The Dancer’s Friend

A Wearable Electronics Investigation

By: Paul Case

Senior Project

Presented to:

Computer Engineering Department

California Polytechnic State University San Luis Obispo

June 2012

In partial fulfillment of the requirements for the degree of Bachelors of Science

Advised by: Professor John Oliver

Abstract

This project is an initial investigation into the use of wearable electronics for wireless gesture detection, with the end goal of an application of augmenting dance performance through the use of gesture controlled sound. Though the system developed is not as yet of practical use to a dance performer, it lays the groundwork for future development and expansion, in both hardware and software design. In particular, it provides an in-depth overview of the lessons learned regarding the special hardware development considerations in wearable electronics to help ensure the success of future developments.

Table of Contents

Introduction and Background

The concept of wearable electronics is something society has been fascinated with for decades. From the vivid, glowing suits of TRON to the futuristic, augmented reality presented by the Google Glasses project, taking technology from in our pocket to on our pocket has long been seen as the next evolution in technological cool. Unfortunately for the world of the future, circuitry and comfort rarely mix. Fabric flows, stretches, and breathes, while circuit boards are hard and inflexible, and insulated wiring is thick and obtrusive.

Enter Dr. Leah Buechley, Assistant Professor of Media Arts and Sciences at Massachusetts Institute of Technology. Dr. Buechley, in cooperation with SparkFun Electronics, is actively involved in the development of a series of breakout boards specially designed for wearable electronics development. These circular 'LilyPad' boards have large conductive pads intended for use with conductive thread, eliminating the need for obtrusive copper wire, and are designed to be washable, allowing for permanent sewing of the electronics.

As the aforementioned TRON might imply, one of the most natural applications for wearable electronics, and the LilyPad system, is costume design. The simplest and most common of these is the integration of light, especially LEDs and EL wire, into costumes, but these only provide an added visual effect to a performance. A more revolutionary application would allow the wearer to remotely actuate lights and sounds, eliminating the need for a stage technician and allowing for improvisational effects.

This project focuses on the specific application of wirelessly controlling sound based on detected gestures, using sensors and other electronic devices sewn directly into the performer's outfit.

Project Definition and Goals

The long term vision for this project can be summarized as follows: Change a "Dance Concert" from a visual experience set to predefined music to one where music is augmented or created in synch with the motions of the performers on stage. To that end, these are the core requirements and design principles of the system:

- Precision - The performer must be confident their motions will reliably trigger the intended effects both in rehearsal and performance. They must also be confident that motions will not be falsely detected, so sounds are not raised haphazardly by the system.
- Responsiveness - The time between the triggering gesture and the system response should not be noticeable by the audience.
- Reusability - Investing in the hardware and software to augment only a single performance is a waste. The hardware must be capable of sensing a wide variety of actions, and the software must be easily configurable for different performances.
- Wearability. A costume that inhibits a dancer's ability to dance is of no practical use, regardless of how advanced the technology driving it is.

Due to time constraints and difficulties encountered during development, the project described here fails to meet the majority of these goals. It instead provides a solid foundation on which future work towards these goals can be constructed. The following work was done towards achieving these goals:

- Construction and analysis of a system measuring arm-based motion and transmitting the data wirelessly using the LilyPad system, with emphases on Reliability and Wearability.
- Design and analysis of an algorithm for converting measured data into meaningful gestures, with emphases on Precision and Responsiveness.

- Development and Testing of a simple system connecting these two elements together, with a discussion of how the Java MIDI library can be used to increase Reusability (at the expense of Responsiveness).

System Overview

The complete system developed in this project consists of four major physical components. The sensors, two ADXL335 analog output accelerometers, have their outputs converted to digital values by the LilyPad Arduino Main Board, based on the Atmel ATMega328P microprocessor. The data is transmitted over a serial communication to a computer using two XBee wireless radios, one connected to the Arduino via wire and the other to the computer via USB. The computer process the data using a Java program capable of reading serial input with the open source RxTx library, which calculates motions and produces audio as appropriate. A block diagram of the system is provided in Figure 1.

