Final Report:
Lloyd's Lethal Left: The Robotic Hand and Control Glove

Jeffrey Caley (BSCE)
Daniel McDonald (BSCS)
Matthew Wuerffel (BSCE, BSCS)

Faculty Mentor: Dr. David A. Wolff

CSCE 499
May 23, 2007

Abstract

The goal of this project was to mimic human motion with a robotic hand. We wanted to achieve a one-to-one mapping of a user's hand movements to a robotic hand. To that end, we designed and implemented a control glove which senses finger movements as well as a servo-driven robotic hand. The input from the sensor glove is converted to a digital signal that is then sent to a control program via a serial interface. The control program performs all relevant calculations and outputs commands to the microcontroller, which sends signals to the servos that control the joints on the robotic hand. The control program also displays relevant information on a GUI including the current reading from all the sensors. The control program has a calibration, motion-recording, and motion-playback functions that are accessible via the GUI, as well as a rock, paper, scissors game.

Acknowledgments

We would like to acknowledge the following people, without whom this project would never have been possible:

- Lloyd Caley, for funding this project.
- Bob Powell, for help with design and for machining the entire hand.
- Chuck Caley, for help with all aspects of the mechanical design.
- Andy Caley, for general help and crazy hair.
- Dr. Wolff, for all the little things.
- Friends, family, and Rachel, for support and brownies.
- Tosh, for being here in spirit.
- Team Saybot, for sharing a capstone room with us.
- All the friendly folks at Hobby Town USA, for advice regarding all aspects of the mechanical construction of the hand and glove.

Table of Contents

List of Figures

1 Introduction

1.1 Project Overview

This project is a foray into the fields of embedded systems, bioengineering and prosthetics, and humanoid robotic design. There are many possible applications of our project, ranging from the manipulation of hazardous materials from a remote location to the simulation of human motion in powered prosthetic devices.

There has been some fascinating research and advancement in the field of prosthetics in recent years. Scientists based in the Rehabilitation Institute of Chicago have helped a former US Marine become the world's first bionic woman. By rerouting shoulder nerves into healthy chest muscle, scientists have enabled Claudia Mitchell to control her bionic arm simply through thought. She is also capable of detecting heat and pressure in the prosthetic limb. Mitchell joins five other amputees that have undergone muscle re-innervation to control their bionic limbs - and all but one have been successful in achieving thought-controlled movement.[i]

Obviously, the bionic arms constructed at the Rehabilitation Institute are far more complex than our robotic hand and control glove. However, the basic challenges that face the two projects are the same: How can we imitate human motion? Why is it so difficult to replicate? How can we translate input from various sensors into the motion of a mechanical hand?

The purpose of this project has been stated above, in the abstract, but we will be more explicit here. We have constructed three distinct items: (1) a robotic imitation of the human hand made out of machined aluminum, consisting of 8 independent joints and servo motors. (2) A glove with embedded sensors for input, consisting of 8 linear potentiometers for monitoring independent voltages at joint locations. (3) A software program that interfaces between the two, allowing a user to place the sensor glove on their hand and have the robotic hand imitate the motion of the user. The software program also includes a GUI that facilitates the testing of the robotic hand, as well as a calibration system that will allow the sensor glove to be used by different individuals. Additionally, the software program has motion-recording and playback functionality, and a rock, paper, scissors game.

For information regarding how we will categorize levels of success for this project, see Section 2.2.

2 Objectives

2.1 Functional and Learning Objectives

There are two categories of goals for this project: functional objectives and learning objectives. Each category is equally important to the success of this project as a whole.

Functional objectives:

- To create a glove that records the movements of the human hand, specifically the movement of the four fingers, through the use of potentiometers
- To create a robotic hand that is capable of recreating the basic movements of a human hand
- To create an interface that allows the input from the glove to control the movement of the robotic hand
- To create a program for the calibration of the glove

Learning objectives:

- To work with embedded systems and microprocessors
- To enhance our knowledge of serial communications
- To learn more about the use of electronically enhanced prosthetics and bio-engineering
- To improve our ability to integrate hardware and software components into a seamless system
- To be able to work with different levels of abstraction (Java vs. assembly)

2.2 Metrics of Project Success

In our requirements document, we defined relative levels of success for the project. The definitions have been modified to take into account design changes.

A partial success would be if a rapid prototype with more than one finger is completed and fully functional, or if the entire glove or entire robotic hand is completed and demonstrates full functionality.

A full success would be if the entire glove and robotic hand are completed and fully functional, and there is a software application available to interface the two with ease.

With respect to the learning objectives, a partial success would be successful communications for the core control program of our project. Complete success would be characterized by successful integration of assembly and Java throughout the system, as well as some knowledge of recent developments in the fields of robotics, prosthetics, and embedded systems.

3 Assumptions and Limitations

3.1 Assumptions

We didn't make any assumptions, because we all know what happens when you assume.

3.2 Limitations

3.2.1 The Sensor Glove

The major problem encountered with the glove was its ability to get accurate readings. The glove needed to be tight fitting and have very little stretch in the material. Due to the tight fitting needs, hand size became a big variable. With people that had larger or smaller hands, the glove would not give proper readings. The glove also is very stiff and restricts movement of fingers. Because of this, moving your fingers into certain positions becomes very difficult.

3.2.2 The Robotic Hand

Finger joints sticking: Friction between the joints caused the hand to stick in certain positions. This made it so the elastic tendon cable was not strong enough to pull the fingers back to resting position.

Durability: The hand has many moving parts and some of the pieces experienced mechanical failure due to overuse. This included the control wires snapping and the JB Weld holding on the finger pads breaking.

3.2.3 The Development Board

While documentation for the HCS12 was phenomenal, documentation for the Dragon12 development board was difficult to find and generally not very helpful. This led to issues with the SCI block as well as some wiring problems.

3.2.4 The Embedded Program

Originally, we wanted to write the embedded program in C, which would have given us much more flexibility. It became clear relatively quickly that this would not be possible based on our level of experience with the language. As a result, the entire embedded program is written in assembly. This means that some parts had to be hard coded which is not the most elegant of solutions. Also, working in assembly meant that creating data packets would be much more difficult, so we had to use single byte communication.

3.2.5 The Control Program

The largest limitation as far as the software was concerned was the protocol we used for serial communication. The idea was to send a byte after you receive a byte. This means that once the first byte is sent, the communication will continue indefinitely until manually stopped. While this proved to work correctly, many workarounds were needed on the software side to ensure this protocol worked properly.

This also led to problems with transfer speed. While our transfer rate was slow, it was fast enough to give real-time mapping of glove and hand, but the rate was not consistent. Some days it would transfer around 100 bytes per second, other times it would be as low as 50. The inconsistency didn't show a considerable effect, but it did become noticeable at extremes, and it seems like a problem that could have been fixed if an interrupt-driven system had been in place.

The major drawback with this inconsistency is that since all components rely on event-handling, processes will act differently with different transfer speeds. For instance, if a user is recording a movement when the transfer rate is 50 bytes/sec, then plays the recording back at 100 bytes/sec, then the playback will be twice as fast as the recording because it is sending data at twice the rate.

The decision to use event-handling for managing data also had its drawbacks, especially when it came to refreshing graphical components. Essentially, every time a byte is received, all components that are listening to events are called to repaint. This means that if the transfer rate increases, then the burden on the CPU increases as well. As far as graphics refreshing goes, this could have been done at a constant rate, and the event-handling didn't necessarily need to rely on every received byte.

As far as the software code is concerned, there were limitations with efficiency. The Java Virtual Machine doesn't maximize the use of the CPU as well as native languages, and the rendering of components didn't seem to utilize the hardware acceleration, so there were problems with performance in reference to rendering components. This was an unforeseen problem: I didn't expect to monopolize the CPU or in any way come close to exhausting its resources. There were measures that could have been taken to make the Java code more efficient, but ideally to increase the performance it would have useful to utilize the graphics hardware acceleration.

3.2.6 Cost

The physical construction of the hand was very expensive, and increased in cost with every additional cut to each piece being made. Because of this, a more complicated initial design had to be discarded.

4 Design Methodology and Implementation

4.1 Research Review

4.1.1 Sensor Glove

Initially, we looked into using a VR glove to capture the motion of the user's hand. However, the price for even the most basic VR glove was prohibitive, and there was no guarantee that it would have the functionality that we need. We decided to construct a simple sensor glove using an array of high-resistance linear potentiometers to monitor the position of 8 joints on the human hand (2 on each finger).

4.1.2 Microcontroller and Dragon12 Development Board

A microcontroller is the logical choice for interfacing the main program, the servo motors, and the input gloves. Two of our group members have experience with the Motorola HCS12 and the Dragon12 development board, so we decided to use it for the project. The HCS12 has two analog to digital blocks, a pulse-width modulation block, a serial communications interface, and other powerful features. The Dragon12 can be worked on through ASMIDE, which allows for programming the chip through the serial port and has a virtual debugging mode in software. The labs from the CSCE480 class, written by Jake Nelson and Tosh Kakar, proved to be a great resource for all things related to the Dragon12 and the HCS12 microcontroller.[ii]

4.1.3 Robotic Hand

The robotic hand is made out of machined aluminum, and is powered by 8 Hitec-475HB servos. We chose the Hitec-475HB servos because of their low cost, high availability, and high stall torque.

4.1.4 Command Program with GUI

We examined the possibility of creating a driver to communicate to the serial port using Windows XP. The driver would have been coded in C++. However, we found that Java has an extension library called Javax.Comm, the Java Communications package, which has an excellent library designed to communicate with the serial port. We decided to implement the control program in Java on a Linux based PC.

4.2 Theory and Implementation of Hardware Operation

4.2.1 Basic Design

A sensor glove is fitted with linear potentiometers. The potentiometers output voltages representing the position of each joint. These voltages are sent into the ATD block of the Dragon12 board, and converted into digital numbers. These numbers are sent to the control program, which updates the GUI display and then performs calculations on the incoming data to determine the commands that need to be sent to the servos. These commands are sent to the Dragon12 board, which uses the PWM block to send pulses out on specific pins that control individual servos. The servos are attached to a robotic hand that is mounted to a base unit.

4.2.2 Hand Design

At resting position, all the fingers of the hand are straight. When a servo is told to move by the computer, it starts to rotate. The rotation of the servo spools in control wire. There are two sets of four control wires. The four that are attached to the palm side servos control the metacarpal joints. The four that are attached to the back-of-the-hand side control the distal and middle joints. These two joints move together, as they are controlled by the same wire. When the servos spool in control wire, the joint that corresponds with that wire constricts. The constriction causes the elastic tendon cable to lengthen, creating more tension. The servo will then hold this joint in place until the joint needs to be relaxed. When this happens, the servo spools out wire and the elastic tendon cable pulls the figure back towards its resting position. Figure 1 shows the robotic hand.

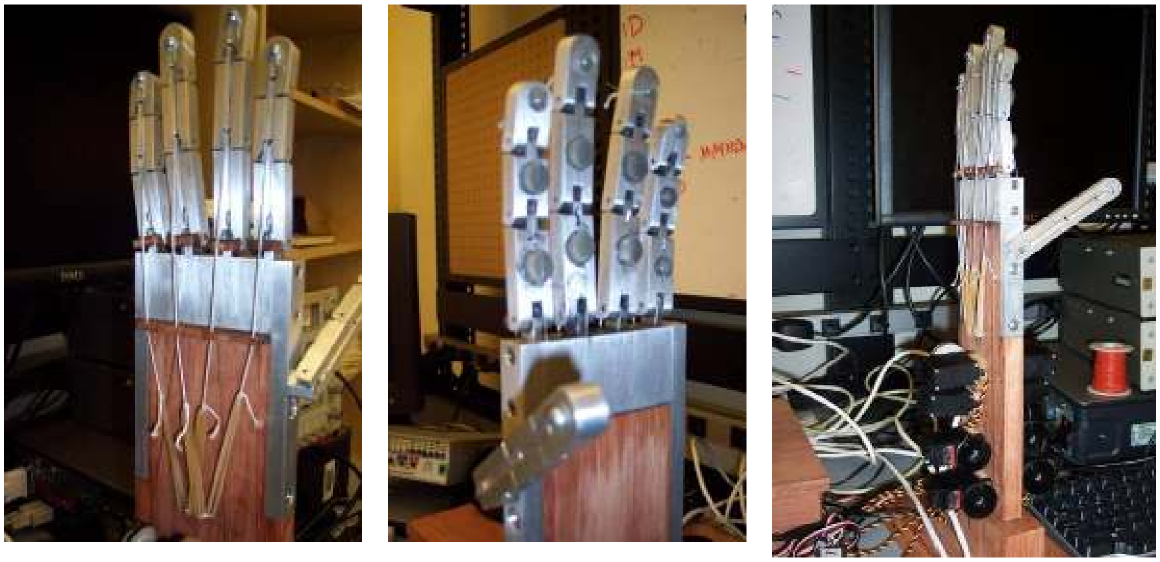


Figure 1: The robotic hand

4.2.3 Glove Design

The sensor glove is fitted with eight linear motion potentiometers. Each potentiometer is spring loaded into a rest position. Two position strings run down the top of each of the fingers on the glove. One position string is attached to the tip of the distal joint, with the other end attached to a potentiometer. The other position string on that finger is attached to the metacarpal joint, with the other attached to a different potentiometer. This means two potentiometers per finger. As a finger is bent, these position strings move the potentiometers to a spot corresponding to the position of the finger. When the finger is moved back out of this flexed position, the springs pull the potentiometers back accordingly. Figure 2 shows the sensor glove.

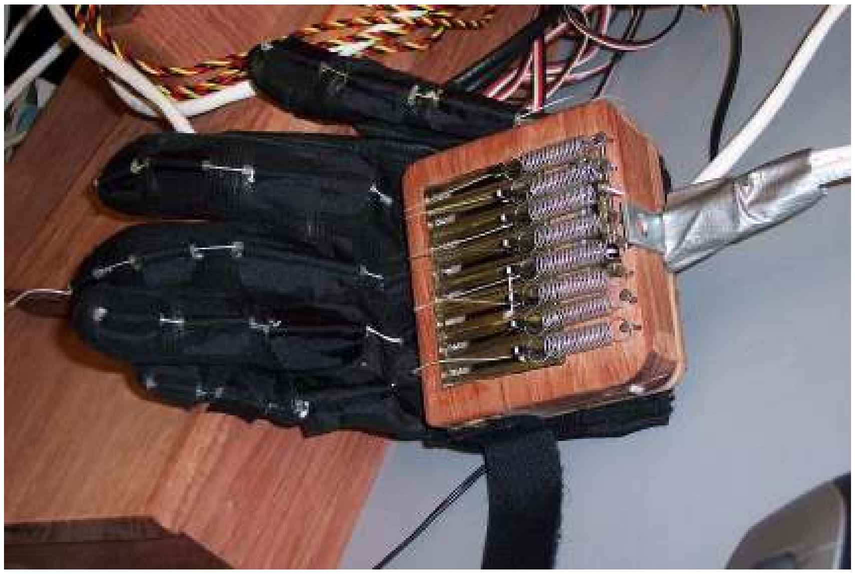


Figure 2: The sensor glove

4.3 Block Diagram of System and Modules

4.3.1 Robotic Hand

The construction of the robotic hand took place in a machine shop in Toledo Washington. We designed each of the 19 aluminum pieces and then schematics of each were sent off to be built. After the pieces were machined we assembled the hand itself. Once the aluminum part of the hand was done, attachments had to be made to make the hand functional. First, finger pads were added. Finger pads are hard plastic cylinders with a groove cut down the middle. The groove allows for an aluminum tube to run down the middle of it to give a place for the control wire to run. Finger pads were attached using JB Weld. Next, torque bridges were constructed and attached. To add more room to attach servos, a long wooden wrist was added to the hand. This gave us places to attach servos and also a wire guide. Finally, two I-bolts were added to hold rubber bands for the elastic tendon cables.

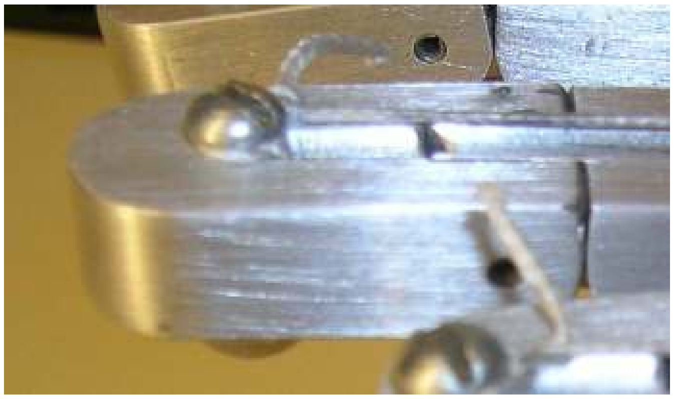


Figure 3: The distal phalanx bone

Figure 3 is the distal phalanx bone. It contains a female interphalangeal joint. This joint connects to a middle phalanx bone. This is one of three bones that will make up an entire finger. Four fingers will then make up the robotic hand. The hole that goes through the middle of the bone is threaded and has two bolts attached. One bolt is used to secure the control wire to the palm side of the bone, while the other is screwed into the top of the bone through the same hole. This bolt is used to attach the elastic tendon cable to the bone. The elastic tendon cable is used to move the fingers back to resting position.

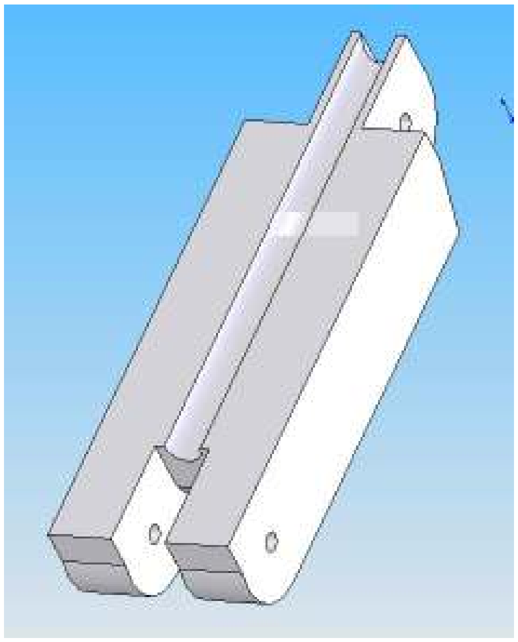


Figure 4: The middle phalanx bone

Figure 4 is the middle phalanx bone. It has a female and male interphalangeal joint. The male interphalangeal joint is attached to a distal phalanx bone and the female interphalangeal joint is attached to a proximal phalanx bone. It has a finger pad (see

Figure 31) attached to the palm side of the bone. The finger pad has a metal tunnel that travels through it that is used as a passage way for a control wire. The back side of the proximal bone is grooved to give a place for the elastic tendon cable to travel.

