAX_PACKET_LENGTH ) {
                usartState.transmitLength = stringLength;
        } else {
                /*long message has been trucated!*/
                usartState.transmitLength = MAX_PACKET_LENGTH;
        }
}

/**
 * Once a string has been enqueued for transmit, this blocks
 * until it is transmitted.
 */
void transmitNextString(void)
{
        /*check for zero-length transmission (otherwise TXC will never get set):*/
        if(usartState.transmitLength > 0)
        {
                enableTransmitInterrupt();

                while( BITVAL(UCSRA, TXC) == 0)
                {
```

```
                /*wait for transmit to complete*/
            }
            /*
              clear TXC by WRITING A ONE to its location. See data sheet.
            */
            SETBIT(UCSRA, TXC);
        }
        usartState.transmitBufferEmpty = 1;
}

/**
 * Blocks until a message is received or a timeout occurrs.
 *
 */
void listenForMessage(void)
{
        resetReceiverStateMachine();
        timer2_init_TimeoutTimer(TIMEOUT_INTERVAL_MS);
        enableReceiveInterrupt();

        while ( (usartState.receiveBufferFull == 0) && (timer2_timeoutOccurred() == 0 ) )
        {
                /*wait for the receive buffer to become full, or for the timeout to occur*/
        }
        /*
          while the receive interrupt ought to be disabled when a complete
          message is received, explicitly disabling is necessary when a
          timeout occurs:
        */
        disableReceiveInterrupt();
}


void dequeueReceivedString(/*in*/ char * dest)
{
        if ( usartState.receiveBufferFull == 1 )
        {
                /*cast discard volatile qualifier:*/
                strcpy(dest, (char *const) usartState.receiveBufferPtr);
                usartState.receiveBufferFull = 0;
        }
}

/* ----- Private Functions --------      */

static void enableTransmitInterrupt(void)
{
        UCSRB |= (1 << UDRIE);
}

static void disableTransmitInterrupt(void)
{
        UCSRB &= ~(1 << UDRIE);
}



static void enableReceiveInterrupt(void)
{
        UCSRB |= (1 << RXCIE);
}

static void disableReceiveInterrupt(void)
{
        UCSRB &= ~(1 << RXCIE);
}

/**
 *
 * Transmitter. Requires transmitBufferPtr to point to data to be sent,
 * and usartState.transmitLength to contain the number of bytes to send.
 *
 * This syntax ISR(...) is dependent on avr-gcc. Beware of other compilers!
 */
ISR(USART_UDRE_vect)
{
        enum TXState {INITIALIZE, TRANSMIT};
        static enum TXState state = INITIALIZE;
        static volatile int buffIndex;
        static int bytesSent;

        switch(state)
        {
        case INITIALIZE:
                buffIndex = 0;
                bytesSent = 0;
                state = TRANSMIT;
                /*
                  no break. Fall-through is intended. I'm being
                  clever, so look here for a bug!
                 */
        case TRANSMIT:
                if( bytesSent < usartState.transmitLength){
                        UDR = usartState.transmitBufferPtr[buffIndex];
                        buffIndex++;
                        bytesSent++;
                } else {
                        /*done*/
                        state = INITIALIZE;
                        bytesSent = 0;
                        usartState.transmitBufferEmpty = 1;
                        disableTransmitInterrupt();
                }
                break;
        }
}

/**
 *
 * Receiver state machine for message syntax <....data.... &cccc> where
 * ...data... is variable-length, but cannot contain '<','>', or '&'
 *
 * This syntax ISR(...) is dependent on avr-gcc. Beware of other compilers!
 *
 */
ISR(USART_RXC_vect)
{
        static int numChecksumBytes;

        static volatile int buffIndex;

        static uint16_t checksum; //crc checksum calculated from received data

        switch(usartState.rxState)
        {
        case WAIT_FOR_START_BYTE:
                if (UDR == '<')
                {
                        timer2_resetTimeoutTimer();
                        buffIndex = 0;

                        usartState.rxState = BUFFER_DATA;
                        checksum = 0xFFFF;
                }
                else
                {
                        /*ignore received byte and wait for start byte*/
                }
                break;

        case BUFFER_DATA:
                if( buffIndex < MAX_MESSAGE_LENGTH )
                {
                        char temp = UDR;
                        // '$' marks the boundary between message and checksum bytes
```

```
            if( temp == '$' )
            {
                    //make the buffer a valid string:
                    usartState.receiveBufferPtr[buffIndex] = '\0';
                    /*set length before incrementing buffIndex:*/
                    usartState.receiveLength = buffIndex;
                    buffIndex++;

                    numChecksumBytes = 0;
                    usartState.rxState = CHECKSUM;
            }
            else            /*just a regular data byte*/
            {
                    usartState.receiveBufferPtr[buffIndex] = temp;
                    buffIndex++;
                    checksum = crcUpdate(checksum, temp);
            }