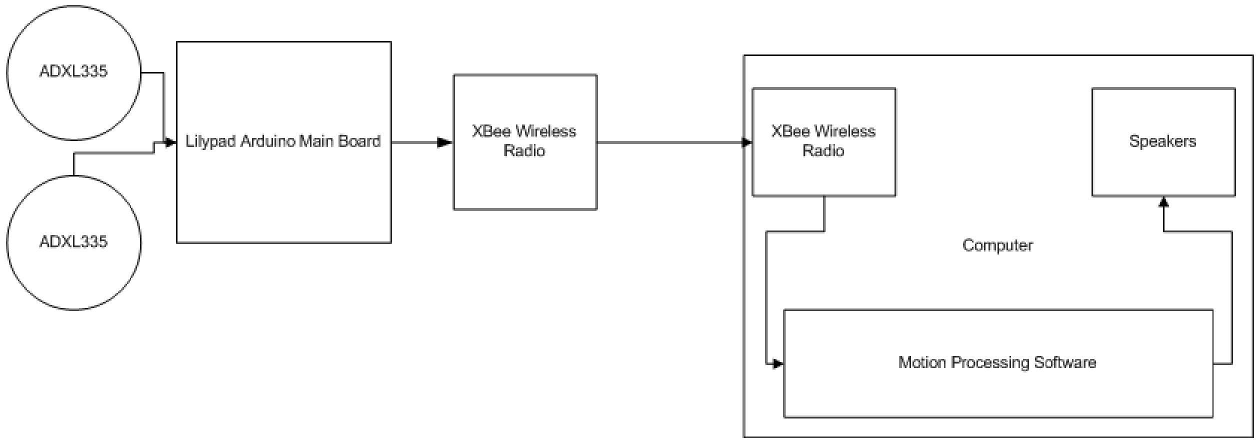


Figure 1 – System Block Diagram

Hardware Design

Component Selection

The selection of hardware components for the system is primarily driven by what is available in LilyPad form. The "heart" of the system – the LilyPad Arduino Main Board and LilyPad XBee – are obvious necessities, but the selection of sensors is limited on two accounts. First, the only motion sensor available in LilyPad form is the ADXL335 analog output MEMS accelerometer. Second, the ATMega328, and therefore the Arduino, has only 6 analog to digital conversion pins. Using two of these three-axis accelerometers, one on each forearm, takes up all 6 of the ADC pins, preventing the use of other analog measurement devices.

The power supply is a 400mAh, 3.7V, rechargeable Lithium Polymer Ion (LiPo) battery connected directly to the system, which lasts approximately six hours when the system is pulling its peak 65mA. This supplies both the 3.3V required by the XBee and 3V required by the Lilypad Arduino while maintaining an extremely small profile. Other considerations for power supply include a 5V LilyPad power supply driven by a AAA battery, and the 5V LilyPad LiPower step up. The former is not used due to the bulk of a AAA battery compared to the thin, flat, LiPo batteries. The latter is the ideal choice for the system , as a higher voltage ensures the system will last longer as the connections wear down, but was out of stock at the time of development.

The hardware schematic is provided in Figure 2.