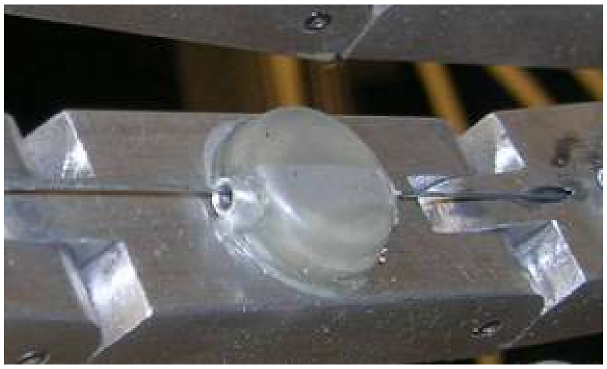


Figure 5: A finger pad

Figure 5 shows a finger pad. Finger pads supply a tunnel for the control wires to travel down. This allows for the movement of the hand. They are made from a hard plastic that has a aluminum tube that runs down the center of it. They are JB Welded to the center of each finger bone on the palm side.

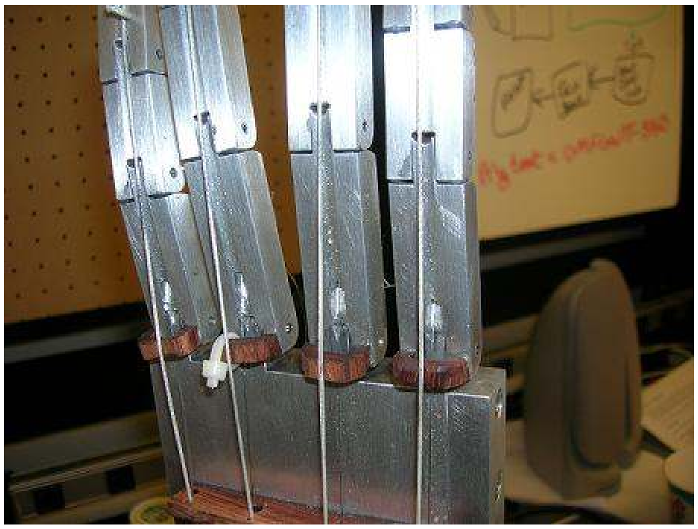


Figure 6: Proximal phalanx bones

Figure 6 shows several proximal phalanx bones. A single bone contains a female and male interphalangeal joint. The male interphalangeal joint attaches to a middle phalanx bone while the female interphalangeal joint attaches to one of the palms male interphalangeal joints. The proximal phalanx bone contains a finger pad that is used to attach a control wire too. It also has a whole drilled diagonally through it (not pictured). This allows for a control wire that attaches to a distal bone to start on the top of the bone, travel diagonally through the bone and come out on the palm side of the bone. This is used to manipulate the distal and middle bone positions without effecting the proximal phalanx bone's position.

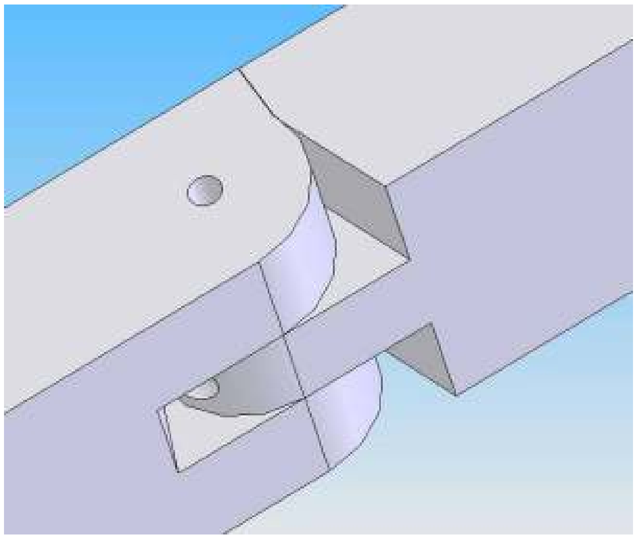


Figure 7: Male and female interphalangeal joints

Figure 7 is a male and female interphalangeal joint. Each joint was designed for 120 degrees of movement. When perfectly relaxed, the joint will be at 180 degrees. When completely flexed, the joint will be at 60 degrees. Due to the machining process, the physical robotic hand only has about 90 degrees of movement. The joints are fastened together used a 1/16in. coder pin.

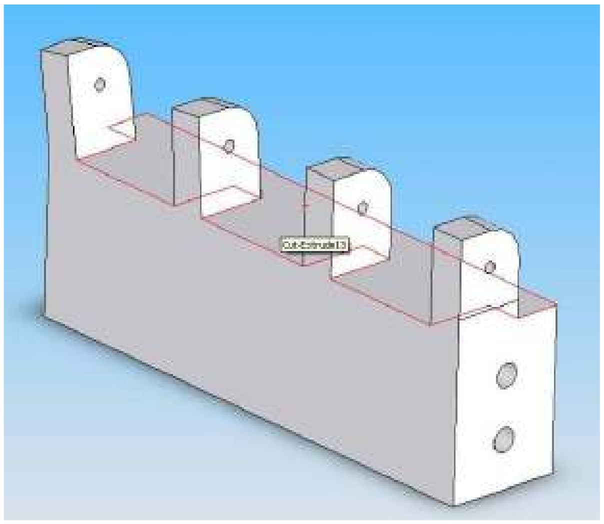


Figure 8: The palm

Figure 8 is the palm. The palm is a piece of machined aluminum with four male interphalangeal joints. Each of these joints is attached to a proximal phalanx bone. Attached to the other side of the wrist is a wooden extension making room for servos.

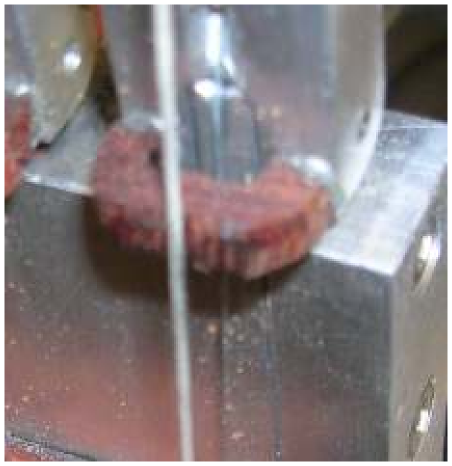


Figure 9: The torque bridge

Figure 9 is a torque bridge. Due to the large amount of torque needed to lift an entire finger from the flexed position, the torque bridges were added to the wrist end of the proximal phalanx bone. This bridge allows the elastic tendon cable to move farther away from the rotation point of the joint and therefore giving the rubber bands more torque to move the joint.

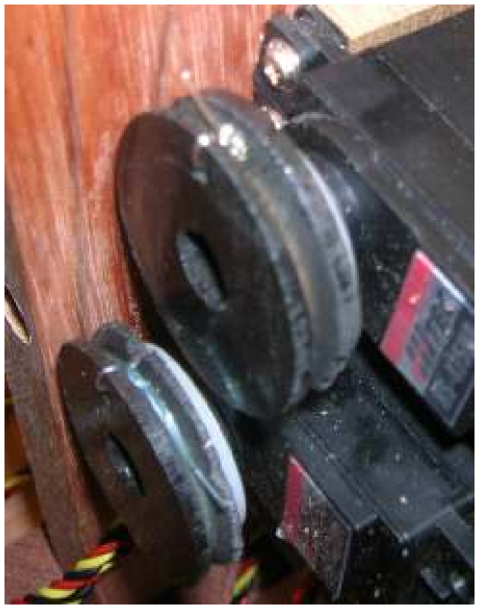


Figure 10: A servo with spool attached

Figure 10 is a servo with a spool attached. This servo rotates 180 degrees, spooling wire in and out to control the motion of each finger. Each servo is responsible for spooling one control wire. The control program is responsible for controlling the full range of motion, making sure not to spool beyond the bounds of the construction.

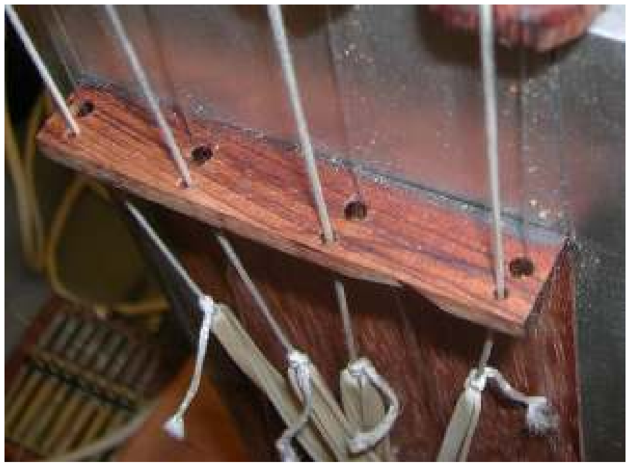


Figure 11: The wire guide

Figure 11 is the wire guide. The wire guide is attached to the wrist and has eight holes drilled through it. Four holes are used to direct control wires while the other four holes are used to direct the elastic tendon cables.

4.3.2 Sensor Glove

The controlling glove is a converted baseball batters glove. Plastic tubes were sewn, glued and then taped to each finger knuckle to give a place for the position strings to travel through. The potentiometer block was then fastened to the top of the hand using Velcro strips. A strap for stiffening the potentiometer block to the hand was also added.

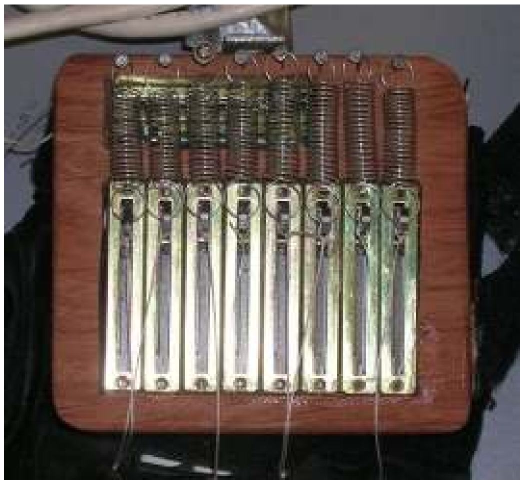


Figure 12: The potentiometer block

Figure 12 is the potentiometer block that is mounted on the sensor glove. Eight potentiometers are placed parallel to each other, with another two running perpendicular to them under the springs. Springs are attached to the potentiometers slider and hold the slider to the back end of the glove when no force is applied. Each slider also has a position string attached that connects a specific joint on each finger.

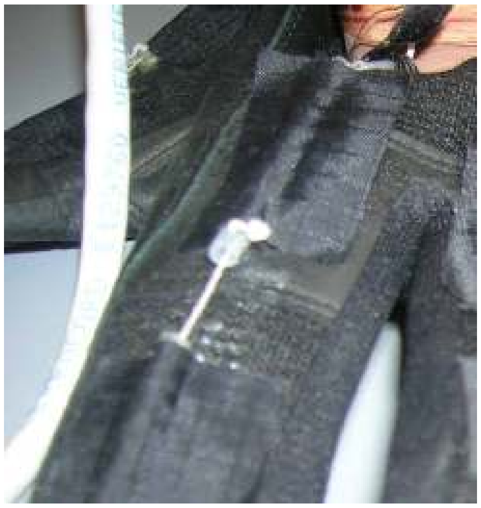


Figure 13: The glove tunnel

Figure 13 is a glove tunnel. The glove tunnels are for the position strings to run down. They keep the position strings on the top of each finger and allow for more accurate readings.

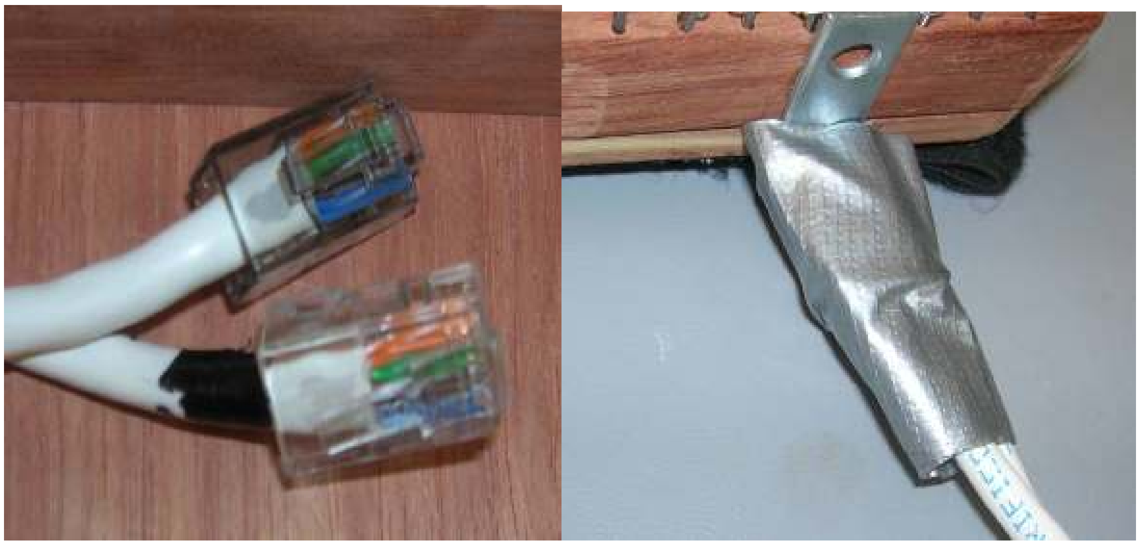


Figure 14: The signal cable

Figure 14 is the cable that carries the signal from the potentiometers to the development board. There are two CAT-5 cables that come out of the glove and are plugged into the base of the hand. Each CAT-5 cables contain 8 single strand wires. Each potentiometer requires a power, ground and an output wire. The Blackened cable supplies power and ground while the other cable supplies an output wire for each of the eight potentiometers. The other end is fitted into CAT-5 cable plugs and plugged into ports that are connected to the development board.

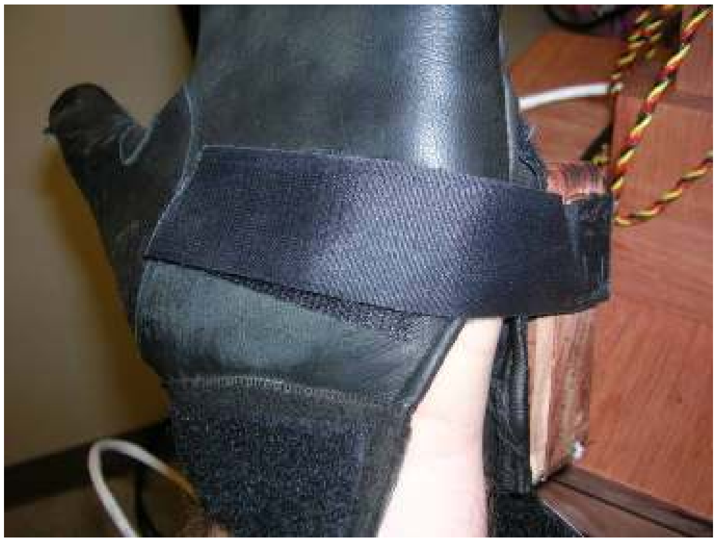


Figure 15: The stiffening strap

Figure 15 is the stiffening strap. This strap is used to secure the glove and potentiometer to the hand. The strap is attached to the block and comes around and fastens to the palm side of the hand using Velcro. This is used to help get more accurate readings from the potentiometers

4.3.3 Analog to Digital Block

The HCS12 has a powerful analog to digital conversion tool, located in the ATD 10B8C block. The block has 8 analog inputs, a sample and hold mechanism, a successive approximation register, and a clock prescaler, which make it ideal for our application. Figure 16 shows the ATD 10B8C block diagram.

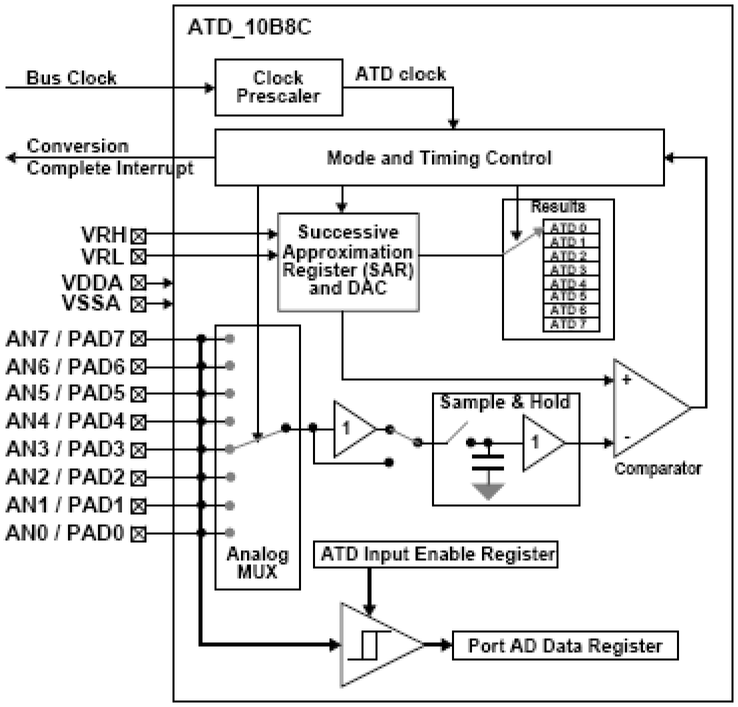


Figure 16: The ATD block[iii]

There are four registers that need to be set up to utilize the ATD block: ATD Control Register 2, 3, 4, and 5 (abbreviated ATDCTL2, ATDCTL3 etc). The documentation available on Freescale's website does an excellent job of describing the registers in detail, so we will only touch on the things that are relevant to this project. Figure 17 shows the ATDCTL2.

	7	6	5	4	3	2	1	0
R	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF
W								▨
RESET:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 17: ATD Control Register 2 (ATDCTL2)[iv]

The ADPU (ATD Power Up) is the only bit that we are interested in here. It toggles the ATD block on and off; a 1 corresponds to normal operation, while a 0 powers down the ATD block to conserve power.

Figure 18 shows the second register that we are interested in: ATDCTL3.

	7	6	5	4	3	2	1	0
R	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
W	▨							
RESET:	0	0	1	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 18: ATD Control Register 3 (ATDCTL3)[v]

Our primary interest in ATDCTL3 is to control the conversion sequence length of the ATD block. Using the data from the table shown below in Figure 19, we can manually set the number of conversions that will take place per sequence.