        }
        else    /*fail: message too long!*/
        {
            usartState.rxState = WAIT_FOR_START_BYTE;
            disableReceiveInterrupt();
            usartState.receiveBufferFull = 0;
        }
        break;

    case CHECKSUM:
        if(numChecksumBytes < 4)
        {
            usartState.receiveBufferPtr[buffIndex] = UDR;
            buffIndex++;
            numChecksumBytes++;
        }
        else /*have received all four checksum bytes.*/
        {
                /*unused*/char temp = UDR; //read to clear the RXC flag

                uint_16_t receivedChecksum;
                sscanf(usartState.receiveBufferPtr[bufferIndex−4], "%u", &receivedChecksum);

                if(checksum == receivedChecksum)
                {
                        usartState.rxState = WAIT_FOR_START_BYTE;
                        disableReceiveInterrupt();
                        usartState.receiveBufferFull = 1;
                }
                else
                {
                        /*checksums don't match. Could broadcast a NACK.*/
                }
            }
            break;
        }
}
```

B.4.7 utils.c

```
#include "utils.h"
#include <string.h>

int stringStartsWith(const char * str1, const char * str2)
{
        if ( strncmp(str1, str2, strlen(str2)) == 0 )
        {
                return 1;
        }
        return 0;
}
```

Appendix C

Instrumented Gripper Java Source Code

This appendix presents the Java control software for performing grasping of assembly materials using the instrumented gripper.

C.0.8 ArmControl.java

```
package arm;

import gripper.Gripper;

import javax.comm.*;

import java.io.DataInputStream;
import java.io.IOException;

import lcm.lcm.LCM;
import lcm.lcm.LCMSubscriber;
import assembly_robot.lcm.cState_t;

/**
 * The armContoller is intended to abstact away the motion of the arm to setting
 * the angle of the arm, the setting of the gripper and the position of the arm.
 * The arm positions are mostly defined relative to the "dropping point", which
 * is a point defined to be exacty 4 cm in front of the roombas we use, but can
 * be safley redefined as long as all robots are calibrated similarly. When
 * initialized the arm takes in the positions {gripper open, arm left, arm pose
 * medium} and {gripper closed, arm rigth, arm pose medium} arguments. positions
 * are defined below.
 * <P>
 * <ul>
 * <li><b> gripper open </b> - the gripper is open(<I>true</i>) or the gripper
 * is not open (<i>false</i>)
 *
 * <li><b> arm angle </b> - the angle of the arm relative to the robot. ranges
 * from (<code>
 * -PI/2</code>) to (<code>PI/2</code>)
 *
 * <li><b> arm position </b> - the position of the arm, has multiple position
 * <li><b> CARRY </b> - the arm is ready to move about the course without the
 * arm getting in the way
 * <li><b> HIGH </b> - the arm is in a high position parallel to it's dropping
 * point
 * <li><b> MEDUIM </b> - the arm is above the dropping point such that a truss
 * would be set on the 2nd tier of a structure
 * <li><b> LOW </b> - the arm is above the dropping point such that it would
 * lift a truss or fastener off the floor
 *
 * </ul>
 *<P>
 * though not strictly enforced, this will not work if not treated as singleton.
 * When using this system there must be one instance of main() running for each
 * robot, and it must be local.
 *
 * @author <a href="mailto:stein@csail.mit.edu">David Stein</a>
 * @version 1.0
 */
public class ArmControl implements LCMSubscriber {
```

```
private static final LCM lcm = LCM.getSingleton();

/**
 * servo values
 */
private final int GRIPPER = 0, WRIST = 1, ELBOW = 2, SHOULDER = 3, BASE = 4;

/**
 * index values
 */
private final int UP = 0, DOWN = 1, CLOSED = 0, OPEN = 1, LEFT = 0, RIGHT = 1;

/**
 * the current base angle. used by the search piece
 */
private double currentBaseAngle, currentShift = Math.PI / 60;

/**
 * the gripper
 */
private Gripper gripper;

/**
 * the servo manager
 */
private LynxServo myArm;

/**
 * the unique robotID of this robot
 */
private int robotID;

/**
 * the {low, high} combo of position calibration values for each of the
 * servos ie servoPosition[servo][low=0, high=1]
 */
```

```
private int[][] servoPositionC;

/**
 * the position of the arm, as defined above.
 */
private enum ArmPosition {
    CARRY, HIGH, MEDIUM, LOW;
}

/**
 * the most recent relevant cState_t message
 */
private cState_t mostRecentCommand;

/**
 * the value of the desired fastener type
 */
private String target;

/**
 * Creates a new <code>ArmControl</code> instance.
 *
 * @param robotID the robot ID
 * @param upPositions the set of servo PWMs corresponsing to scanning height (see top)
 * @param downPositions the set of serve PWMs corresponting to grabbing height (see top)
 */
public ArmControl(int robotID, int[] upPositions, int[] downPositions, String target) {
    System.out.println("\t\t\t\t\tREVISION 1");
    servoPositionC = new int[5][2];
    for (int j = 0; j < 5; j++) {
        servoPositionC[j][1] = upPositions[j];
        servoPositionC[j][0] = downPositions[j];
    }
    this.robotID = robotID;
    this.target = target;
    this.myArm = new LynxServo();
```