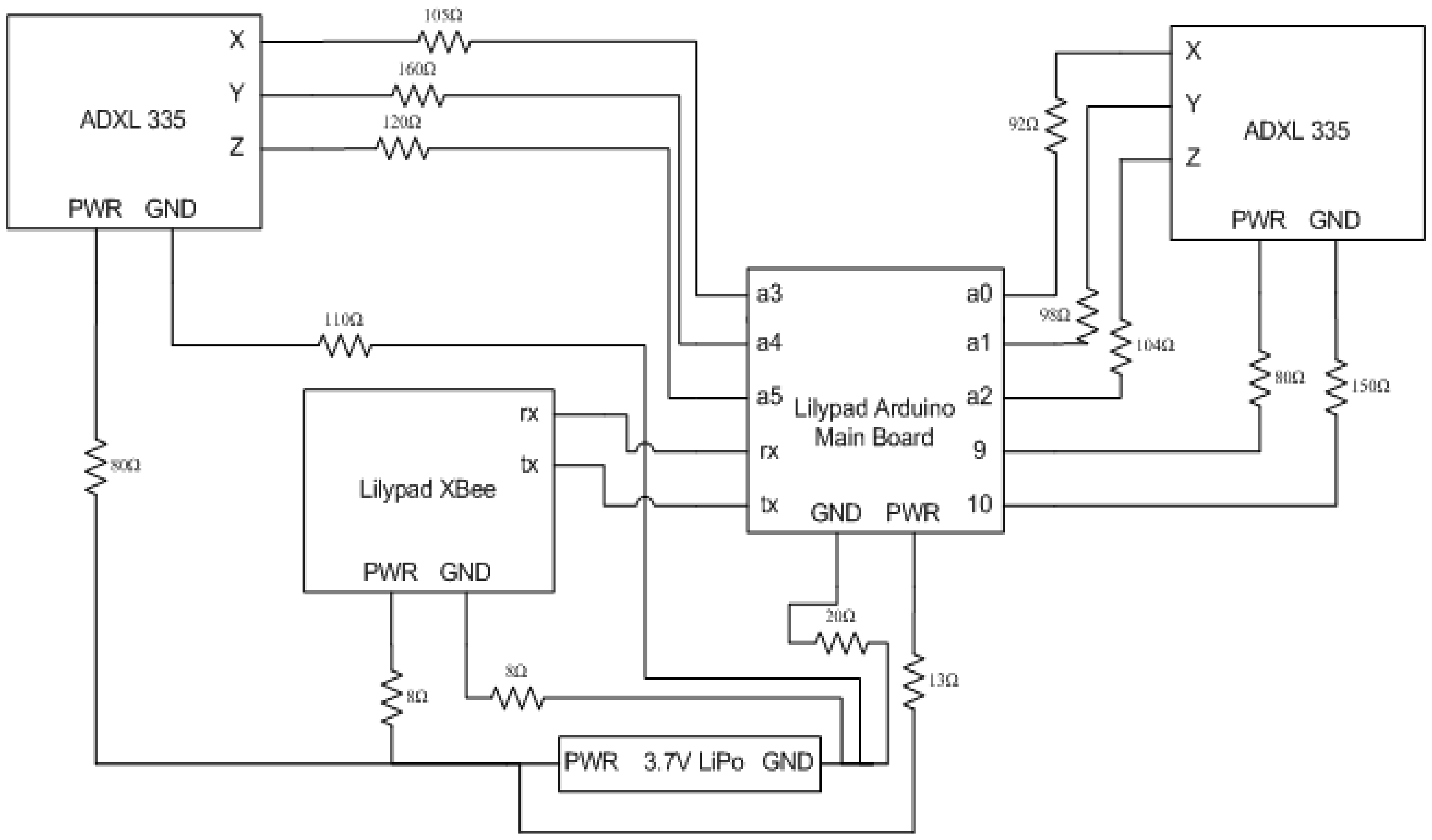


Figure 2 – Hardware Schematic. Resistances are due to conductive thread. See below.

Special Wearable Electronics Considerations

Prototyping and developing a wearable electronics system presents a number of unique challenges not normally present in embedded systems development. The first challenge is prototyping a circuit using the components. Due to the unique, circular shape of the LilyPad boards, they are incompatible with the breadboards traditionally used in prototyping. The most obvious solution is a large number of alligator clips, taping the electronics to the clothing in place, but this is far from a perfect solution. When being worn and tested, the alligator clips have a tendency to slide around the LilyPad, causing shorts or broken connections. In addition, the prototype will be missing a significant component of the final product – resistance.

Because of the relative inflexibility of copper wiring, the ideal wiring solution for wearable electronics is to sew the components together using special conductive thread. This allows the connections to ebb and flow with the movement of the fabric, and maintains a much lower profile than the thick insulation on most copper wire. It also comes with a non-trivial resistance – for the wire used in this project, it is approximately 100Ω/ft. This is the source of the resistances in Figure 2. This resistance is a trivial concern for the accelerometers in the system, as they only pull 350µA of current, but for the XBee wireless radio, which pulls 40-50mA, even 10Ω can result in a .5V drop in supply voltage. In fact, the power and ground connections on the XBee are sewn over four times in order to bring the resistances down to the values shown.