S8C	S4C	S2C	S1C	Number of Conversions per Sequence
0	0	0	0	8
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	X	X	X	8

Figure 19: Conversion Sequence Length Coding[vi]

Although each ATD block has 8 inputs, some input channels on the Dragon12 are reserved, so we have to use both of the ATD blocks to convert all 8 of our inputs. The ATD0 block will be performing 6 conversions per sequence, and the ATD1 block will be performing 2 conversions per sequence.

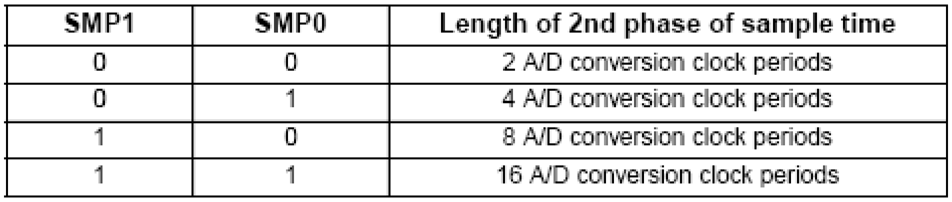
ATDCTL4, shown in Figure 20, controls the resolution, length of sample time, and the ATD clock prescaler select.

	7	6	5	4	3	2	1	0
R / W	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
RESET:	0	0	0	0	0	1	0	1

▨ = Unimplemented or Reserved

Figure 20: ATD Control Register 4 (ATDCTL4)[vii]

SRES8 (A/D Resolution Select) controls the resolution of the ATD conversion results; it can be set for 8 or 10 bit precision (1 = 8, 0 = 10). In our case, we will be using 8 bit resolution. The next bits of interest are SMP1 and SMP0 (Sample Time Select). These two bits allow us to specify how many conversion clock periods we want to pass while the analog signal is stored, as shown in Figure 21.

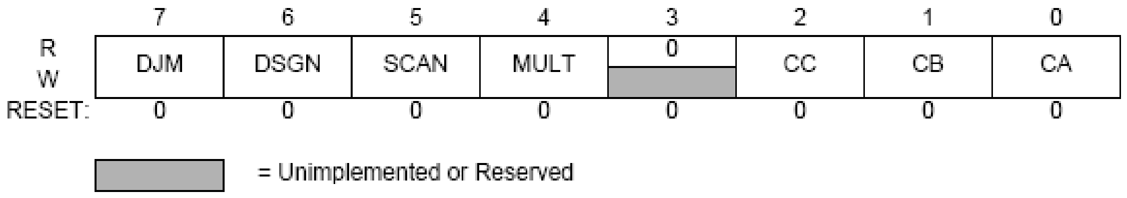


SMP1	SMP0	Length of 2nd phase of sample time
0	0	2 A/D conversion clock periods
0	1	4 A/D conversion clock periods
1	0	8 A/D conversion clock periods
1	1	16 A/D conversion clock periods

Figure 21: Sample Time Select[viii]

Since we want to minimize servo stutter due to inaccurate data, we use 16 A/D clock periods for the 2nd phase of sample time. Of course, the actual amount of time that this corresponds to is dependent on the ATD conversion clock frequency. This frequency can be manually set through the manipulation of the PRS4 - 0 (ATD Clock Prescaler) bits. The ATD conversion clock frequency can vary from 500KHz to 2MHz. We use a 1MHz clock frequency for the glove inputs, which means we need to divide the board's bus clock by 24. This corresponds to a prescale value of 01011, which is placed into the PRS4 – 0 bits.

The final ATD register that we are interested in is ATD Control Register 5, which is shown in Figure 22. ATDCTL5 controls which channels are sampled, how the results are stored, and whether the results are signed or unsigned.



	7	6	5	4	3	2	1	0
R	DJM	DSGN	SCAN	MULT	0	CC	CB	CA
W					▨			
RESET:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

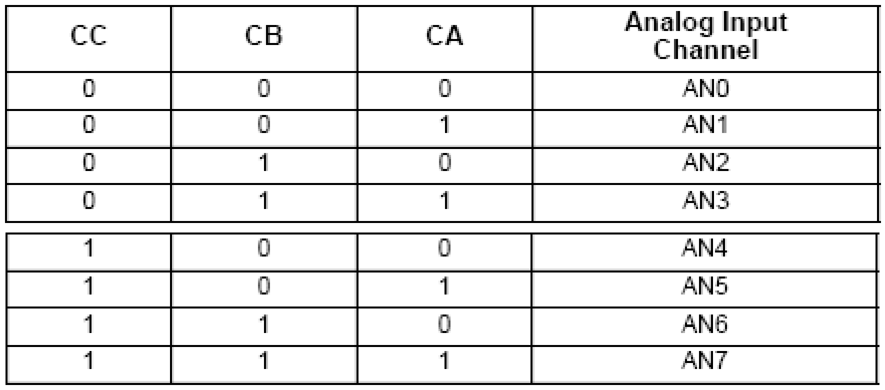
Figure 22: ATD Control Register 5 (ATDCTL5)[ix]

The DJM (Result Register Data Justification) bit toggles between left and right justified results. The default is left justified, which is what we want. The DSGN (Result Register Data Signed or Unsigned Representation) bit toggles between signed and unsigned conversion, and like the DJM bit, is written with a '0' on reset. We want unsigned conversion for our sensor glove, so this is not a problem.

Since we want to constantly be updating our sensor information, we need to have the ATD working continuously. This is accomplished by writing a '1' to the SCAN (Continuous Conversion Sequence Mode) bit.

The next bit of interest is the MULT (Multi-Channel Sample Mode) bit, which selects between sampling across multiple channels or sampling from a single channel. We need 8 inputs for the eight joints that we are monitoring, so we need to write a '1' to the MULT bit.

To use multi-channel scanning, we must designate which input channel we will be sampling from first. Then the hardware will automatically cycle through the number of channels specified by the conversion sequence length in ATDCTL3. To first channel is specified by writing to the CC, CB, and CA (Analog Input Channel Select Code) bits. The options are shown below in Figure 23.



CC	CB	CA	Analog Input Channel
0	0	0	AN0
0	0	1	AN1
0	1	0	AN2
0	1	1	AN3
1	0	0	AN4
1	0	1	AN5
1	1	0	AN6
1	1	1	AN7

Figure 23: Analog Input Channel Select Coding[x]

We will be sampling 6 channels on ATD0 and 2 channels on ATD1, so we need to start at AN2 (since some of the channels are reserved, as mentioned earlier). From the table above, we can see that we have to write '011' to the Analog Input Channel Select Code bits.

The code segment shown below sets up all the ATD registers as described above.

```
; Set up the ATD converter
MOVB    #$80,atd0ctl2    ; power up ATD0
MOVB    #$30,atd0ctl3    ; 6 conversions per sequence
MOVB    #$EB,atd0ctl4    ; 8 bit resolution, 16
                         ; periods / conversion
                         ; divide bus by 24 (1 MHz)

MOVB    #$32,atd0ctl5    ; left-justified, unsigned,
                         ;continuous multi-channel,
                         ;channel 2 -> 3 -> 4 -> 5 -> 6 ->
7
.
.
```

```
.
LDAA    ATD0DR0         ; load atd result reg 0
```

4.3.4 Pulse Width Modulation Block

The HCS12 has a PWM block that has 8 independent output channels with programmable period and duty cycle, dedicated counters, 8 8-bit channel or 4 16-bit channel resolution, and multiple clock sources and clock select logic. These powerful tools allow us to easily output waveforms to control the servos that will drive the mechanical hand. The PWM 8B8C block diagram is shown below in Figure 24.

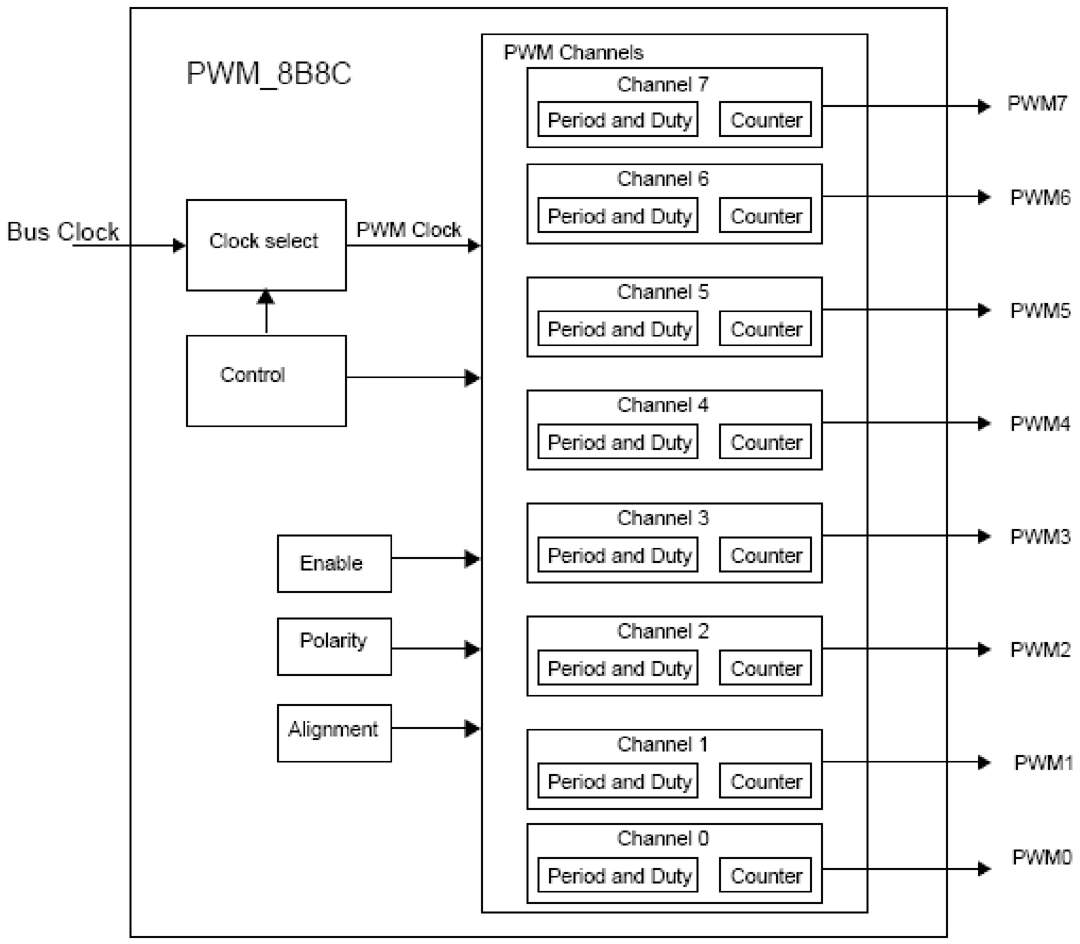
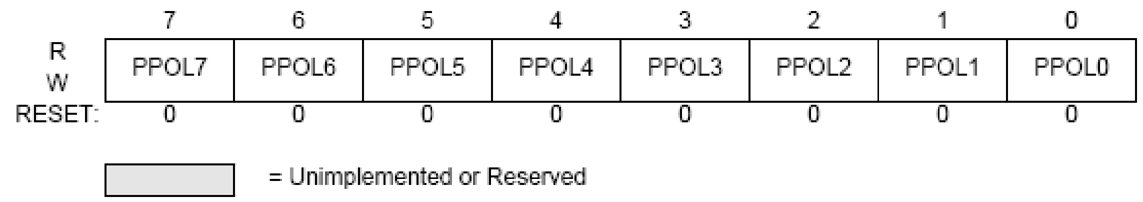


Figure 24: The PWM Block[xi]

To output servo control signals for the 8 servos that power the robotic hand, it is necessary to set up the PWM block by writing various values to the control registers.

To drive the servos, we need to set the polarity of the output channels. The PWM Polarity Register (abbreviated PWMPOL), is shown in Figure 25.



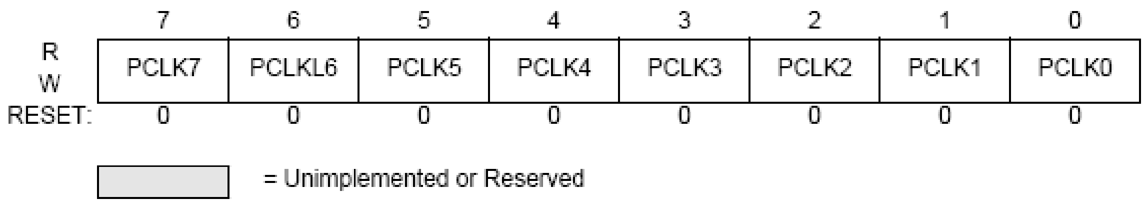
	7	6	5	4	3	2	1	0
R W	PPOL7	PPOL6	PPOL5	PPOL4	PPOL3	PPOL2	PPOL1	PPOL0
RESET:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 25: PWM Polarity Register (PWMPOL)[xii]

The bits in the PWMPOL toggle between a high-to-low or low-to-high transition for each PWM output. Signals are low-to-high on reset, which won't work to drive the servos. Instead, we need to pulse high at the start of the waveform and transition to low when the duty cycle is reached. This is achieved by writing a '1' to each bit in the PWMPOL.

As mentioned previously, the PWM block has multiple clock sources and clock select logic. Both of these features must be configured to drive the servos. The relevant registers are the PWM Scale Registers (PWMSCLA and PWMSCLAB), as well as the PWM Prescale Clock Select Register (PWMPRCLK) and the PWM Clock Select Register (PWMCLK). The PWMCLK can be seen in Figure 26.



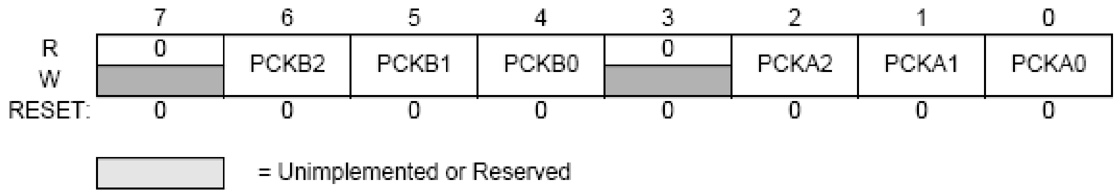
	7	6	5	4	3	2	1	0
R W	PCLK7	PCLKL6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0
RESET:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 26: PWM Clock Select Register (PWMCLK)[xiii]

A '0' corresponds to clock A or B, while a '1' corresponds to using clock source SA and SB. We need to use the scaled clock sources to generate waveforms with the correct period to drive the servos. This means we need to write '1's to all of the bits in PWMCLK.

The PWM Prescale Clock Select Register (PWMPRCLK) is shown in Figure 27.



	7	6	5	4	3	2	1	0
R	0	PCKB2	PCKB1	PCKB0	0	PCKA2	PCKA1	PCKA0
W	▨				▨			
RESET:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 27: PWM Prescale Clock Select Register (PWMPRCLK)[xiv]

Bits PCKA2, PCKA1, and PCKA0 (Prescaler Select for Clock A) control the clock select as shown below in Figure 28.

PCKA2	PCKA1	PCKA0	Value of Clock A
0	0	0	bus clock
0	0	1	bus clock / 2
0	1	0	bus clock / 4
0	1	1	bus clock / 8
1	0	0	bus clock / 16
1	0	1	bus clock / 32
1	1	0	bus clock / 64
1	1	1	bus clock / 128

Figure 28: Prescaler Select for Clock A[xv]

We will use a MOVB command to write $11 to the PWMPRCLK. This, along with our choice of clock scaling for clock A and B, will reduce the bus clock down to the right frequency for driving the servos.

To generate clock SA and SB, we need to manipulate the PWMSCLA and PWMSCLB registers. They are identical in form and function, so we will only look at the PWMSCLA register, shown in Figure 29.

	7	6	5	4	3	2	1	0
R W	Bit 7	6	5	4	3	2	1	Bit 0
RESET:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 29: PWM Scale A Register (PWMSCLA)[xvi]

The amount that clock A is scaled is given by the formula: Clock A / (2 * PWMSCLA). When PWMSCLA is $00, the scale factor is 256, which is the factor that we want, so we will write '0' to PWMSCLA (and PWMSCLB).

To control the period and duty cycle, we must manipulate the PWM Channel Period Registers (PWMPERx) and PWM Channel Duty Registers (PWMDTYx). The form of these registers is identical, and is shown below in Figures 30.

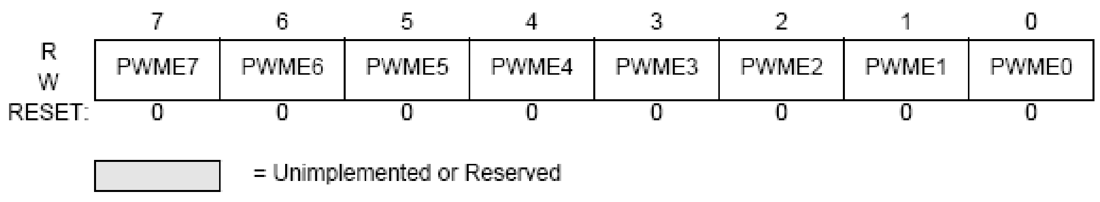
	7	6	5	4	3	2	1	0
R W	Bit 7	6	5	4	3	2	1	Bit 0
RESET:	1	1	1	1	1	1	1	1

▨ = Unimplemented or Reserved

Figure 30: PWM Channel Period Registers (PWMPERx) and PWM Channel Duty Registers (PWMDTYx)[xvii]

By writing to PWMPER0 through PWMPER7 and PWMDTY0 through PWMDTY7, the 8 output channels can be individually customized. However, for our application, we want the period of all the outputs to be the same. Only the duty cycle will change.

Finally, with all the clock and polarity settings finalized, we are ready to enable the channels of the PWM block. This is accomplished by writing to the PWM Enable Register, or PWME, shown in Figure 31.



	7	6	5	4	3	2	1	0
R W	PWME7	PWME6	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0
RESET:	0	0	0	0	0	0	0	0

▭ = Unimplemented or Reserved

Figure 31: PWM Enable Register (PWME)[xviii]

Since we want to use all 8 channels, we simply write $FF to the PWME.

The code below shows how to set up the PWM block as described:

```
MOVB    #$FF,pwmpol      ; pulse high on block 0-7
MOVB    #$11,pwmprclk         ; prescale clock tap
MOVB    #0,pwmscla       ; scale clock by this amount
MOVB    #0,pwmsclb       ; clock b scale select
MOVB    #$FF,pwmclk      ; select prescaled or scaled clock
                              ; for 0 - 5

MOVB    #255,pwmper0     ; set period for channel 0
MOVB    #30,pwmdty0      ; set duty cycle for channel 0
.
.
.
MOVB #$FF,pwme           ; enable channels 0 - 7
```

4.3.5 Serial Communications Interface Block

The final system on the Dragon12 that we are utilizing for this project is the Serial Communications Interface (SCI). *A word of warning: the Dragon12 has two SCI blocks, but SCI1 has only limited functionality when you are working in EVB mode!* This caused a *lot* of trouble and many days of work were spent trouble shooting this aspect of the project. That being said, the SCI block diagram is shown below in Figure 32.