```
        this.gripper = new Gripper(SensorPort.getInstance());
    }

    /**
     * the main loop. checks the state and calls the appropriate sub method
     */
    public void start() {
        System.out.println("ArmControl.start(): started");
        while(true){
            try {Thread.sleep(500);}catch(Exception e){}
            if (myArm.isArmReset()) {
                System.out.println("ArmControl.start: recovering...");
                recover();
                System.out.println("ArmControl.start: recovered!");
            }
            String messageFromGripper = gripper.getFastenerMessage(Gripper.BELOW);
            System.out.println("\nFastenerMessage:  \t(" + messageFromGripper+")");
            System.out.println("Do I Like It? ("+target+")\t" + target.equals(messageFromGripper));
        }
    }

    /**
     * does a slow scan untill it hits a fastener with the target value or sends
     * a falure message after four scans
     */
    private boolean search() {
        // TODO needs to be more loop like and locked
        int i = 0;
        positionArm(ArmPosition.MEDIUM);
        setGripper(false);//closed
        System.out.println("ArmControl.search: starting");
        while (i <= 4) {
            // change direction at edges
            currentShift = Math.abs(currentBaseAngle) >= Math.PI / 4 ? -currentShift : currentShift;
            i += Math.abs(currentBaseAngle) >= Math.PI / 4 ? 1 : 0;// increment i when directection changes
            currentBaseAngle += currentShift;
        System.out.println("CurrentShift = " +currentShift + "\nCurrentBaseAngle = PI*" + (currentBas

        // move the base a bit.
        boolean seenIt = smoothSetBase(getBasePWM(currentBaseAngle));

        // if we hit then we scan
        if (seenIt) {
            currentBaseAngle -= currentShift;
            seenIt = smoothSetBase(getBasePWM(currentBaseAngle));
            while (!seenIt){
                currentBaseAngle += currentShift/3;
                seenIt = smoothSetBase(getBasePWM(currentBaseAngle));
            }
            double oneSide = currentBaseAngle;
            double otherSide = currentBaseAngle;
            // find the edges and go to the middle
            while (seenIt&& Math.abs(currentBaseAngle) < Math.PI / 2) {
                seenIt = smoothSetBase(getBasePWM(currentBaseAngle));
                currentBaseAngle += currentShift/3;
                otherSide = currentBaseAngle;
            }

            // do the move
            setBase(getBasePWM(otherSide / 2 + oneSide / 2));
            grabAtTheGround();
            positionArm(ArmPosition.MEDIUM);
            return true;
        }
    }
    System.out.println("search failed");
    return false;
}

/**
 * moves the arm to a pose
 */
```

```
private void positionArm(ArmPosition pos) {
    int[] armPosition = getPosition(pos);
    String commandString = "";
    for (int i = 0; i < 3; i++) {
        commandString += "#" + (i + 1) + " P" + armPosition[i] + " S75 ";
    }
    commandString += (char)13;
    System.out.println("sent command: "+commandString);
    myArm.writeToPort(commandString);
    while (checkMovingState()) {
    }
    System.out.println("DONE POSITIONING!");
    return;
}

/**
 * sets the position of the base
 */
private void setBase(int PWM){
    myArm.writeToPort("#4 p"+PWM+" s200" + (char)13);
    while (checkMovingState()) {
    }
}

/**
 * sets the position of the base
 */
private boolean smoothSetBase(int PWM){
    myArm.writeToPort("#4 p"+PWM+" s70"+(char)13);
    boolean returnVal = target.equals(gripper.getFastenerMessage(Gripper.BELOW));
    while (checkMovingState()) {
    }
    return returnVal;
}

/**
```

```
 * grab at the ground. Should probably check something is there first.
 */
private void grabAtTheGround() {
    setGripper(true); // open
    positionArm(ArmPosition.LOW);
    try{Thread.sleep(500);}catch(Exception e){}
    setGripper(false); // closed
    return;
}

/**
 * grab at the ground. Should probably check something is there first.
 */
private void setOnTheGround() {
    positionArm(ArmPosition.LOW);
    setGripper(true); // open
    positionArm(ArmPosition.CARRY);
    currentBaseAngle = 0;
    setBase(getBasePWM(0));
    return;
}

/**
 * moves and blocks untill done moving
 */
private boolean checkMovingState() {
    // send a query
    try {
        myArm.writeToPort("q" + String.valueOf((char) 13));
        // get the response, return false if still moving
        Thread.sleep(50);
    } catch (Exception e) {
    }
    return myArm.isArmMoving();
}
```

```
/**
 * recovers from a hardware reset
 */

private void recover() {
    System.err.println("arm reset detected");
    try {
        myArm.writeToPort("#" + 0 + " p800" + String.valueOf((char)13));
        Thread.sleep(2000);
        System.out.println("ArmControl.recover: next");
    } catch (Exception e) {
    }
    for (int i = 4; i >= 1; i--) {
        try {
            myArm.writeToPort("#" + i + " p1300" + String.valueOf((char)13));
            Thread.sleep(2000);
            System.out.println("ArmControl.recover: next");
        } catch (Exception e) {
        }
    }
    System.out.println("ArmControl.recover: go!");
    return;
}