In addition to the non-trivial resistance, using conductive thread presents other problems in the area of Reliability. In order to maintain a small enough profile to be both sewn and worn, conductive thread is not insulated. A stray bit of frayed thread too close to another connection can cause a short, and so all lines should be kept at least an inch apart where possible. Where connections get close or fraying is likely, such as at knots and LilyPad connections, a small bit of fabric glue serves as an insulator that both coats the thread as well as binds to the underlying fabric. This can also be used to insulate any location where lines need to cross, though the components should be laid out to minimize line crossing as much as possible. This implementation, for example, only has a single crossed line (the ground of A1 over the Rx and Tx lines, as seen in Figure 2).

It initially seems prudent to coat all sewn connections in fabric glue to increase reliability, but fabric glue is difficult to work with and dries thick. This leads to minor discomfort where applied liberally, especially around joints, impeding wearability. As an alternative to insulation for avoiding shorts, the following two design considerations are useful. First, avoid placing more than one line on the inside of a human joint so when the joint flexes it the line will only cross itself. Second, ensure that no lines are exposed to the outside world when the costume is fully assembled so that shorts will not occur when the one part of the costume contacts another. The easiest way to achieve this second consideration, and the one used in this project, is to design the system to be worn underneath another costume. This way lines are all be covered in another layer of fabric, the electronics are not be visible to the audience, and the system is reusable with multiple different outer costumes. It is also recommended that another layer of fabric be sewn on the inside between the connections and the user, or that the user wear something underneath the electronics, as sweat built up over the course of a performance can lead to shorts between neighboring lines.

A final important consideration with wearable electronics is the fabric itself. Because this project is designed to be worn underneath a costume, and because dance costumes have a tendency to be tight to the body, the system is sewn into a highly elastic, skin-tight body suit. Unfortunately, conductive thread has no lateral give, in contrast to most sewing threads, and so the electronics must be sewn onto a mannequin to ensure the thread does not break when the system is put on. This also means that there is a significant quantity of extra thread which bunches and loops when the system is not being worn, increasing the likelihood of shorts, and frustration, during testing.

Software Design

The Software portion of the project consists of three major components: The sending of the data from Arduino to PC over Serial, the processing of data on the computer, and the production of audio output when a gesture is detected. The software block diagram is shown in Figure 3.

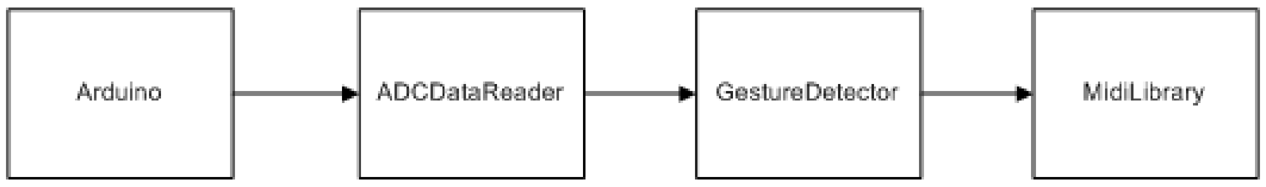


Figure 3 – Software Block Diagram

Data Communication

Data acquisition and transmission is handled by the Arduino. On startup, the Arduino waits to receive a ready signal from the computer. Once received, it begins transmitting single byte values read from each of the 6 ADC inputs, followed by a “synchronizing” 0. Even though the Arduino is capable of 10 bit ADC resolution, only the 8 most significant bits are transmitted in the interest of responsiveness, as it would take transmitting a second byte over serial to communicate the extra two bits. The synchronizing byte is used to ensure the computer knows which byte corresponds to which ADC input in case a byte is dropped. Any actual ADC readings of 0 are converted to 1’s to avoid conflict with the synchronizing byte.

The Arduino sketch code is provided in Appendix B.