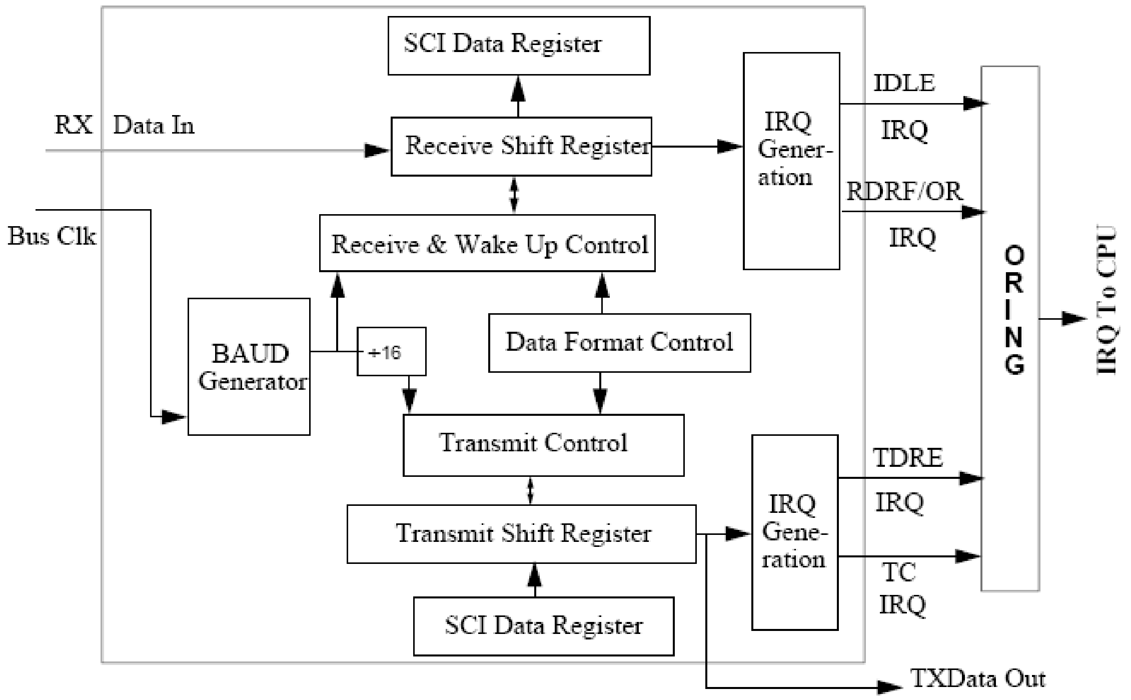


Figure 32: Serial Communications Interface Block (SCI)[xix]

To set up the SCI block, we need to look at a couple of registers: the SCI Baud Rate Registers (SCI BDH/L) and SCI Control Register 2 (SCICR2). The SCI BDH/L registers can be seen in Figure 33.

	7	6	5	4	3	2	1	0
R	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8
W	▨	▨	▨					
RESET:	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

	7	6	5	4	3	2	1	0
R/W	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
RESET:	0	0	0	0	0	1	0	0

▨ = Unimplemented or Reserved

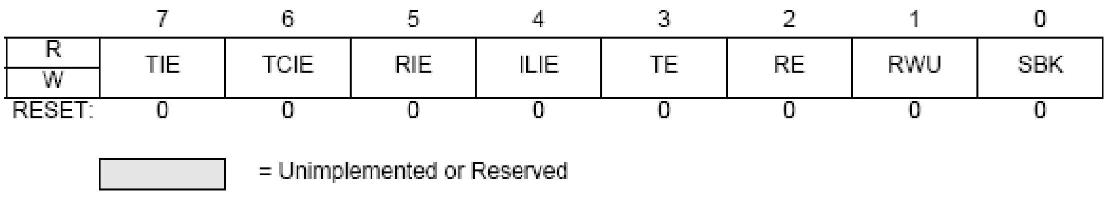
Figure 33: SCI Baud Rate Registers (SCI BDH/L)[xx]

These thirteen bits control the baud rate of the SCI. A couple quick notes about these registers:

- The baud rate generator is disabled unless the TE or RE bit have been set for the first time.
- SCI baud rate = SCI module clock / (16 * BR)

- Writing to SCIBDH has no effect unless you also write to SCIBDL

To generate our baud rate of 9600, we simply write $9C to SCIBDL on block 0. Now all that remains is to enable the transmit and receive functions. This is accomplished by writing to the SCI Control Register 2, which is shown below in Figure 34.



	7	6	5	4	3	2	1	0
R / W	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
RESET:	0	0	0	0	0	0	0	0

▭ = Unimplemented or Reserved

Figure 34: SCI Control Register 2 (SCICR2)[xxi]

To send and receive on the SCI block, we need to write '1's to the Transmitter Enable and Receiver Enable bits (TE and RE, respectively). The code below shows how to set up the SCI block for simple transmission and reception. It is followed by two subroutines based on Huang's HCS12 text that receive or send a single byte via SCI0.

```
MOVB    #$9C,sci0bdl              ; set baud rate to 9600
MOVB    #$0C,sci0cr2              ; enable tx, rx

;PUTC, GETC subroutines based on Huang's HCS12 textbook
code
;Subroutine PUTC2SCI0: outputs a character to SCI0 using
polling method
PUTC2SCI0:
     BRCLR     SCI0SR1,TDRE,*     ; wait for TDRE set
     STAA      SCI0DRL            ; write to SCI0DRL
     RTS

;Subroutine GETCSCI0: reads a character from SCI0 using
polling method
GETCSCI0:
     BRCLR     SCI0SR1,RDRF,*     ; wait for RDRF set
     LDAA      SCI0DRL            ; read the character
     CMPA      #255               ; check for break char
     BNE       pass
     JMP       start
pass RTS
```

4.4 Theory and Implementation of Software Operation

4.4.1 Package logicalComponents

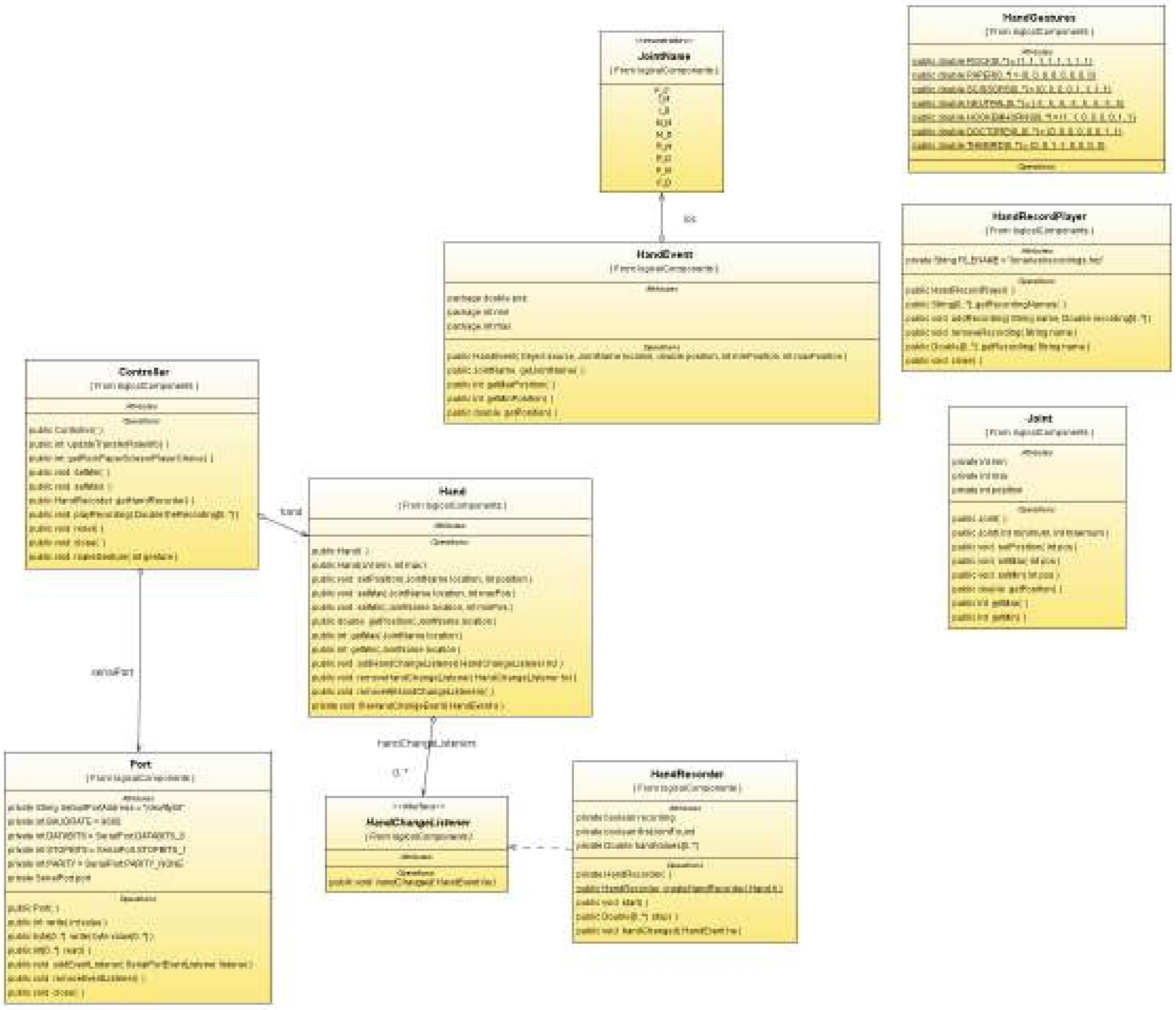


Figure 35: The logicalComponents package

This package contains all classes which deal with logically representing the hand. The smallest unit of the hand for this project is the Joint class, which is contained by the Hand class. There are classes in charge of maintaining position and other information called HandChangeListener and HandEvent, which use the idea of event-handling to update the state of the Hand. The Port class deals with managing serial IO communication. The

JointName class is an enum type which holds custom information about the range of motion of each servomechanism on the physical robotic hand. The classes HandRecordPlayer and HandRecorder are built to manage and save prerecorded hand movements for playback at a later time. The HandGestures class holds different information for joint positions which make up popular gestures with the hand. The Controller class is the boundary class which provides a single entry point to the UI that allows human interaction with the code.

The main purpose of using an event-driven mechanism for updating information was based on the idea of having multiple processes observing the hand at the same time. I wanted to be able to add components which would do a variety of tasks, from displaying graphical renditions of the Hand's state, to recording a history of movement. I also wanted to be able to add and remove these processes on the fly, in the middle of program execution. These requirements lent themselves well for implementing an event-listener interface.

This program was aimed more at abstraction than it was at performance. I wanted to be able to represent each part as an independent but incomplete portion of the whole picture. I used practices which encapsulate data that is not pertinent to other methods.

4.4.2 Joint Class

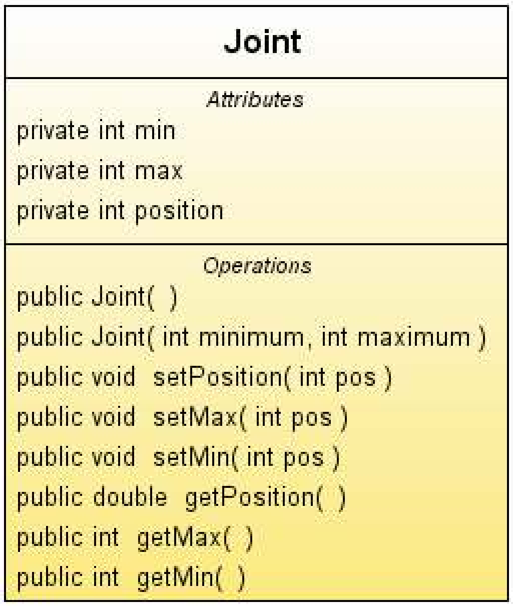

```
Joint
Attributes
private int min
private int max
private int position
Operations
public Joint( )
public Joint( int minimum, int maximum )
public void  setPosition( int pos )
public void  setMax( int pos )
public void  setMin( int pos )
public double  getPosition( )
public int  getMax( )
public int  getMin( )
```

Figure 36: The Joint class

The Joint class (shown in Figure 36) logically represents a single hinged joint with a fixed range of motion. The position of the joint is monitored in relation to a maximum and minimum value, and its position is reported as a value between 0 and Joint.MAX_VALUE.

The maximum value need not be greater than the minimum value in order to record accurate values for the position of a joint. The position of a joint is interpreted to be the percentage of the range from the minimum value to the position in comparison to the total fixed range of motion of the joint. This percentage is then multiplied by the maximum value to represent a point in a fixed range regardless of the limitation of the actual maximum and minimum values.

Some important methods to note in this class:

setPosition

Sets the position of this Joint to the specified value, if the argument is within the range of the Joint's minimum and maximum values.

setMax

Sets the maximum range of motion value for this Joint. If the max position is the same as the min position, then the min position is decremented by 1 so as to prevent the joint from appearing fixed. This property has no use for this project.

setMin

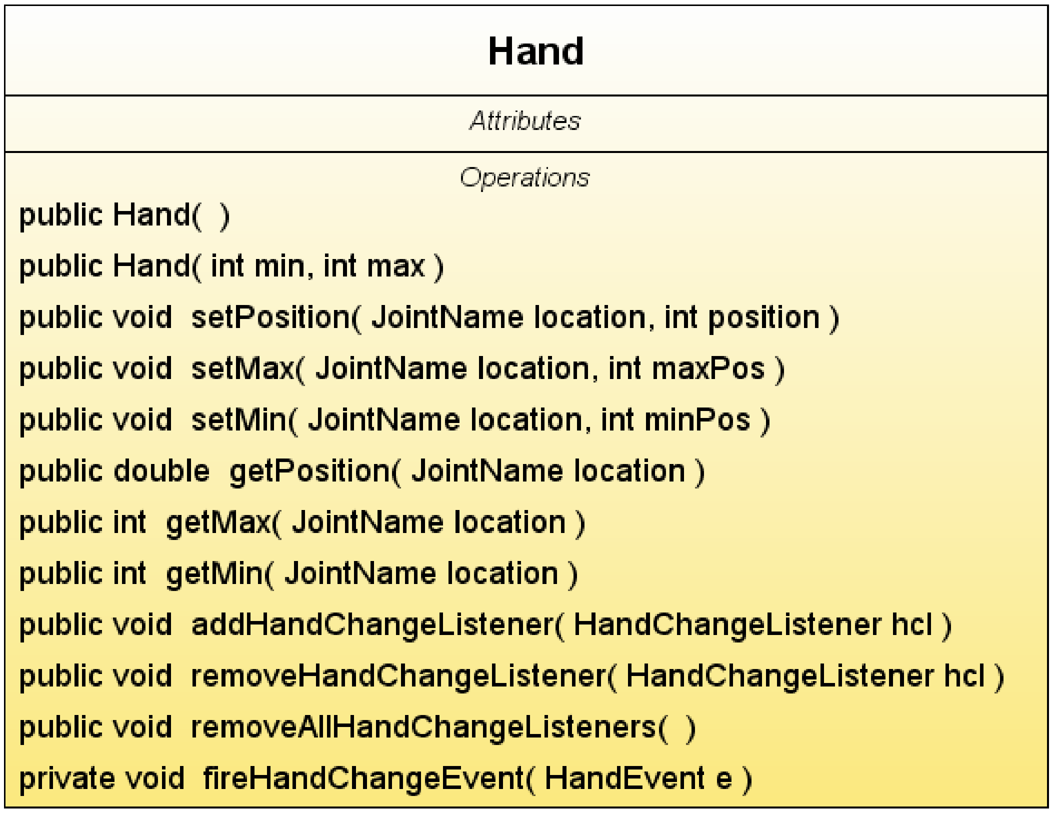
Sets the minimum range of motion value for this Joint. If the max position is the same as the min position, then the max position is incremented by 1 so as to prevent the joint from appearing fixed. This property has no use for this project.

getPosition

Returns the position of this Joint in relation to its range of motion. It determines the percentage of "bend" of the Joint and then reflects that percentage in terms of the range from 0 to 1.

Notice the difference between setting and getting the position of the Joint. To set the Joint, an integer must be passed in. To get the position, a double is returned.

4.4.3 Hand Class



```
Hand
Attributes
Operations
public Hand( )
public Hand( int min, int max )
public void  setPosition( JointName location, int position )
public void  setMax( JointName location, int maxPos )
public void  setMin( JointName location, int minPos )
public double  getPosition( JointName location )
public int  getMax( JointName location )
public int  getMin( JointName location )
public void  addHandChangeListener( HandChangeListener hcl )
public void  removeHandChangeListener( HandChangeListener hcl )
public void  removeAllHandChangeListeners( )
private void  fireHandChangeEvent( HandEvent e )
```

Figure 37: The Hand class

The Hand class, shown in Figure 37, contains a collection of `Joint`s which, when combined, form a representation of the human hand. It contains a key-value data set in which the key is an instance from the `JointName` and the value is an instance from `Joint`.

This class also utilizes a custom-built event propagation system which sends update notifications to all classes registered to it. In order for a class to receive events based on this class, it must implement the `HandChangeListener` interface, and then be added via the Hand.addHandChangeListener() method.

As a result of containing the `Joint` class, many of the methods from that class are overridden. To get a more detailed description of some of the methods, see `Joint`.

Some important methods to note in this class:

addHandChangeListener

This method adds the passed argument to the list of classes which are to receive notifications of raised events.

fireHandChangeEvent

This method is called anytime a change is recorded with the Hand. Essentially anytime a "set" method is invoked, it will call this method, which iterates through the list of listeners, passing it the information of the state change to each one.

4.4.4 JointName Class

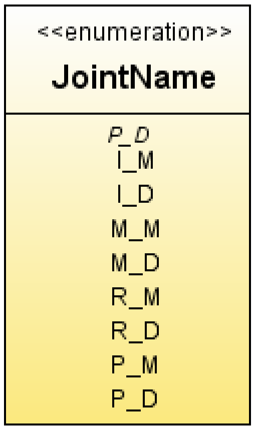


Figure 38: The JointName class

The JointName class, shown in Figure 38, serves as the structural base for the data in this project. It holds the order in which the data is interpreted from the serial port, as well as contains information to determine the pulse width modulation values that need to be sent to the servo that corresponds to the joint at the given position. If an additional entity is added to this enumeration, then it will be created in the Hand class, as well as in the HandMeter class, and will be accessed as any other joint would. This leads to extremely easy expandability, as all that needs to be done to ensure that another joint can be supported is to add it to this class. There are a few exceptions, such as classes that have literal values in them like the HandGestures class.