/**
 * sends a message to the arm telling it to move. Takes PWMs
 */
private void moveArm(int[] position) {
    String commandString = "";
    for (int i = 0; i < 5; i++) {
        commandString += "#" + i + " P" + position[i] + " S75 ";
    }
    commandString += String.valueOf((char) 13); // a carrage return marks the end of a message
    myArm.writeToPort(commandString);
    System.out.println("armMoved");
}
```

```
/**
 * sets the gripper to open or closed
 *
 * @param open - true means open, false means closed.
 */
private void setGripper(boolean open) {
    if (open) {
        myArm.writeToPort("#0 p" + getGripperOpenPWM() + " s175" + ((char) 13));
    } else {
        myArm.writeToPort("#0 p" + getGripperClosedPWM() + " s175" + ((char) 13));
    }
    while (checkMovingState()) {
    }
    System.out.println("gripper moved");
}

/**
 * returns the value of servos 1, 2, and 3 neccessary to acheive a given
 * position. Does computation by hardcoded linear combinations of calibrated
 * positions
 *
 * @param pos an <code>ArmPosition</code> value
 * @return an <code>int[]</code> value
 */
private int[] getPosition(ArmPosition pos) {
    double[] lowMultiplier = new double[3];
    double[] highMultiplier = new double[3];
    switch (pos) {
    case HIGH:
        highMultiplier[0] = -1;
        highMultiplier[1] = 2;
        highMultiplier[2] = 1;
        // highMultiplier = {-1, 2, 1};
        lowMultiplier[0] = 1;
        lowMultiplier[1] = -1;
        lowMultiplier[2] = .25;
        // lowMultiplier = { 1, -1, .25};
        break;
    case LOW:
        highMultiplier[0] = 0;
        highMultiplier[1] = 0;
        highMultiplier[2] = 0;
        lowMultiplier[0] = 1;
        lowMultiplier[1] = 1;
        lowMultiplier[2] = 1;
        // highMultiplier = {0,0,0};
        // lowMultiplier = {1,1,1};
        break;
    case MEDIUM:
        highMultiplier[0] = 1;
        highMultiplier[1] = 1;
        highMultiplier[2] = 1;
        lowMultiplier[0] = 0;
        lowMultiplier[1] = 0;
        lowMultiplier[2] = 0;
        // highMultiplier = {1,1,1};
        // lowMultiplier = {0,0,0};
        break;
    case CARRY:
        highMultiplier[0] = −1;
        highMultiplier[1] = 0;
        highMultiplier[2] = 1.5;
        lowMultiplier[0] = 1;
        lowMultiplier[1] = 1;
        lowMultiplier[2] = −.125;
        // highMultiplier = {-1, 0, 2};
        // lowMultiplier = { 1, 1, -.125};
        break;
    }

    int[] returnSet = new int[3];
    for (int i = 0; i < 3; i++) {
        int nextVal = (int)((((double) servoPositionC[i+1][0]) * lowMultiplier[i]) + (((double) servoPositior
        System.out.println("index: " + i + "\tvalue: " +nextVal + "\thighMultiplier, servoPos[i][1]
        returnSet[i] = nextVal;
    }
    return returnSet;
}

private int getGripperOpenPWM() {
    return servoPositionC[GRIPPER][OPEN];
}

private int getGripperClosedPWM() {
    return servoPositionC[GRIPPER][CLOSED];
}

private int getBasePWM(double angle) {
    double lowM, highM;
    lowM = Math.PI / 2 - angle;
    highM = Math.PI / 2 + angle;
    double returnVal = ((double) servoPositionC[BASE][LEFT]) * lowM + ((double) servoPositionC[BASI
    return (int) (returnVal / Math.PI);
}

/**
 * recieved an LCM message and passes it to a handler
 */
public void messageReceived(LCM lcm, String channel, DataInputStream dis) {
    if (channel.equals("GLOBAL_POSITION")) {
        // TODO: assembler will need to use this information.
    } else if (channel.equals("WORLD_STATE")) {
        /*
         * the world state relays commands from the central robot state
         * machine singlton
         */
        try {
                cState_t msg = new cState_t(dis);
                System.out.println("RobotNavigator.messageRecieved:State update if " + robotID + " == "
                if ((msg.status == cState_t.SENT || msg.status == cState_t.REPUBLISHING)) {
                    if (msg.srcRobotID == robotID && msg.module == cState_t.ARM){
                        parseWorldState(msg);
                    } else if (msg.srcRobotID == robotID && msg.module == cState_t.NAVIGATOR){
                        positionArm(ArmPosition.CARRY);
                        currentBaseAngle = 0;
                        setBase(getBasePWM(0));
                    }
                }
            } catch (Exception e) {
            }
        }
    }

    /**
     * reads a {@link cState_t} and updates the state of the Navagation FSM
     *
     * @param msg the new state
     */
    private void parseWorldState(cState_t msg) throws Exception {
        mostRecentCommand = msg;
        msg.status = cState_t.ACK;
        lcm.publish("WORLD_STATE", msg);
        System.out.println("ArmControl.parseWorldState: recieved");
        switch (msg.command) {
        case cState_t.PICK:
            System.out.println("Picking");
            if (search()) {
                sendTaskCompleteMessage(msg);
            } else {
                sendTaskFailureMessage();
            }
            break;
        case cState_t.RELEASE:
            setOnTheGround();
            sendTaskCompleteMessage(msg);
            break;
        }
    }