Wireless serial communication between the Arduino and PC is handled by the two XBee wireless radios. For instructions on setting up and configuring the XBee radios, see appendix A.

The data is received on the computer for processing using the open-source RxTx library, found at http://rxtx.qbang.org. The RxTx library allows for the opening of a Serial port (COM ports on Windows) as standard Java Input and Output streams. Bytes are read in groups of seven, one for each of the six ADC inputs and one for the synchronizing byte. Before any data is passed on for processing, the synchronizing byte is checked for 0. If the check fails, the data is discarded and bytes are read one at a time until a 0 is found, re-synchronizing the data stream, and then another 7 bytes of good data are read.

The Java class ADCDataReader, is provided in Appendix C.1.

Gesture Detection

Gesture detection is performed using a method suggested by Goh Chun Fan et. al. of Nanyang Technological University in Singapore. It relies on two principles of human motion: All forceful motion has a strong starting force and a strong ending force, and motion of human joints is circular rather than linear. With this in mind, the presence and direction of a sharp, sweeping motion of the forearm can be detected using two factors: A pair of extrema of large magnitude and opposite direction on the axis of translational motion, and a single extremum of large magnitude on a second axis indicative of the centripetal acceleration. The two axes are used to determine the plane of motion, and the order of the extrema on the translational axis can be used to determine the direction.

The Java class GestureDetector, designed to detect gestures on a single, 3-axis accelerometer, is provided in Appendix C.2. The max and min threshold values passed to the constructor should be determined through manual testing or an automatic calibration step. Separate values are provided for each axis to handle varying resistances between the each accelerometer output and ADC input due to the conductive thread. The timeThreshold value determines the number of data points that should be seen between the two threshold values before a gesture is considered complete. A higher timeThreshold value will help prevent false positives and negatives due to noise, but will also increase the amount of data points required to detect a gesture.

Sound Output

In order to provide maximum sound configurability with minimum setup, this project uses the javax.midi package to produce various sound outputs. Using the MIDI package, a wide selection of instruments available to the MIDI device can be listed for user selection and played back with little effort. Due to time constraints, development of a GUI and custom sound configuration tool fell outside the scope of the project – a large miss on the reusability goal, but one that is easily remedied in future work.

Sample javax.midi code for listing instruments and performing audio playback are provided in Appendix C.3.

Software analysis - Responsiveness concerns

The implementation of the software provided here has two major concerns relating to responsiveness. The first is a delay of up to 100ms built into the default Java MIDI synthesizer. This can be resolved using an alternate, more responsive synthesizer or a non-MIDI method of producing sound output.

The second, more significant concern is that the time required to perform the six ADC conversions and transfer them to the computer is shorter than the time required to run the GestureDetector on the data points. This results a data backlog in the XBee’s data output buffer, preventing the motions from being processed in real time. For example, if it takes 5ms to send one wave of ADC data but 15ms to process it, the computer and Arduino will become 10ms more out of sync with each wave. With the XBee’s data output buffer being 202 bytes in size, up to 28 waves of data can be backed up, resulting in an maximum processing delay of 280ms, at which point data is dropped.

There are a few ways to remedy this problem. The first is to delay the sending of data on the Arduino side so that data will only be processed and sent as frequently as the code running on the computer can handle it. This approach allows the bulk of the calculations to be performed on the computer, but the delay needs to be recalibrated for different computers of different speeds. A second approach is to have the computer signal the Arduino when it is ready for more data. This adds a slight delay in sending and receiving the signal, but it is negligible.

For optimal performance, the GestureDetector code should be converted to C++ and implemented Arduino side. This adds significant development overhead, as compilation, upload, and testing takes much longer when working with embedded systems than it does when working in Java, but provides significant performance benefits. First, the Arduino will perform ADC conversions the moment it is ready to perform them, eliminating any data backlog or signaling delay problems. Second, much less data is transferred over the wireless link, reducing the overall power consumption of the system. Some extra configuration is required on XBees, however, to ensure data is not lost when a detected motion is sent from the Arduino to the PC for handling – dropping a motion is a much more serious concern than dropping a data point.