Some important methods to note in this class:

getServoPosition

This method takes a double as a parameter and returns an int. Based on which enum is invoking this method, a different value will be returned. Basically what this means is that custom information about each servo attached to the robotic hand is contained in here, and the integer that is returned from this method call is essentially the position that the servo needs to be at in order to represent the double value passed in to it. An example call to this method is shown below. This would return the servo position required to mimic the position of the middle-distal joint of the hand.

JointName.M_D.getServoPosition(hand.getPosition(JointName.M_D));

4.4.5 HandChangeListener Class

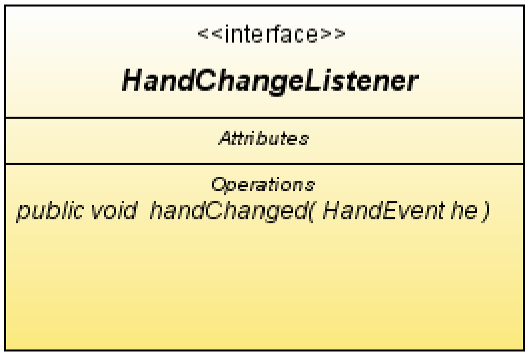


Figure 39: The HandChangeListener class

This is the interface all classes must implement if they wish to receive event notifications from the `Hand` class. All events are fired on the event thread, identically to `ActionListener`.

All known implementing classes:

- HandGraph
- HandMeter
- HandRecorder

4.4.6 HandEvent Class

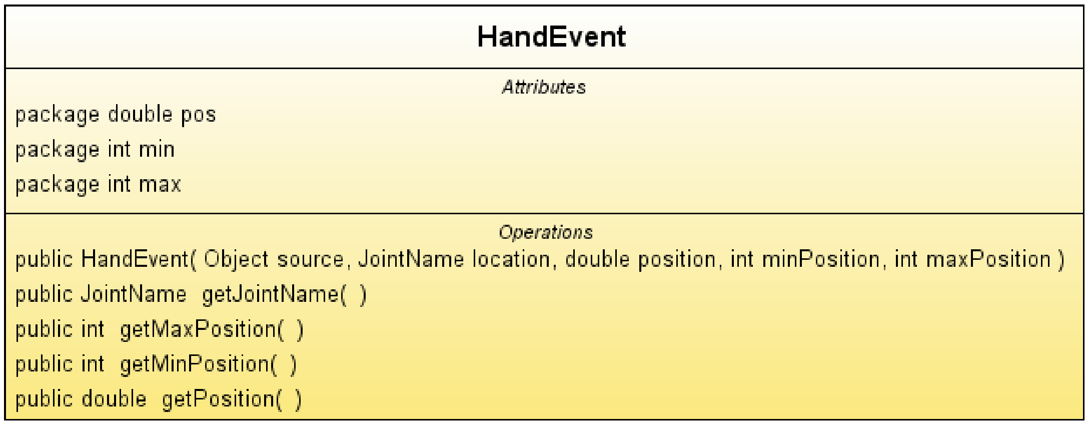
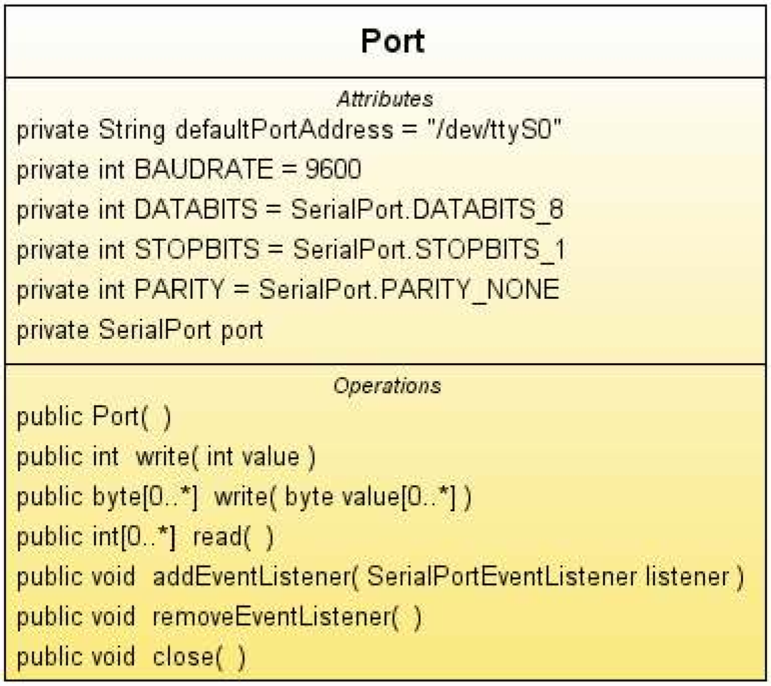

HandEvent
Attributes
package double pos
package int min
package int max
Operations
public HandEvent(Object source, JointName location, double position, int minPosition, int maxPosition)
public JointName getJointName( )
public int getMaxPosition( )
public int getMinPosition( )
public double getPosition( )

Figure 40: The HandEvent class

This is the class which holds information about the changes in state of the `Hand` class. Whenever an event is fired by the Hand class, an instance of this class is passed to each `HandChangeListener` which is registered to receive it.

Each instance of this class contains position, max, and min information about the affected Joint. With this implementation, it is impossible to discern if the change was because of a change in the position, or in the min or max, so all three must be accounted for each instance of this class.

4.4.7 Port Class



```
Port
Attributes
private String defaultPortAddress = "/dev/ttyS0"
private int BAUDRATE = 9600
private int DATABITS = SerialPort.DATABITS_8
private int STOPBITS = SerialPort.STOPBITS_1
private int PARITY = SerialPort.PARITY_NONE
private SerialPort port
Operations
public Port( )
public int  write( int value )
public byte[0..*]  write( byte value[0..*] )
public int[0..*]  read( )
public void  addEventListener( SerialPortEventListener listener )
public void  removeEventListener( )
public void  close( )
```

Figure 41: The Port class

This class handles opening and closing the port, as well as reading and sending data. Any program that wishes to use this program with event-driven capabilities must implement the SerialPortEventListener interface described in the javax.comm package.

This class was built as a wrapper class to the javax.comm.SerialPort class. It takes only the methods of interest, and also casts arguments from bytes to ints to make it easier to work with in Java. For the most part the methods in this class merely mimic their counterparts in the javax.comm package, except for making them appear as ints.

Some important methods to note in this class:

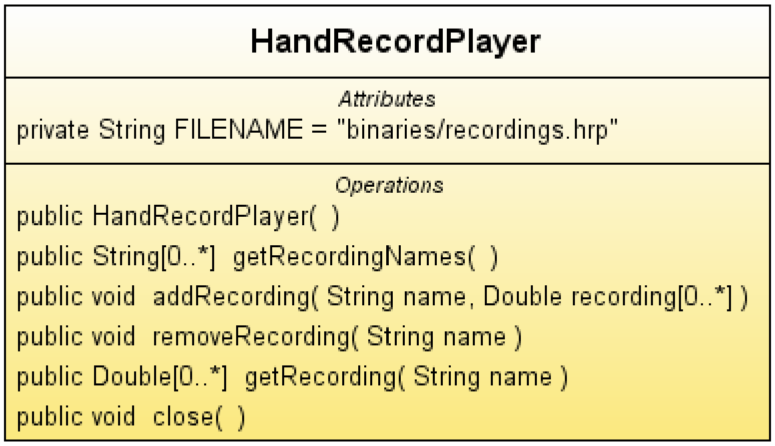
write

This method writes the given int value to the serial port. Since it is a wrapper for the write(byte) method, it is unclear as to exactly what happens when a value greater than 255 is passed. That has been a case that has not occurred in this project as of yet, so it was left untested. The original method allowed more than one value to be sent through the port at a time(in the form of an array of bytes), but since the buffer size of the development board is one byte, this method only allows a single byte to be sent per method call.

read

This method removes data from the input buffer of the serial port. It returns an array of integers because sometimes data is sent quickly enough for the buffer to contain more than one value. In our case, it is most common that only one integer will be in the array, but there are regular occurrences of more than one, but never more than four, bytes in the buffer.

4.4.8 HandRecordPlayer Class



```
HandRecordPlayer
Attributes
private String FILENAME = "binaries/recordings.hrp"
Operations
public HandRecordPlayer( )
public String[0..*]  getRecordingNames( )
public void  addRecording( String name, Double recording[0..*] )
public void  removeRecording( String name )
public Double[0..*]  getRecording( String name )
public void  close( )
```

Figure 42: The HandRecordPlayer class

This class contains a collection of recordings which can be identified with a title. It reads in a file from a default location and holds information in the form of a Hashtable with String as a key and ArrayList as the value.

A recording is formatted as an ArrayList<Double>. It works by storing the position values of each Joint in the Hand in the order in which they would be sent through the serial port to the servomechanisms. A more detailed account of how a recording is made is in the description of the HandRecorder class.

The methods in this class show a resemblance to the data field Hashtable. In essence, there is nothing more going on than what would be going on with a Hashtable, except for it loading from a file. This class was designed with the UI in mind, making it easier to load data without having to access information inside of UI code.

4.4.9 HandRecorder Class

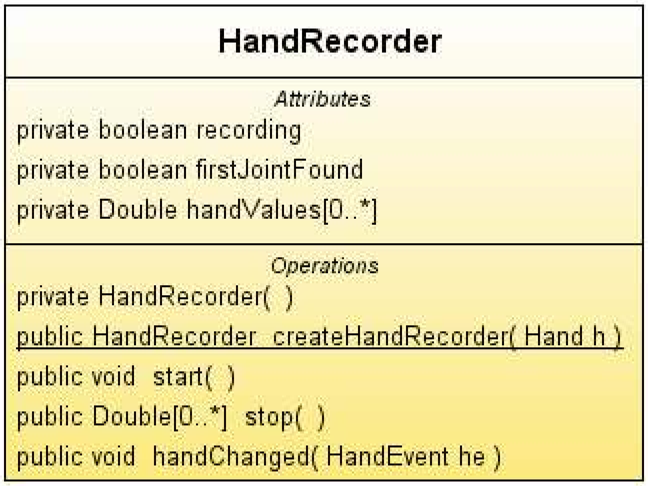

HandRecorder
Attributes
private boolean recording
private boolean firstJointFound
private Double handValues[0..*]
Operations
private HandRecorder( )
public HandRecorder createHandRecorder(Hand h)
public void start( )
public Double[0..*] stop( )
public void handChanged(HandEvent he)

Figure 43: The HandRecorder class

This is one of the classes which implement the HandChangeListener class for observing HandEvents. It receives notification of every change of state to the hand which is it attached to, and if it is recording, it appends the position value to the end of an ArrayList<Double>. This process is continued until the *stop* method is invoked, where it then returns the ArrayList. The result of this is a list of every position in the order which it was received from the time *start* was invoked. It is a little more complicated than this, but that is the general format of the class.

Some important methods to note in this class:

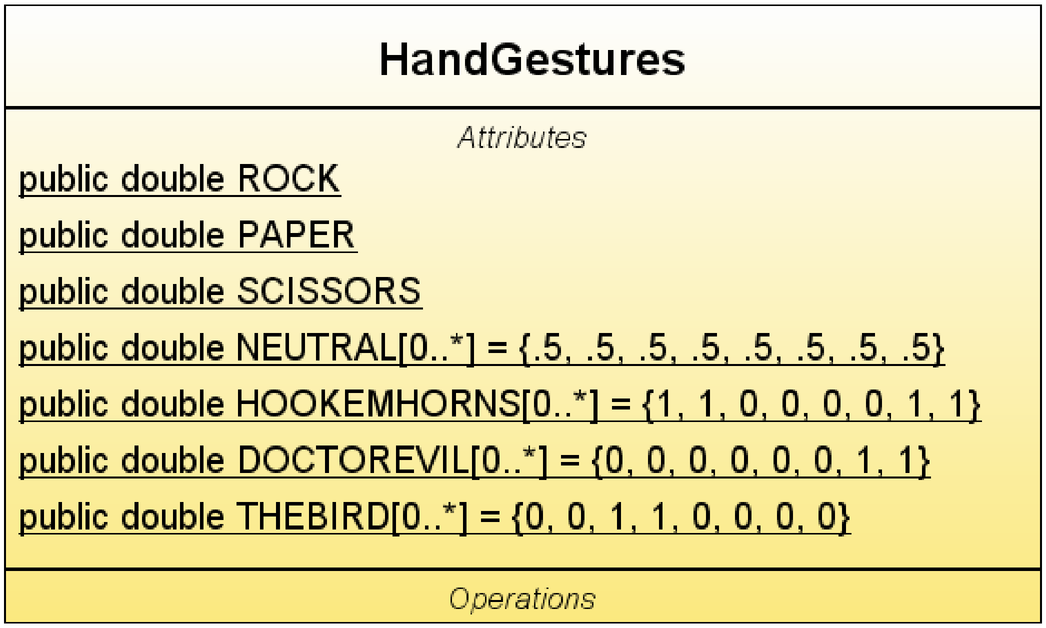
start

When start() is invoked, a flag is set notifying the listener that it needs to start recording the position values of the events it receives. It actually only starts doing so once it finds the first JointName (and thus the beginning of the serial port transmission cycle).

stop

When stop() is invoked, the flag is cleared and the list of position values accumulated up to this point is returned.

4.4.10 HandGestures Class



HandGestures

Attributes

public double ROCK
public double PAPER
public double SCISSORS
public double NEUTRAL[0..*] = {.5, .5, .5, .5, .5, .5, .5, .5}
public double HOOKEMHORNS[0..*] = {1, 1, 0, 0, 0, 0, 1, 1}
public double DOCTOREVIL[0..*] = {0, 0, 0, 0, 0, 0, 1, 1}
public double THEBIRD[0..*] = {0, 0, 1, 1, 0, 0, 0, 0}

Operations

Figure 44: The HandGestures class

This class only contains information for making gestures with the hand. These gestures are used through the Controller class' *makeGesture* method. Further information on how this works can be found in the Make Gesture Use Case.

4.4.11 Package graphicalComponents

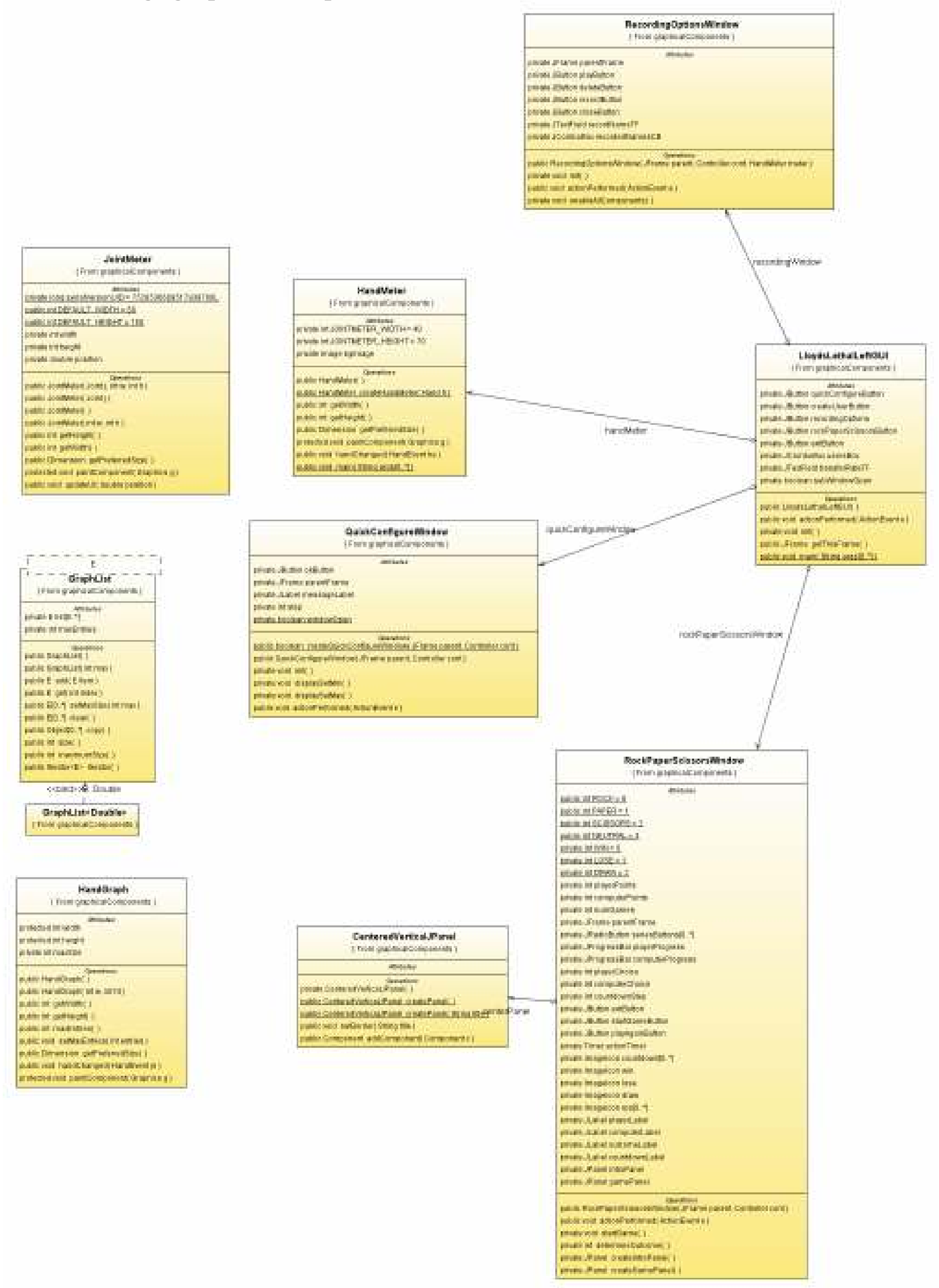
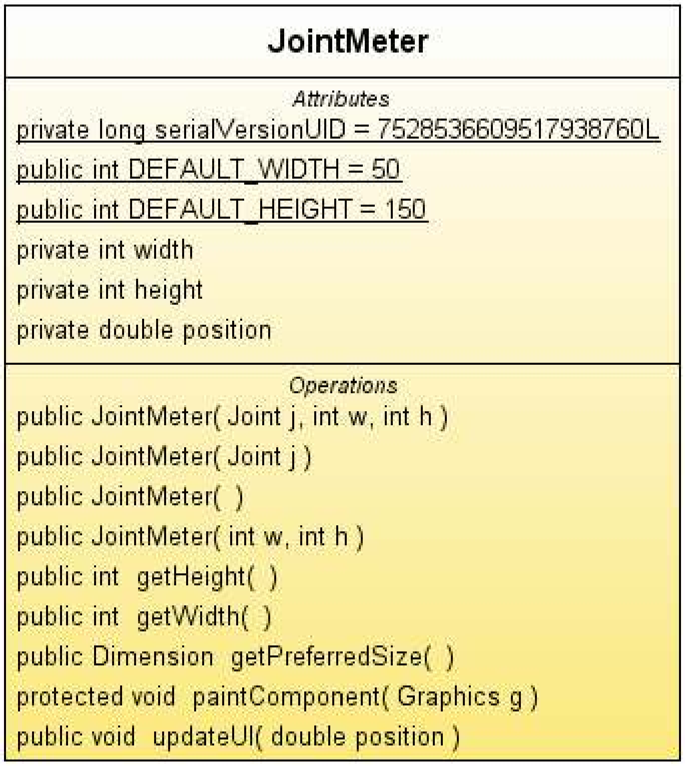


Figure 45: The graphicalComponents package

This package contains all classes that manage interfacing with the user and visually representing the hand. It parallels with the logicalComponents package in many ways when it comes to visually representing the hand, but differs greatly when it comes to interfacing with the user. The JointMeter and HandMeter classes parallel their logical counterparts, the Joint and Hand classes, and deal with displaying a visual rendition of the hand. The three main classes that deal with the GUI are the LloydsLethalLeftGUI, QuickConfigureWindow, RecordingOptionsWindow, and RockPaperScissorsWindow. These classes are the boundary classes that interact with the user and control the flow of the program. There are classes such as GraphList and HandGraph which are still part of the package, but did not get integrated into the final GUI design, so details about them are left out of this document. If you wish to obtain information about them, you can explore the source code to understand their meaning.