    /**
     * informs the planner of the robot that the current task is complete
     *
     */
    private void sendTaskCompleteMessage(cState_t msg) {
        if (mostRecentCommand == null)
            return;
        // make and send a success response
        msg.utime = System.currentTimeMillis();
        msg.status = cState_t.CMD_SUCCEED;
        lcm.publish("WORLD_STATE", msg);
        System.out.println("COMPLETED TASK");
    }

    /**
     * informs the planner of the robot that the current task is complete, but
     * failed
     *
     */
    private void sendTaskFailureMessage() {
        if (mostRecentCommand == null)
            return;
        // make and send a success response
        cState_t msg = mostRecentCommand;
        msg.utime = System.currentTimeMillis();
        msg.status = cState_t.CMD_FAIL;
        lcm.publish("WORLD_STATE", msg);
        System.out.println("FAILED TASK");
    }
```

```
/**
 * starts up the arm controller, expects the values of UP and DOWN, which
 * are used to generate all the positions of the arm by assuming a linear
 * mapping from the angle of a servo to the PWM provided to it.
 * <P>
 * the <B>UP</B> position corresponds to the position the servo is in
 * immediatly after having lifted an object out of its container. (arm all
 * the way left, gripper closed)
 * <P>
 * the <B>DOWN</b> position corresponds to the position the servo is in
 * immediately before grabbing an object. (arm all the way right, gripper
 * open)
 *
 * @param args "-r RobotID -u u0 u1 u2 u3 u4 -d d0 d1 d2 d3 d4 -t targetString"
 * <br>ie:
 * <code>java ArmControl -r 5 -u 1400 1300 1200 800 -d 850 1300 1238 1429 -t GREEN</code>
 */
public static void main(String[] args) {
    // set static instace variables
    System.out.println("ArmControl.main: staring");
    int robotID = -1;
    int[] upPosition = { -1, -1, -1, -1, -1 };
    int[] downPosition = { -1, -1, -1, -1, -1 };
    String target = "-1";
    // parse args
    if (args.length > 1) {
        for (int i = 0; i < args.length - 1; i++) {
            if ("-r".equals(args[i].trim()) && args.length >= i + 1) {
                System.out.println("robot id: " + args[i + 1]);
                robotID = Integer.parseInt(args[i + 1]);
                i += 1;
            } else if ("-u".equals(args[i].trim()) && args.length >= i + 5) {
                upPosition[0] = Integer.parseInt(args[i + 1]);
                upPosition[1] = Integer.parseInt(args[i + 2]);
                upPosition[2] = Integer.parseInt(args[i + 3]);
                upPosition[3] = Integer.parseInt(args[i + 4]);
            upPosition[4] = Integer.parseInt(args[i + 5]);
            i += 5;
        } else if ("-d".equals(args[i].trim()) && args.length >= i + 5) {
            downPosition[0] = Integer.parseInt(args[i + 1]);
            downPosition[1] = Integer.parseInt(args[i + 2]);
            downPosition[2] = Integer.parseInt(args[i + 3]);
            downPosition[3] = Integer.parseInt(args[i + 4]);
            downPosition[4] = Integer.parseInt(args[i + 5]);
            i += 5;
        } else if ("-t".equals(args[i].trim()) && args.length >= i + 1) {
            System.out.println("target: \'" + args[i + 1] + "\'");
            target = args[i + 1];
            i += 1;
        } else {
            System.err.println("unrecognized input.");
        }
    }
}

// check all args were entered
if (robotID < 0 || upPosition[0] < 0 || downPosition[0] < 0 || "-1".equals(target)) {
    System.err.println("insufficient args entered into ArmControl.main");
    System.exit(1);
}

// startup
ArmControl ac = new ArmControl(robotID, upPosition, downPosition,target);

// subscibe to LCM
LCM.getSingleton().subscribe("GLOBAL_POSITION", ac);
LCM.getSingleton().subscribe("WORLD_STATE", ac);

System.out.println("ArmControl.main: running main. . .");
// start the program
ac.start();
}
```

```
}
```

C.0.9 Gripper.java

```
package gripper;

import java.io.IOException;
import java.util.TooManyListenersException;

import message.AssemblyMessage;
import message.FirmwareInfoMessage;
import message.FirmwareInfoRequestMessage;
import message.IDMessage;
import message.QueryMessage;
import message.SetMessage;
import message.SetPowerMessage;
import serial.PortManager;

/**
 * the gripper class abstracts away the gripper and allows simple calls
 * to "look" through the gripper.
 *
 * @author mfaulk
 * @date Jun 27 2009 - minor edit by David Stein
 */
public class Gripper {
        private MessageQueue messageQueue;
        public static final String BELOW = "A", FRONT = "B", IN_GRIPPER = "C";