4.4.12 JointMeter class



JointMeter

Attributes

private long serialVersionUID = 7528536609517938760L
public int DEFAULT_WIDTH = 50
public int DEFAULT_HEIGHT = 150
private int width
private int height
private double position

Operations

public JointMeter(Joint j, int w, int h)
public JointMeter(Joint j)
public JointMeter( )
public JointMeter(int w, int h)
public int getHeight( )
public int getWidth( )
public Dimension getPreferredSize( )
protected void paintComponent(Graphics g)
public void updateUI(double position)

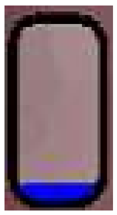


Figure 46: The JointMeter class

This class graphically represents a single Joint as a vertical bar. When the bar is completely filled, it is meant that the joint is completely flexed. When the bar is completely empty, then the joint is interpreted as completely straight. It was originally designed to rely on an ActionListener which kept track of a single `Joint`'s state, but then was modified to act as a part of the `HandMeter` class, and is manually updated as a result of events received from the ActionListener of the Hand class. A lot of the graphics have been disabled as a result of poor performance, but remain in comments in the paintComponent method.

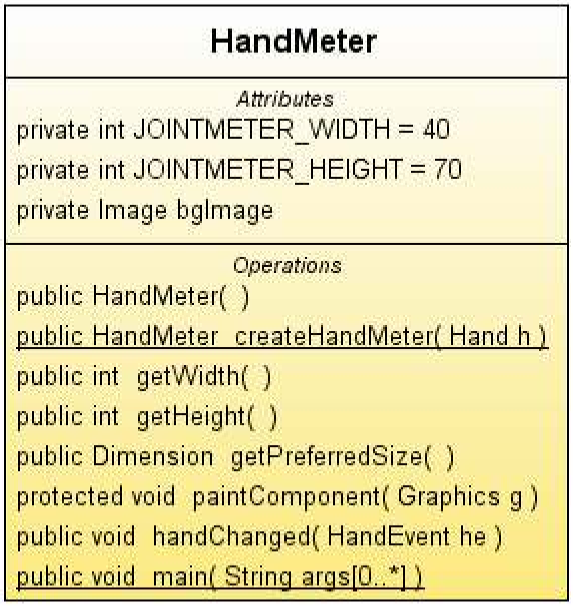
Some important methods to note in this class:

updateUI

This method updates the image of the JointMeter. It is passed a double value as an argument, and fills the bar up from the bottom depending on how close the value is to 1. It then calls the repaint() method, which re-renders the image to screen. Below is an example of how this method would monitor the position of a joint:

```
jointMeter.updateUI(joint.getPosition());
```

4.4.13 HandMeter class



```
HandMeter

Attributes
private int JOINTMETER_WIDTH = 40
private int JOINTMETER_HEIGHT = 70
private Image bgImage

Operations
public HandMeter( )
public HandMeter  createHandMeter( Hand h )
public int  getWidth( )
public int  getHeight( )
public Dimension  getPreferredSize( )
protected void  paintComponent( Graphics g )
public void  handChanged( HandEvent he )
public void  main( String args[0..*] )
```

Figure 47: The HandMeter class

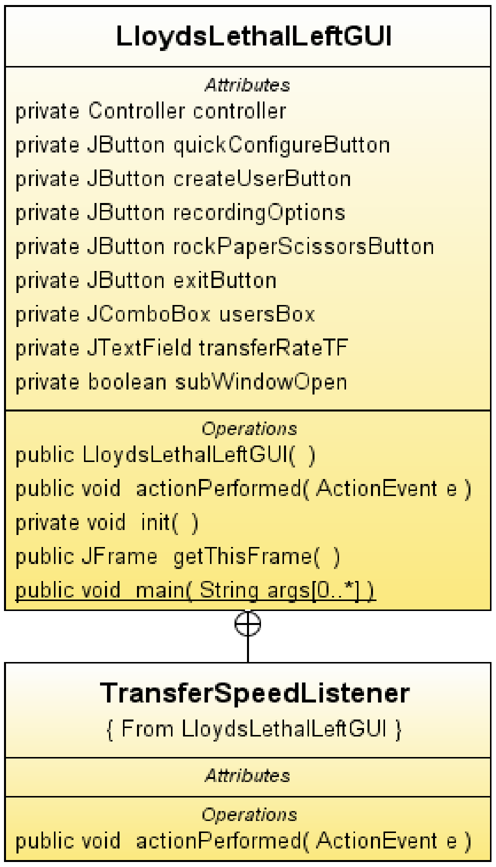
This class represents visually the hand. It mirrors the logicalComponents.Hand class. The underlying data structure which holds it together is a Hashtable<JointName, JointMeter>. Like the HandRecorder class mentioned prior, this class implements the HandChangeListener interface which allows it to be notified of HandEvents. How it handles events, however, is different.

When an event is received, the class updates only the JointMeter which is located at the index specified by the JointName passed in from the HandEvent class. Here is an example of how that works:

```
handChanged(HandEvent he)
{
	JointName affectedJoint = he.getJointName();
	handMeterHashtable.get(affectedJoint).updateUI(he.getPosition());
}
```

What this does is allows the graphical component to be up to date with the Hand class without extra code.

4.4.14 LloydsLethalLeftGUI class


LloydsLethalLeftGUI
Attributes
private Controller controller
private JButton quickConfigureButton
private JButton createUserButton
private JButton recordingOptions
private JButton rockPaperScissorsButton
private JButton exitButton
private JComboBox usersBox
private JTextField transferRateTF
private boolean subWindowOpen
Operations
public LloydsLethalLeftGUI( )
public void actionPerformed(ActionEvent e)
private void init( )
public JFrame getThisFrame( )
public void main(String args[0..*])
TransferSpeedListener
{ From LloydsLethalLeftGUI }
Attributes
Operations
public void actionPerformed(ActionEvent e)

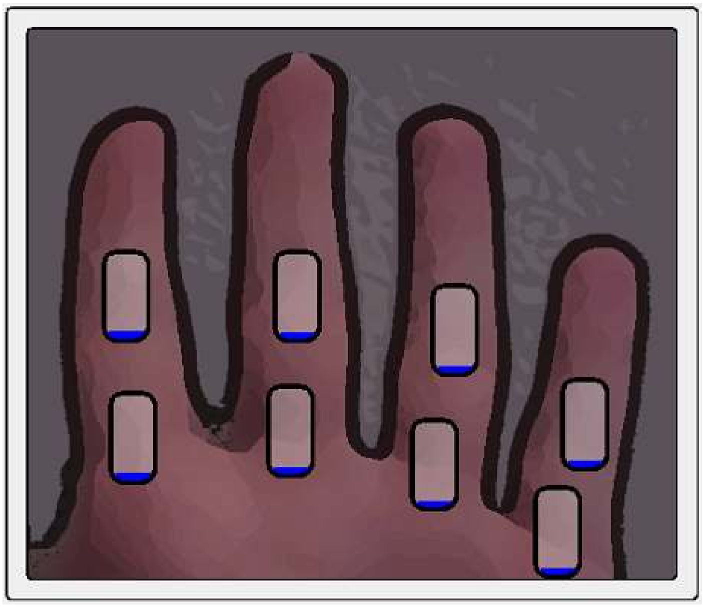
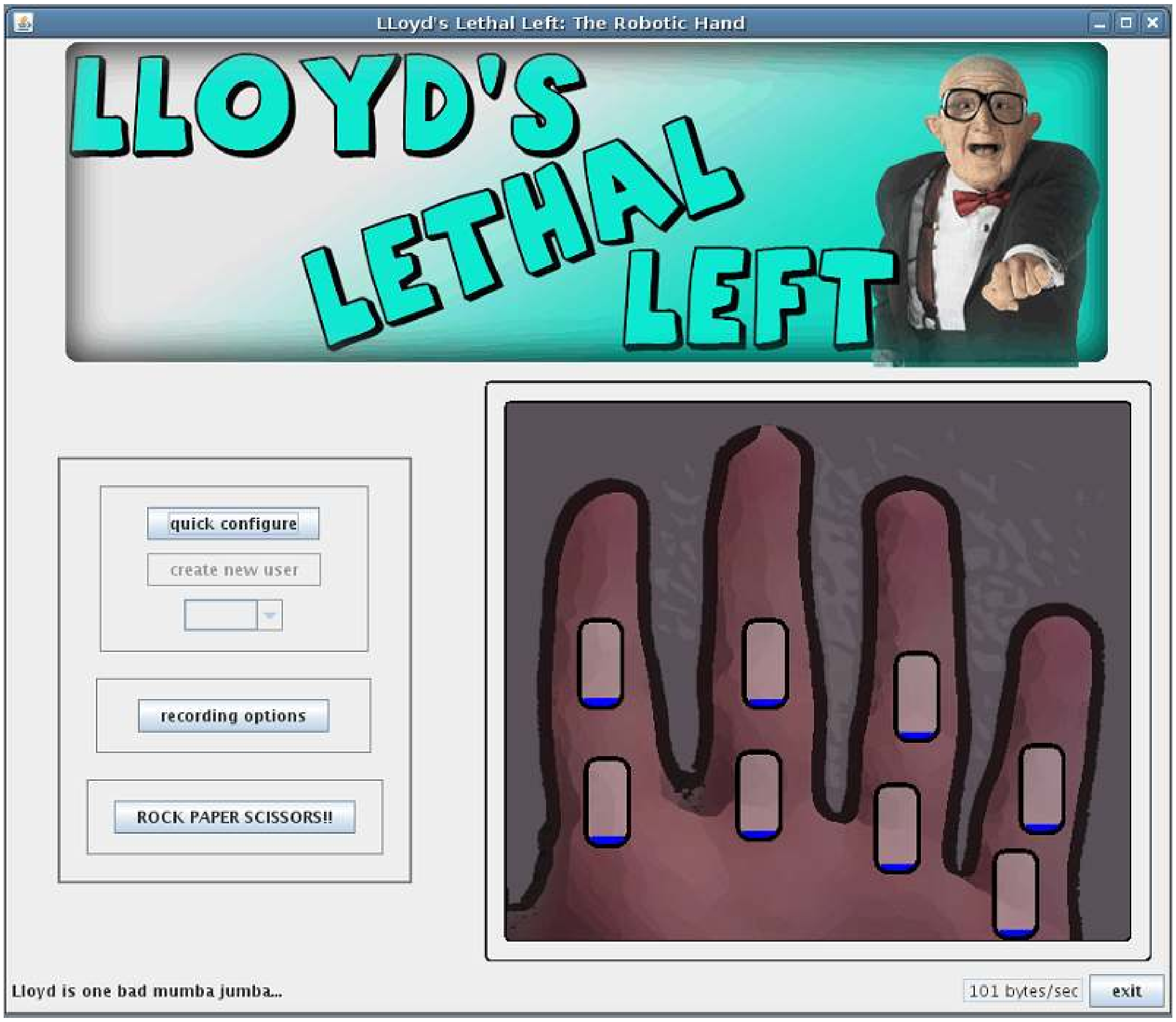


Figure 48. This is a screenshot of the HandMeter class. Each of the vertical bars is an instance of the JointMeter class superimposed on an image of a hand. When an event is received, a call to getJointName() will show which joint has been changed. Calling that JointMeter's updateUI() method will update that joint to resemble the new position of the joint.

This is the main GUI which interfaces with the user and connects with all of the logical components via the `Controller` class. This was created very rapidly and not intended to be as integral a part as the logical aspect. It presents a title bar on top, a menu on the left, a `HandMeter` on the right, and an exit button on the bottom, as well as a display to show the current transfer rate through the serial port.

The TransferSpeedListener is just a class that utilizes a javax.swing.Timer to update the transfer rate of the serial port every second.

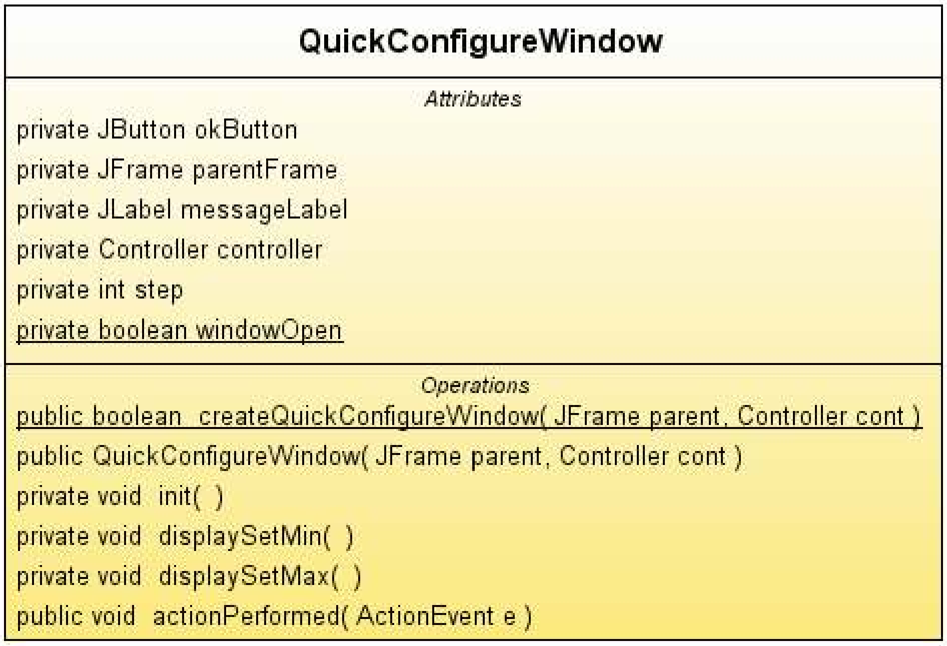


Lloyd is one bad mumba jumba...

101 bytes/sec

Figure 49. This is a screenshot of the main GUI window. You can see the title bar on top, the HandMeter on the right, the menu options on the left, and the transfer rate indicator on the bottom.

4.4.15 QuickConfigureWindow class



QuickConfigureWindow

Attributes

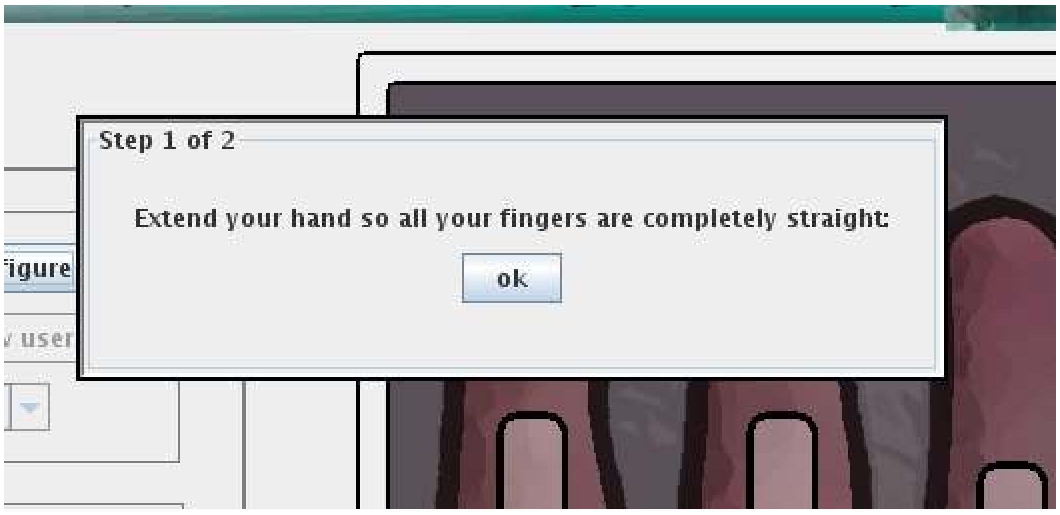
private JButton okButton
private JFrame parentFrame
private JLabel messageLabel
private Controller controller
private int step
private boolean windowOpen

Operations

public boolean createQuickConfigureWindow(JFrame parent, Controller cont)
public QuickConfigureWindow(JFrame parent, Controller cont)
private void init( )
private void displaySetMin( )
private void displaySetMax( )
public void actionPerformed(ActionEvent e)

Figure 50: The QuickConfigureWindow class

This class opens a new window which allows a user to configure the sensor glove to fit their hand. It walks the user through a two-step process that is outlined in greater detail in the sequence diagram for the Calibrate use case.