        /**
         * initialize gripper
         */
        public Gripper(PortManager portManager){
                try {
                        this.messageQueue = new MessageQueue(portManager);
                } catch (TooManyListenersException e) {
                        e.printStackTrace();
                }
}

/**
 * Requests the contents string from fastener with ID idNumber.
 * idNumber
 */
public String getFastenerMessage(String channel, int idNumber) {
        IDMessage msg = requestID(channel, idNumber, 750, 1);
        if (msg == null)
                return null;
        else{
                String returnStr = msg.getContents();
                //returnStr = returnStr.subString(0,returnStr.length()-6);
                return returnStr;
        }
}

public String getFastenerMessage(String channel){
        return getFastenerMessage(channel, QueryMessage.BROADCAST_ID);
}

/**
 * returns true if there is a fastener seen on the given channel
 */
public boolean isFastenerSeen(String channel, int idNumber){
        boolean returnVal = false;
        IDMessage msg = requestID(channel, idNumber, 750, 1);
        if(msg != null){
                returnVal = true;
        }
        return returnVal;
}

public boolean isFastenerSeen(String channel){
        return isFastenerSeen(channel, QueryMessage.BROADCAST_ID);
}
```

```
/**
 * sets the Message on the fastener the gripper currently seen
 * <b> not guarenteed to succeed</b> please use in a closed loop
 * with getFastenerMessage
 */
public void sendSetMessage(String channel, int idNumber, String contents){
	setID(channel, idNumber, contents);
}

public void sendSetMessage(String channel, String contents){
	sendSetMessage(channel, SetMessage.BROADCAST_ID, contents);
}



/**
 * Retrieve information about the gripper interface board firmware.
 * @return Firmware info, or null interface board does not respond.
 */
public String getFirmwareInfo(){
	int timeOutMS = 500;
	messageQueue.clear();
	try {
		messageQueue.put(new FirmwareInfoRequestMessage());
	} catch (IOException e) {
		e.printStackTrace();
	}

	AssemblyMessage responseMsg = messageQueue.next(timeOutMS);
	if( (responseMsg != null) && (responseMsg instanceof FirmwareInfoMessage) ){
		FirmwareInfoMessage info = (FirmwareInfoMessage) responseMsg;
		System.out.println("Firmware: "+ info.getContents());
		return info.getContents();
```

```
        }

        return null;
}

/**
 *
 * @param power An integer between 1 and 10
 */
public void setGripperTransmitterPower(int power){
        try{
                messageQueue.put(new SetPowerMessage(power*10));
        }catch(Exception e){
                e.printStackTrace();
        }
}

/**
 * Free resources
 */
public void cleanup() {
        messageQueue.cleanup();
}


/**
 *
 * @param channel Channel to send ID request through
 * @param timeOutMS milliseconds to wait for a response
 * @return the received ID message, or null if no response is received
 */
private IDMessage requestID(String channel, int address, long timeOutMS, int numAttempts){

        IDMessage idMessage = null;
        for(int i = 0; i < numAttempts; ++i){
                messageQueue.clear();
                try {
                        messageQueue.put(new QueryMessage(channel, address));
                } catch (IOException e) {
                        e.printStackTrace();
                }

                AssemblyMessage responseMsg = messageQueue.next(timeOutMS);
                if( (responseMsg != null) && (responseMsg instanceof IDMessage) ){
                        System.out.println("got an ID response: "+ responseMsg.getMessageString() );
                        idMessage = (IDMessage) responseMsg;
                        break;
                }
        }
        return idMessage;
}

/**
 * @param channel The channel on which to send the SET message.
 * @param contents The receiver's ID is set to these contents. May not contain whitespace.
 */
private void setID(String channel, int address, String contents){

        if(contents.contains(" ")){
                System.err.println("setID contents may not contain whitespace");
        }

        try {
                messageQueue.put(new SetMessage(address, channel, contents));
        } catch (IOException e) {
                e.printStackTrace();
        }
}
}
```

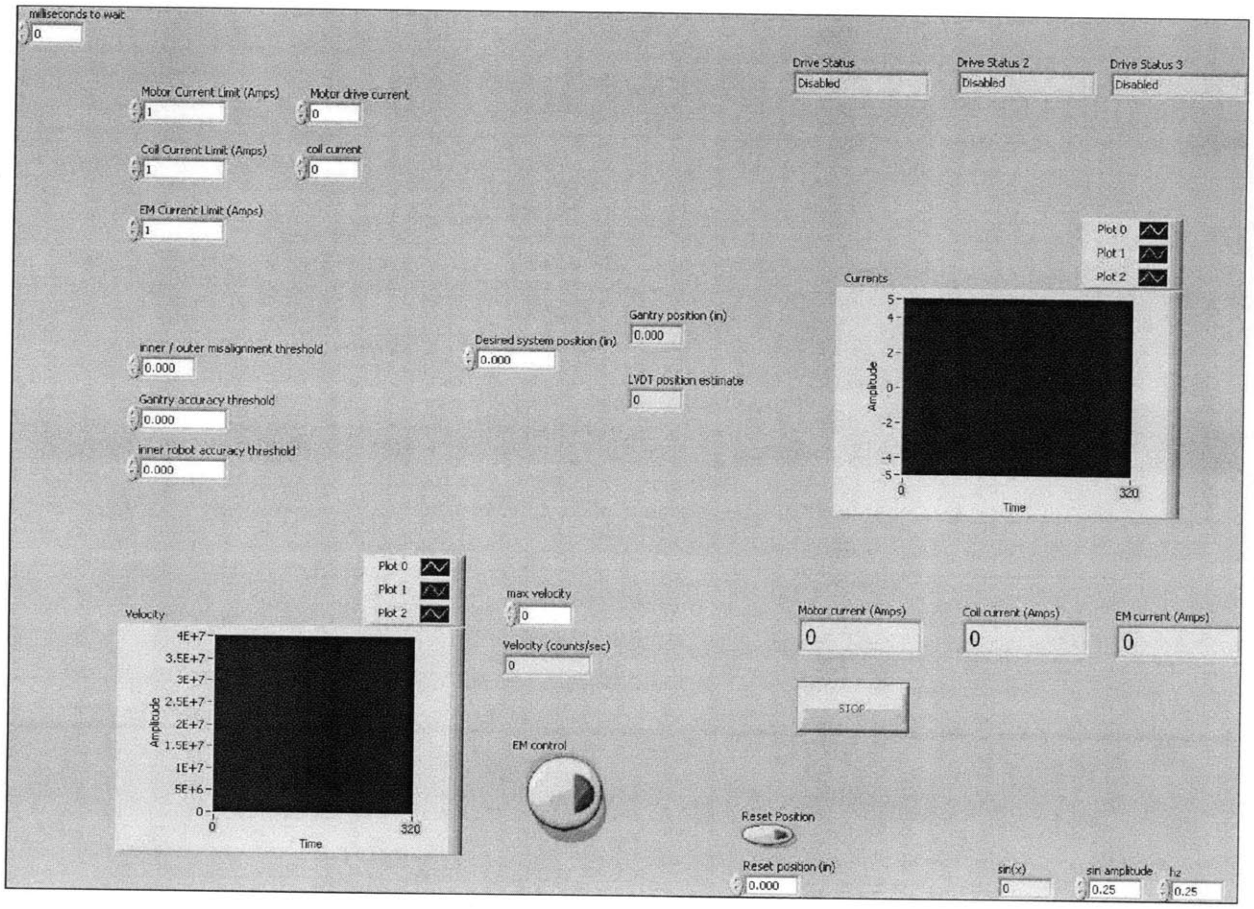
Appendix D

Optical Positioning Feedback Controller

This appendix presents the Labview control software, written in the G programming language, to perform closed-loop control of a magnetically actuated robot.

D.1 Graphical User Interface

D.2 Control Loop Block Diagram



milliseconds to wait
0

Motor Current Limit (Amps)
1

Motor drive current
0

Coil Current Limit (Amps)
1

coil current
0

EM Current Limit (Amps)
1

Drive Status
Disabled

Drive Status 2
Disabled

Drive Status 3
Disabled

inner / outer misalignment threshold
0.000

Gantry accuracy threshold
0.000

inner robot accuracy threshold
0.000

Desired system position (in)
0.000

Gantry position (in)
0.000

LVDT position estimate
0

max velocity
0

Velocity (counts/sec)
0

Motor current (Amps)
0

Coil current (Amps)
0

EM current (Amps)
0

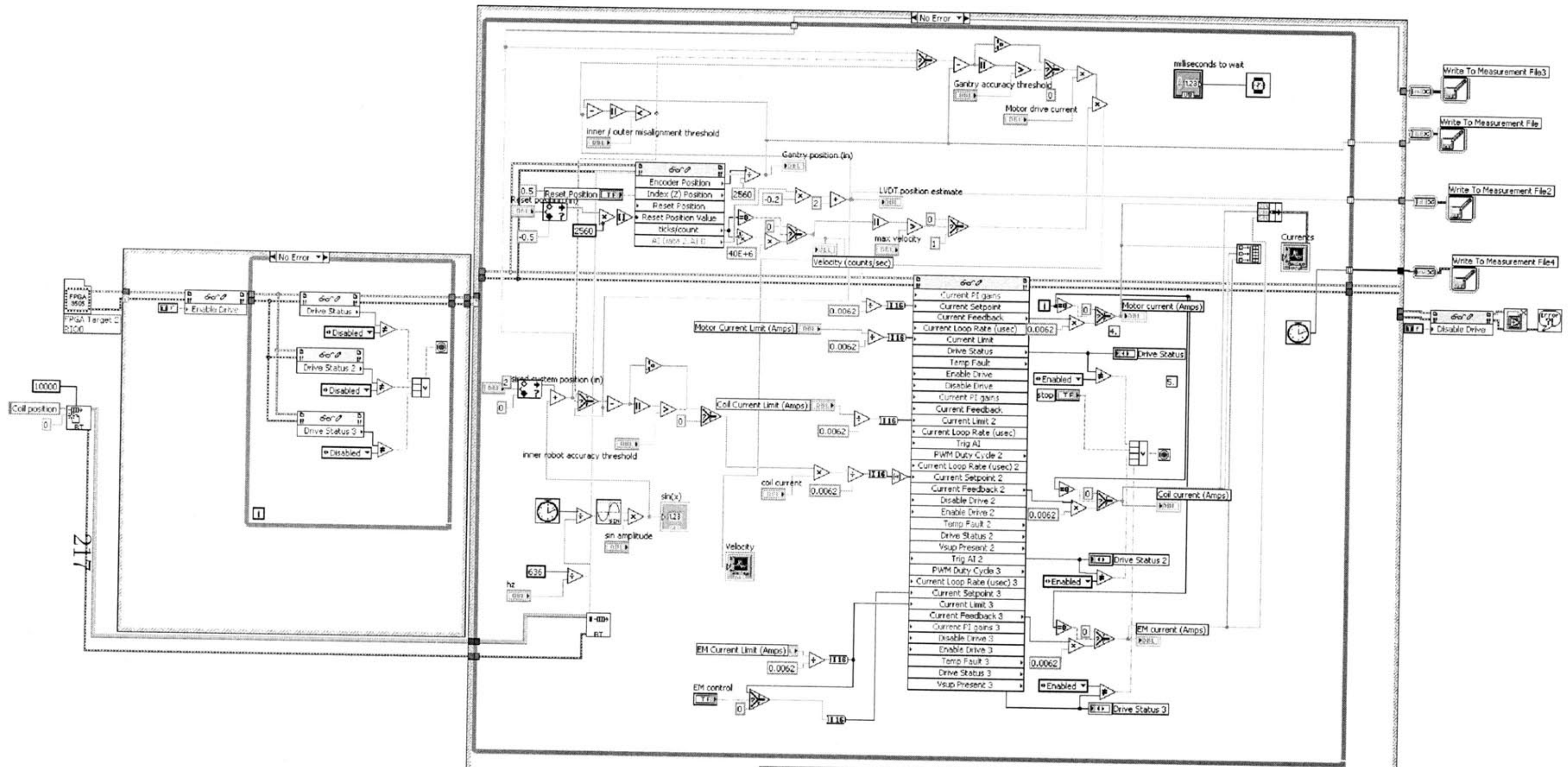
Reset position (in)
0.000

sin(x)
0

sin amplitude
0.25

hz
0.25

Figure D-1: Control loop graphical interface.



1. Convert the setpoint (in amps) to an integer for the FPGA. The conversion factor is 6.2mA/integer value.
2. The direction of the drive is extracted from the sign bit of the setpoint. A positive sign is clockwise, negative is counter clockwise. The integer setpoint sign is set based upon the direction.
3. Convert the current limit (in amps) to an integer for the FPGA.
4. Convert the feedback current from an integer to amps. Return a current of 0 if it is the first iteration to avoid erroneous values.
5. Stop VI execution if 1)Stop button is pressed or 2)the drive is disabled.

If an error occurs while opening a reference to the FPGA, do not execute the VI. If no error occurs, Enable the drive and wait for the drive condition to be anything other than Disabled. Other conditions include Enabled, Drive Fault, and Over Temp Fault. If a Fault condition occurs, the subsequent loop will stop VI execution and indicate the failure in Drive Status.