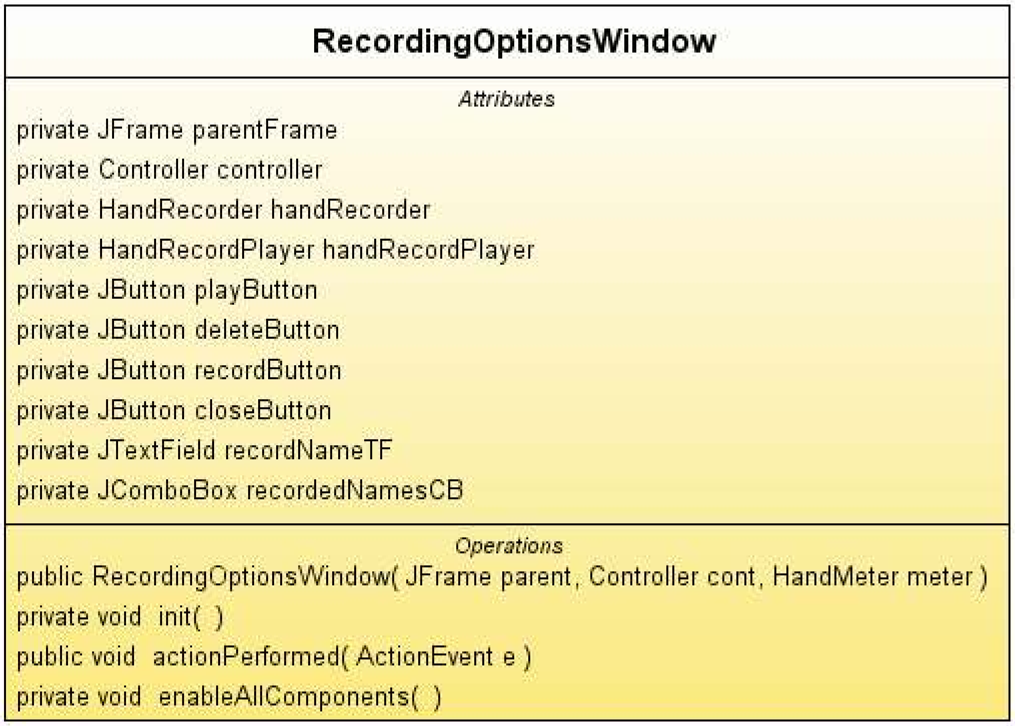


Step 1 of 2

Extend your hand so all your fingers are completely straight:

Figure 51. This is a screenshot of the window that pops up in the QuickConfigureWindow class. After clicking ok, another prompt appears, and then the class exits.

4.4.16 RecordingOptionsWindow class



RecordingOptionsWindow

Attributes

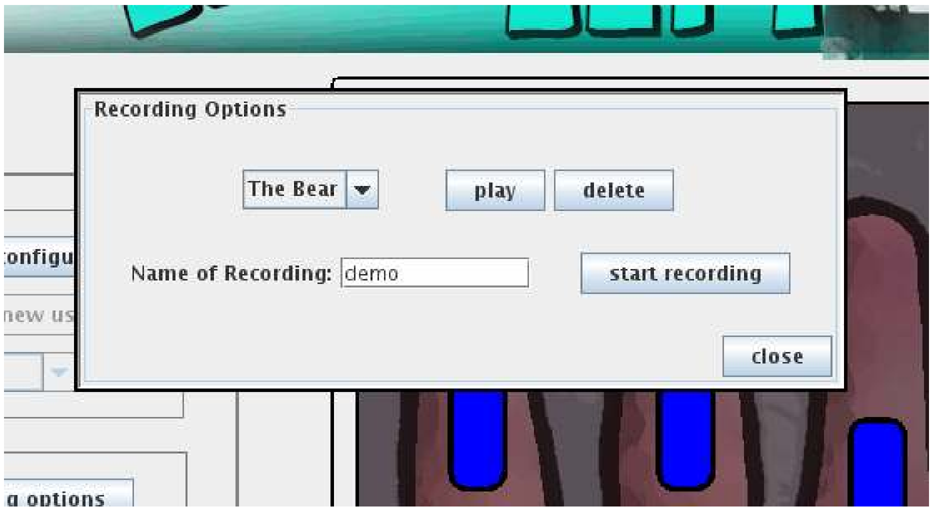
private JFrame parentFrame
private Controller controller
private HandRecorder handRecorder
private HandRecordPlayer handRecordPlayer
private JButton playButton
private JButton deleteButton
private JButton recordButton
private JButton closeButton
private JTextField recordNameTF
private JComboBox recordedNamesCB

Operations

public RecordingOptionsWindow(JFrame parent, Controller cont, HandMeter meter)
private void init( )
public void actionPerformed(ActionEvent e)
private void enableAllComponents( )

Figure 52: The RecordingOptionsWindow class

This class presents a menu for the user to perform two different use cases: "play a recording" and "make a recording". A detailed description of how the "make a recording" use case works is located in the sequence diagram located in the use cases section.



Name of Recording: demo

Figure 53. This is a screenshot showing the Recording Options window.

4.4.17 RockPaperScissorsWindow class

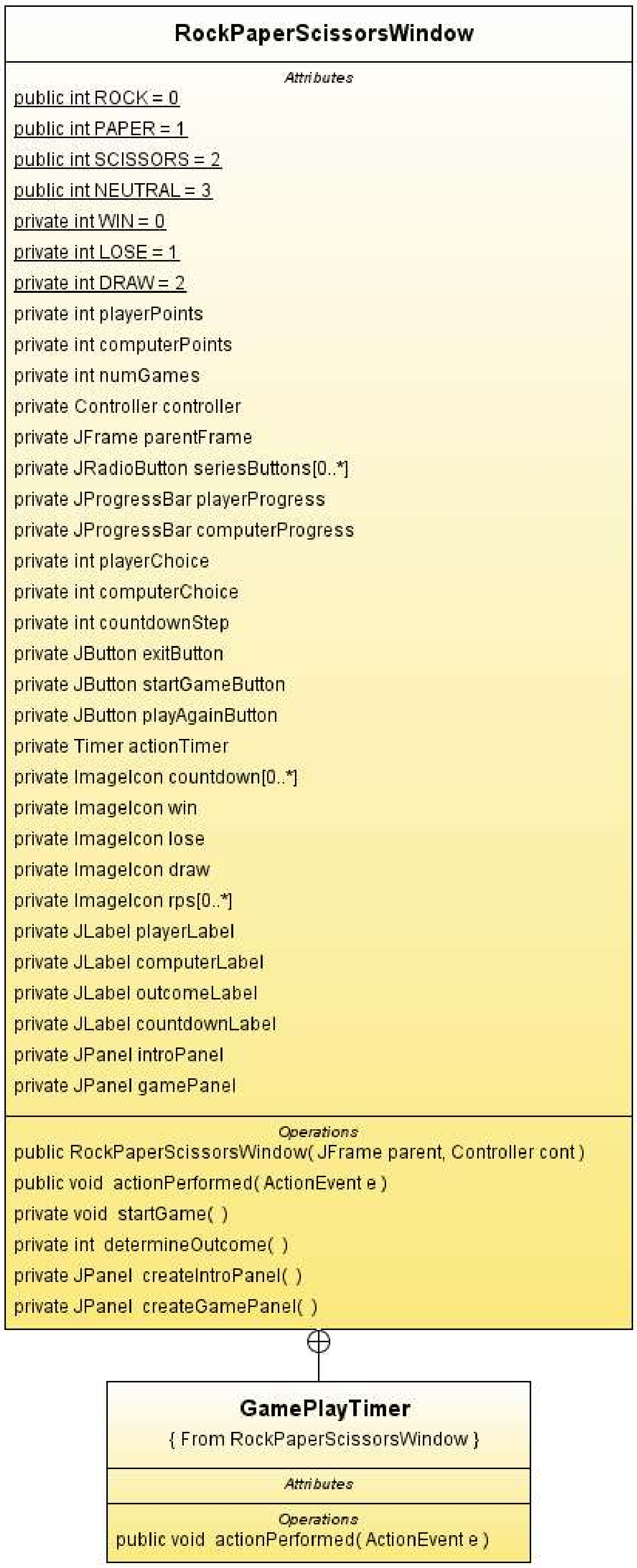

RockPaperScissorsWindow
Attributes
public int ROCK = 0
public int PAPER = 1
public int SCISSORS = 2
public int NEUTRAL = 3
private int WIN = 0
private int LOSE = 1
private int DRAW = 2
private int playerPoints
private int computerPoints
private int numGames
private Controller controller
private JFrame parentFrame
private JRadioButton seriesButtons[0..*]
private JProgressBar playerProgress
private JProgressBar computerProgress
private int playerChoice
private int computerChoice
private int countdownStep
private JButton exitButton
private JButton startGameButton
private JButton playAgainButton
private Timer actionTimer
private ImageIcon countdown[0..*]
private ImageIcon win
private ImageIcon lose
private ImageIcon draw
private ImageIcon rps[0..*]
private JLabel playerLabel
private JLabel computerLabel
private JLabel outcomeLabel
private JLabel countdownLabel
private JPanel introPanel
private JPanel gamePanel
Operations
public RockPaperScissorsWindow(JFrame parent, Controller cont)
public void actionPerformed(ActionEvent e)
private void startGame( )
private int determineOutcome( )
private JPanel createIntroPanel( )
private JPanel createGamePanel( )
GamePlayTimer
{ From RockPaperScissorsWindow }
Attributes
Operations
public void actionPerformed(ActionEvent e)

Figure 54: The RockPaperScissorsWindow class

This class creates a user interface for the game Rock Paper Scissors. It takes readings from the sensor glove as input for the human player, and pits it against a simple computer opponent. This class was hastily made, and, if it were more carefully done with more time allowed to work on development, it may have created more classes and thus a more robust framework. A detailed summary of the functionality of this class is located in the sequence diagram in the use cases section.

4.5 Use Case Design

There are four use cases: “Calibrate Hand”, “Play Recording”, “Make Recording”, and “Play Rock Paper Scissors”. The “Calibrate Hand” use case involves configuring the sensor glove to better fit a user’s hand. The “Play Recording” use case involves reading values from a file and sending those out to the robotic hand so that is imitates a motion that has been recorded from the user. The “Make Recording” use case involves creating the file that is read by the “Play Recording” use case. The “Play Rock Paper Scissors” use case allows the user to play a game of RPS against the computer.

Also included in this section is a state diagram describing the various states involved when data is received through the serial port. As it is the source of program flow, it has been added to this section.

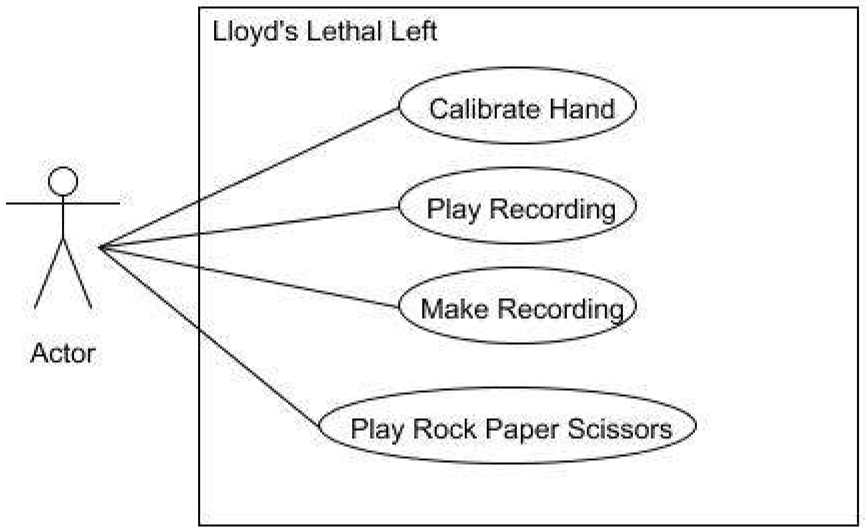
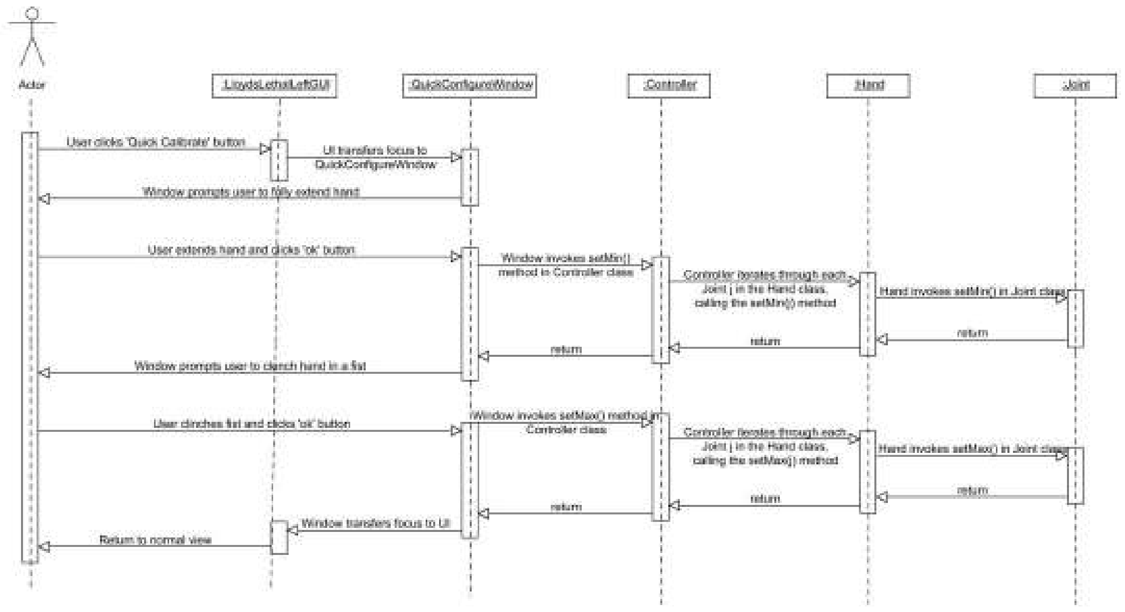


Figure 55: An overview of the four use cases

4.5.1 Calibrate Hand



Actor
:LloydsLethalLeftGUI
:QuickConfigureWindow
:Controller
:Hand
:Joint
User clicks 'Quick Calibrate' button
UI transfers focus to QuickConfigureWindow
Window prompts user to fully extend hand
User extends hand and clicks 'ok' button
Window invokes setMin() method in Controller class
Controller iterates through each Joint j in the Hand class, calling the setMin() method
Hand invokes setMin() in Joint class
return
return
return
Window prompts user to clench hand in a fist
User clinches fist and clicks 'ok' button
Window invokes setMax() method in Controller class
Controller iterates through each Joint j in the Hand class, calling the setMax() method
Hand invokes setMax() in Joint class
return
return
return
Window transfers focus to UI
Return to normal view

Figure 56: The Calibrate Hand use case

In this use case, the user activates a submenu which steps him through a two-step process of calibrating the hand.

4.5.2 Play Recording

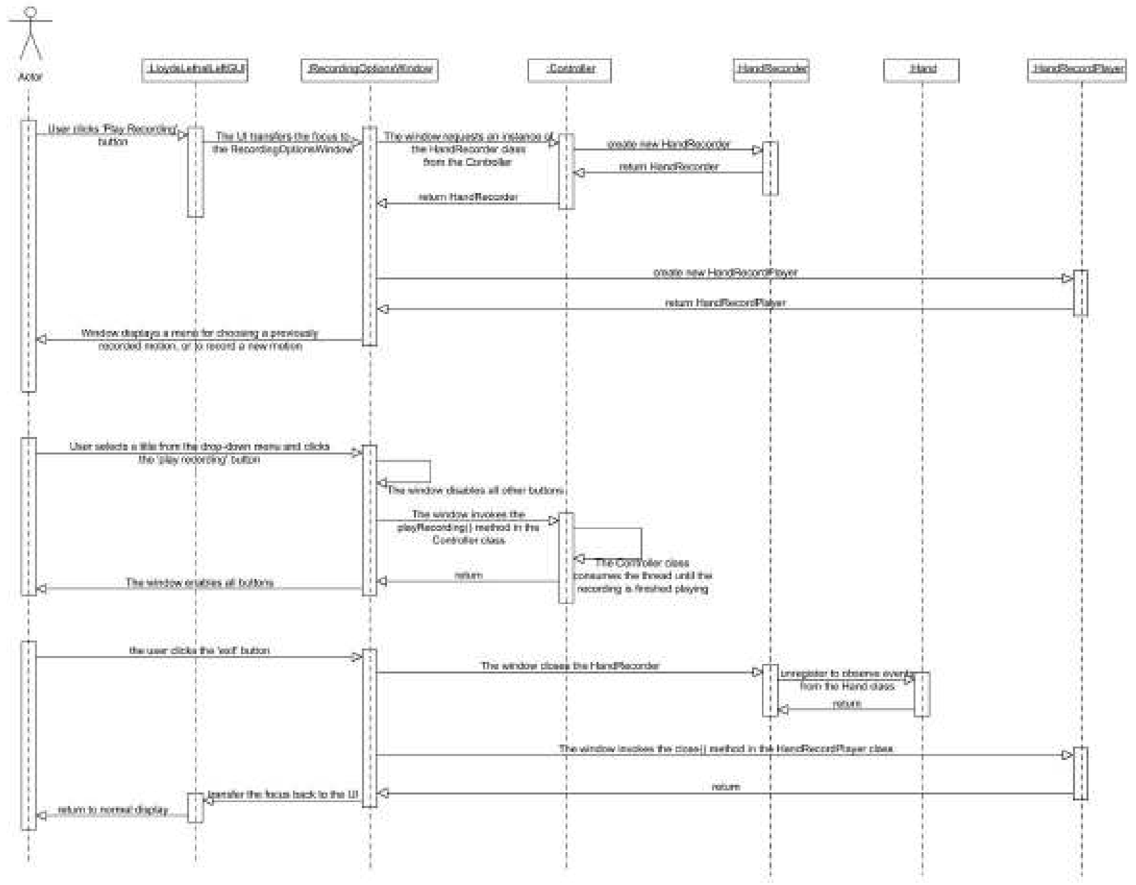

Actor
:LloydsLethalLeftGUI
:RecordingOptionsWindow
:Controller
:HandRecorder
:Hand
:HandRecordPlayer
User clicks 'Play Recording' button
The UI transfers the focus to the RecordingOptionsWindow
The window requests an instance of the HandRecorder class from the Controller
create new HandRecorder
return HandRecorder
return HandRecorder
create new HandRecordPlayer
return HandRecordPlayer
Window displays a menu for choosing a previously recorded motion, or to record a new motion
User selects a title from the drop-down menu and clicks the 'play recording' button
The window disables all other buttons
The window invokes the playRecording() method in the Controller class
The Controller class consumes the thread until the recording is finished playing
return
The window enables all buttons
the user clicks the 'exit' button
The window closes the HandRecorder
unregister to observe events from the Hand class
return
The window invokes the close() method in the HandRecordPlayer class
return
transfer the focus back to the UI
return to normal display

Figure 57: The Play Recording use case

In this use case, the user is brought to a submenu where he selects a recording to be played.