Figure D-2: Control loop block diagram

Bibliography

[1] Allen, P K. "Automated tracking and grasping of a moving object with a robotichand-eye system." IEEE transactions on robotics and automation 9.2 (1993):152.

[2] Berkelman, P. "Single magnet levitation by repulsion using a planar coil array." Control Applications, 2008 IEEE International Conference on. (2008). :108.

[3] Deyle, T. "A foveated passive UHF RFID system for mobile manipulation." Intelligent Robots and Systems, 2008 IEEE/RSJ International Conference on. (2008). :3711.

[4] Edelman, S. "Computational theories of object recognition." Psychological science 8:135.

[5] Lowe, D G. "Object recognition from local scale-invariant features." Proceedings of the Seventh IEEE International Conference on Computer Vision. (1999). :1150.

[6] Maruyama, K. "3D Object Localization Based on Occluding Contour Using STL CAD Model." Pattern Recognition, 2008 19th International Conference on. (2008). :1.

[7] Oh, S R. "Precision assembly with a magnetically levitated wrist." [1993] Proceedings IEEE International Conference on Robotics and Automation. (1993). :127.

[8] Qiu, R G, andQIU. "RFID-enabled automation in support of factory integration." Robotics and computer-integrated manufacturing 23.6 (2007):677.

[9] Roberts, C M, andROBERTS. "Radio frequency identification (RFID)." Computers & security 25.1 (2006):18.

[10] Roh, S. "Object Recognition Using 3D tag-based RFID System." 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems. (2006). :5725.

[11] Wang, C, and ChonggangWang. "On object identification reliability using RFID." Communications and Networking in China, 2008 Third International Conference on. (2008). :768.

[12] Werfel, J. "Distributed construction by mobile robots with enhanced building blocks." IEEE International Conference on Robotics & Automation (ICRA) (2006):2787