4.5.3 Make Recording

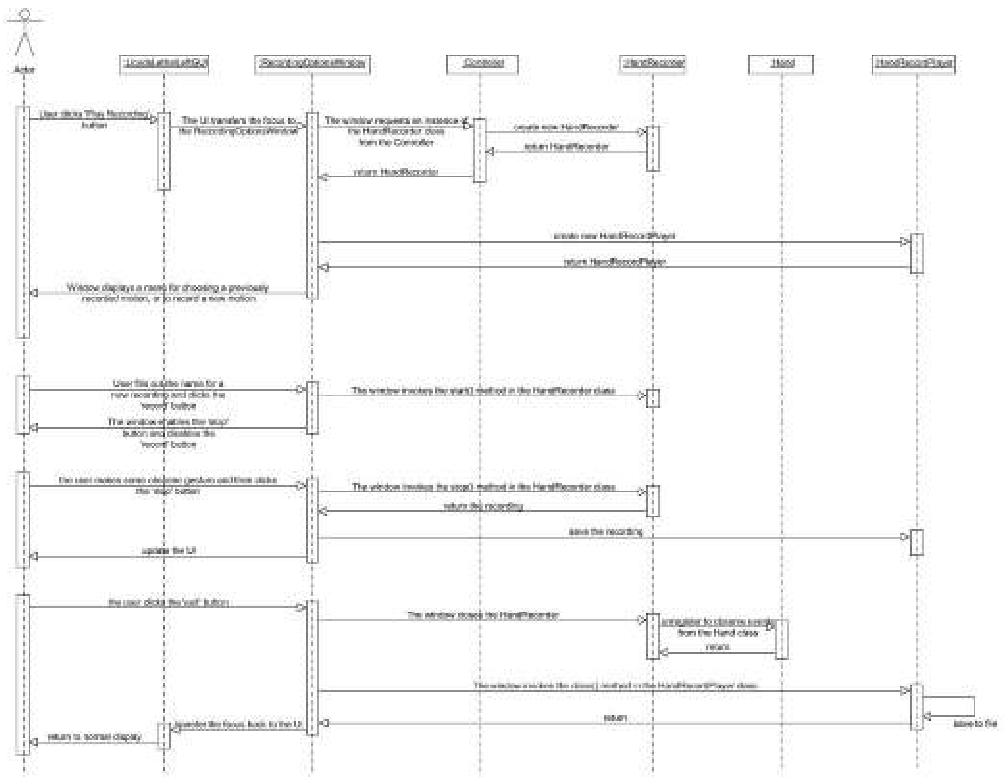

Actor
Controller
HandRecorder
Hand
HandRecordPlayer
The UI transfers the focus to the RecordingOptionsWindow
The window requests an instance of the HandRecorder class from the Controller
create new HandRecorder
return HandRecorder
return HandRecorder
create new HandRecordPlayer
return HandRecordPlayer
Window displays a menu for choosing a previously recorded motion, or to record a new motion
The window invokes the start() method in the HandRecorder class
The window enables the 'stop' button and disables the 'record' button
the 'stop' button
The window invokes the stop() method in the HandRecorder class
return the recording
save the recording
update the UI
The window closes the HandRecorder
from the Hand class
return
return
save to file
return to normal display

Figure 58: The Make Recording use case

In this use case, the user is brought to a submenu where he inputs a title and starts a new recording.

4.5.4 Play Rock Paper Scissors

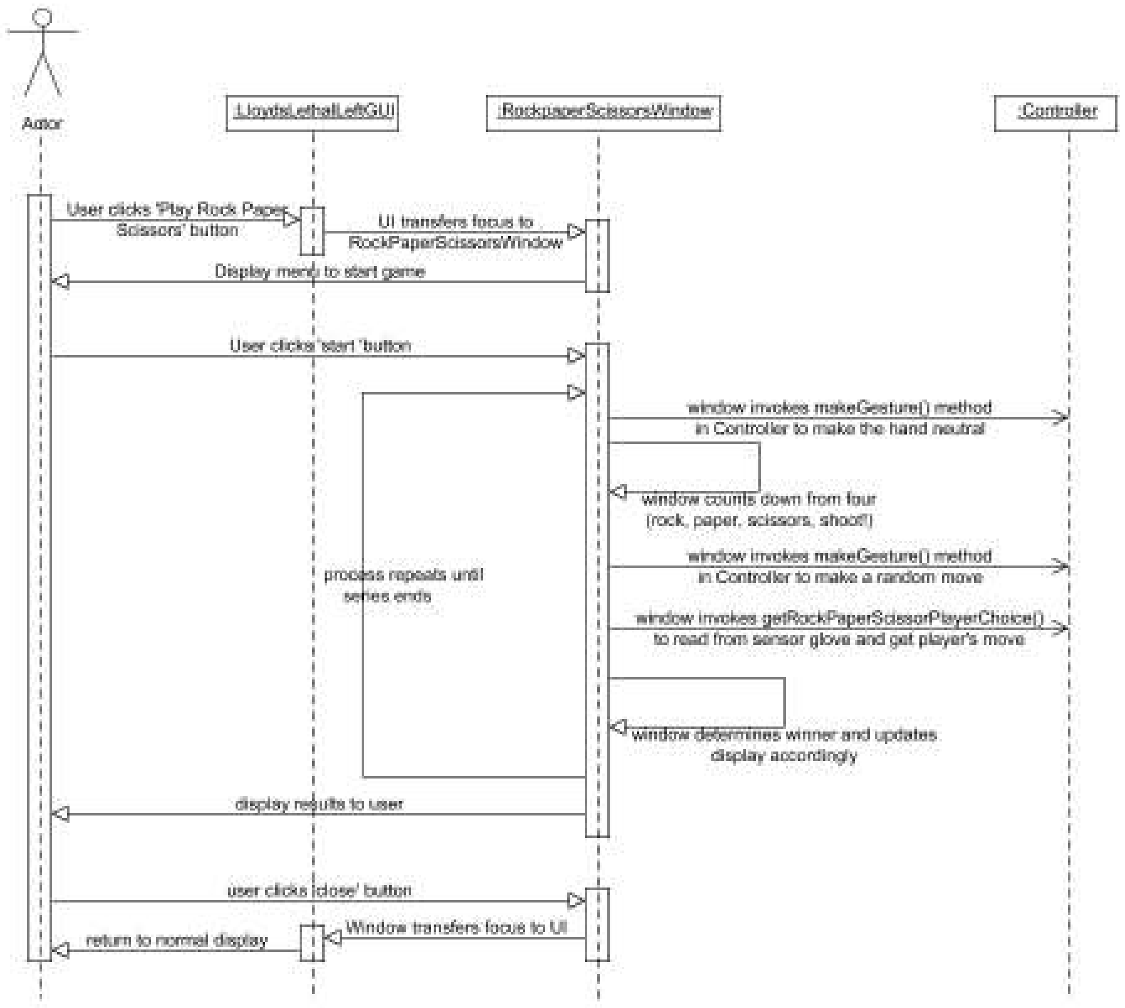

Actor
:LloydsLethalLeftGUI
:RockpaperScissorsWindow
:Controller
User clicks 'Play Rock Paper Scissors' button
UI transfers focus to RockPaperScissorsWindow
Display menu to start game
User clicks 'start 'button
window invokes makeGesture() method in Controller to make the hand neutral
window counts down from four (rock, paper, scissors, shoot!)
window invokes makeGesture() method in Controller to make a random move
window invokes getRockPaperScissorPlayerChoice() to read from sensor glove and get player's move
window determines winner and updates display accordingly
process repeats until series ends
display results to user
user clicks 'close' button
Window transfers focus to UI
return to normal display

Figure 59: The Play Rock Paper Scissors use case

In this use case, the user is directed to a new frame which plays Rock Paper Scissors. The GUI boundary class interacts with the Controller class to interpret the player's move, and also to make the robotic hand form a move.

4.6 Program Flow State Diagram

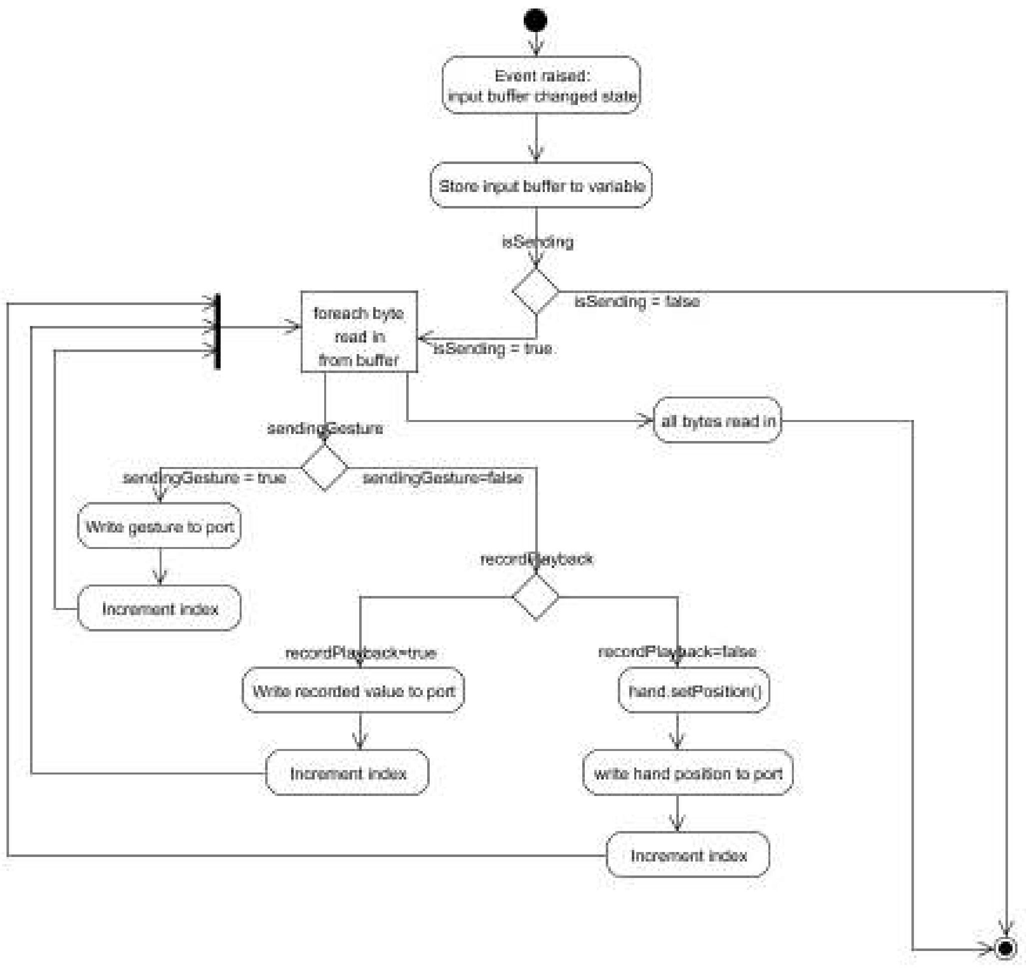


Figure 60: The Program Flow state diagram

This diagram describes the steps taken to manage data incoming from the serial port. The first thing needed to do is flush the buffer. Then a flag is evaluated to determine if anything needs to be sent out to the serial port. If so, then each byte is processed sequentially and the appropriate data is sent out through the serial port accordingly. The final step for each branch before continuing through the loop is to update the respective index for that procedure.

5 Results

5.1 Significant Changes

A small number of significant changes were made over the course of the project

- The initial plan was to write the GUI in C++. This was changed to Java to minimize the amount of development time spent on the GUI.

- We moved to a Linux box for serial communication, because it is difficult to access PC hardware in Windows XP.

- Prices for VR gloves were prohibitive. This led to the decision to construct our own sensor glove consisting of an array of linear potentiometers.

- Permissions for Shemp became problematic enough for the acquisition of an account with administrative privileges.

- The hand and control software went through numerous revisions to reach their final stage. Much of the design and our objectives changed as the project progressed.

5.2 Project Deliverables

The first deliverable was a rapid prototype of a single finger sensor interfaced with a single servo-driven robotic finger.

The second deliverable was a second prototype consisting of two robotic fingers responding to the movements of two independent stimuli.

The final deliverable was a complete robotic hand logically containing eight hinged joints as well as a sensor glove that monitored the movements of a user's hand.

The software deliverables included a user-friendly program with an incorporated GUI that allowed the user to interface the glove and hand and calibrate the robotic hand. There was also an option to record and playback hand motions, as well as a "Rock, Paper, Scissors" game.

5.3 Review of the Requirements Document

5.3.1 Configuration Management

We were initially storing files on the project account provided for us. However, due to difficulties with permissions, we decided to store future files on the local drive of our project computer. All hardware was stored in the project room in MCLT 212B.

5.3.2 Testing

Because we used the rapid prototype model for the creation of our hardware and software, testing was inherent throughout the life cycle of the entire project. In terms of mechanics, each prototype of the hand was tested with the hardware that we had available. The single finger prototype, as well as all subsequent models satisfied our requirements for power consumption and available torque. The software that we produced was a testing tool for whatever particular hardware prototype we were working on at the time.

5.3.3 Documentation

Our project was documented primarily through the required documents assigned through the course. Weekly reports were the main constituent of our documented progress. We also created some video clips for the code reviews.

5.4 Analysis of the Project

5.4.1 The Sensor Glove

The finished glove tracks the movement of fingers through the use of potentiometers. It outputs a voltage that does a reasonable job of expressing the position of a joint.

5.4.2 The Robotic Hand

The hands ability to mimic movement of a gloved hand depends greatly on the accuracy of the data coming from the glove. Because of this, the hand does not mimic the gloved hand perfectly but does a reasonable job. It mimics motion of each finger very well at the extremes. If a finger is fully extended or completely clenched, it mimics almost flawlessly. Between these two extremes is where it has trouble. Because the distal and middle joints are both controlled by the same servo, they move together even if the controlling hand as just moved the middle joint.

5.4.3 The Embedded Program

The embedded program performs what is required for the project to work. The main program loop can be restarted at any time by transmitting special control bytes from the Linux PC to the dev board; this allowed for us to modify software on the Linux PC without having to reload and restart the embedded system, which greatly facilitated development.

5.4.4 The Control Program

Given the limitations noted in Section 3.2, the control program successfully manages the data to interpret incoming sensory data and also controls the movement of the robotic hand to mimic the interpreted inputs. The flaws noted in Section 3.2 are barely visible to the user, and do not severely disrupt the flow of the program. Also, the ability to use the existing framework to extend the program to things such as record playback and a simple game demonstrate the effectiveness of the control program.

6 Conclusions

With regard to our functional objectives in section 2.1, this project successfully records the movements of four fingers, and recreates that movement through a robotic hand. Thus, the functional objectives are met.

With regard to our learning objectives in section 2.1, working with the dragon12 development board concludes that the objective to work with embedded systems is met. Our knowledge of serial communication has been greatly extended, so the objective to enhance our knowledge of serial communications has been met. Research in prosthetics and the experience gained from mimicking the hand concludes that the objective to learn more about prosthetics has been met. The objective to improve our ability to integrate hardware and software components has been met, as this project seamlessly integrates a java program with hardware components to form a single product. Finally, this project gives an example of the comparison of different levels of programming languages, and thus different layers of abstraction as it utilizes both higher-level programming (Java) and low-level programming (assembly), so the objective to work with different levels of abstraction is met.

Overall, this project met all objectives to be considered a success. Limitations are discussed in Section 3.2.

7 Recommendations

7.1 Robotic Hand

The robotic hand has a few problems that could be fixed with some work. Because we used JB Weld to connect the finger pads to the bones, it is not very solid. If it were possible to bolt these to the bones instead, it would increase the strength and durability of the hand. This would enable it to use the full torque of the servos. The thumb could also be made operational with some good mechanical engineering design.

7.2 Sensor Glove

The glove gets reasonable output, but does not get perfect data. A completely rebuilt glove, built from a new, stiffer material would probably lead to better data. Another option would be to use wire potentiometers instead of slide potentiometers. Wire potentiometers are just a thin wire that has leads attached to each side. As the wire bends, the resistance of the wire changes, resulting in a different output voltage.

7.3 Embedded Program

The embedded program needs to be rewritten in C to minimize hardcoding and allow for added functionality such as replacing single byte communication with packets.

7.4 Control Program

If this project were to be reproduced again in Java, the javax.comm package is an excellent tool for serial port communication, but is limited to the Linux operating system. There is, however, a package made by an independent source (i.e. not Sun) that works with Windows XP, but I cannot vouch for its performance.

Something that would boost performance greatly is using the native graphics hardware acceleration resources. There is a Java version of this in the JOGL package, or its equivalent in other languages.

The event-handling code could be optimized. As of now, when a HandEvent is passed, it is accurate as to which joint has changed, but is ambiguous as to what aspect of that joint has changed. Something as simple as adding another method in the HandChangeListener interface could help resolve this problem.

With a more general regard to event-handling, a recommendation would be made to have all components listening to changes to the hand class be routed instead to a separate event-propagation method which is centered around a clock instead of serial port events, so as to more completely standardize the execution of those components.

Perhaps the best recommendation would be to have the serial communication protocol be asynchronous so as to allow greater control of the software program. This would require changes in the hardware as well, and not just a software fix.

7.5 The Future

This project has probably reached its final point in development. If other groups wanted to expand on the project, they could work on any of the items listed above. However, the project was privately funded, so the hand and glove will remain in the possession of the group and will not be available to future students, which makes this section somewhat irrelevant.

8 Tools Used

Tools Used	***Source***
Wood working tools	Toledo High School wood shop
Machining tools	Toledo machine shop
Electrical engineering equipment	PLU electronics lab
ASMIDE	Internet
Eclipse	Internet
Linux Computer	D Wolff hooked it up
Dragon12 Board	CSCE 480 class
Caffeine	$2.00 at the pop machine

9 Cost Analysis

Item Name	***Cost***
Misc Construction material (BJ Weld, WD40, etc)	$50
Linear Potentiometers x11	$30
Servo motors x8	$165
Robotic hand Construction	$588
Babinga Wood for Base	$50
Dragon12 Board	$150
Mats. for glove	$50
Total	**$1108**

10 References

[i] "First bionic woman can feel it when people shake her prosthetic hand", Associated Free Press. September 18, 2006. http://www.physorg.com/news77786279.html
[ii] Jake Nelson's website: http://www.cs.plu.edu/~nelsonj/csce480
[iii] ATD_10B8C Block User Guide V02.12, p10. Courtesy of http://www.freescale.com
[iv] Ib. at 14
[v] Ib. at 16
[vi] Ib. at 17
[vii] Ib. at 18
[viii] Ib. at 19
[ix] Ib. at 21
[x] Ib. at 22 - 23
[xi] PWM_8B8C Block User Guide V01.17, p14. Courtesy of http://www.freescale.com
[xii] Ib. 20
[xiii] Ib. 22
[xiv] Ib. 23
[xv] Ib. 24
[xvi] Ib. 28
[xvii] Ib. 34
[xviii] Ib. 19
[xix] HCS12 Serial Communications Interface (SCI) Block Guide V02.05, p10. Courtesy of http://www.freeescale.com
[xx] Ib. at 14.
[xxi] Ib. at 17