The implementation presented in the Appendix uses the first method of resolving these concerns as much time was spent gathering sample data to test the GestureDetector offline, in which case data backup was acceptable. Future extensions of the project should consider revising the implementation presented here to use the second or third options, where the second is the easier to implement and the third is the most optimal.

Overall System Analysis

Overall, the final developed product does not perform to expectations. Significant development time was lost to the issues discussed in *Special Wearable Electronics Considerations* above. Broken connections, shorts, and power failures are common during use, and the system needs to be reset every few minutes as a result.

The system is able to reliably detect motion only in the plane of motion parallel to the user's wrist, as well as the direction of motion, resulting in four detectable gestures – two directions for each hand. This number is small compared to the wide array of gestures originally envisioned, but was limited by both physical construction failings as well as only being able to use two accelerometers.

At their core, the critical failings of the system stem from the developer's lack of talent as a tailor and failure to enlist help from an available professional from the Theatre and Dance Department, resulting in a poor and unreliable sewing job.

Overall, while the system is capable of providing bursts of short, entertaining demonstration, it requires much further development before it would be of any practical use in a dance performance scenario.

Future Development and Extensions

The first and foremost consideration for any developer planning to extend the work described here should be to design the hardware layout and enlist a professional costumer to sew the electronics in place as quickly as possible. This will eliminate the vast majority of development headaches and crippling system failures produced by performing an amateur sewing job.

The next major area for expansion is adding capability to detect more than just simple, arm-based gestures to the system. In order to do this, more analog inputs or digital sensors are required. At present, there are no LilyPad motion sensors available with digital output, but LilyPad protoboard is available, which could be used to create a custom LilyPad board for a through-hole digital output motion sensor. For creating extra analog inputs, there are two main approaches available. One option is to use LilyPad protoboard to create a sewable analog multiplexor, which would allow for one Arduino to handle a multitude of different analog inputs. One Arduino processing a large number of analog inputs may have serious responsiveness concerns, however, due to the time required to multiplex and convert all of the analog data, so an alternative is to add multiple Arduinos to the system to parallelize analog conversion and data processing while adding extra ADC pins to the system. This approach creates a few additional concerns, such as the need to somehow share XBee serial inputs, extra power draw, and some impedance on wearability due to the extra hardware. Adding extra XBee modules to the system is strongly advised against, as they are the bulkiest part of the system and the greatest impediment to wearability.

Other suggestions for non-accelerometer sensors include: Gyroscopes for 1-to-1 motion detection, flex sensors to detect bends in the knees and elbows, and force sensitive resistors for the hands and feet to detect steps and claps.

The system described here uses a skin-tight outfit to ensure the profile of the system is small, but also to ensure the accelerometers are held down to the user's body. An alternative system that merits consideration is to place primary sensors on modular pieces of clothing, such as gloves and socks, with some sort of plug or interface connecting the module to the main system. With careful sewing and application of fabric glue, the lines coming from these interfacing points could form a tightly knit conductive wire bus which would feel to the user no more out of place than a fabric seam. This would allow easier replacement of sensors in case of failure, as they are not directly sewn to the rest of the system, but would also allow the use of non skin-tight fabrics as the sensors would be held close to the body by the socks or gloves.

A much larger expansion of the system is the production of multiple instances of the wearable electronics system, all configured to communicate to one computer endpoint. If the software is restructured so the gesture detection happens on the Arduino, as described in *Software analysis - Responsiveness concerns* above, it would be trivial to have the computer play multiple sounds simultaneously based on the actions of the dancers. With enough performers, this could transform a dance concert from a live visual performance to present music into a live visual performance which generates its own music without the use of a musical or percussive instrument.

Conclusion

Though the final product ultimately failed to meet the majority of its goals (see *Overall System Analysis* above), it provides a solid foundation upon which future work can be built. The lessons learned, especially with regards to the special considerations involved with wearable electronics, should save future developers a large number of headaches and a large amount of development time. In addition, the software suite, especially the gesture detection code, provides a reliable springboard that should only need minor updates as new sensors and motions are added to the system.

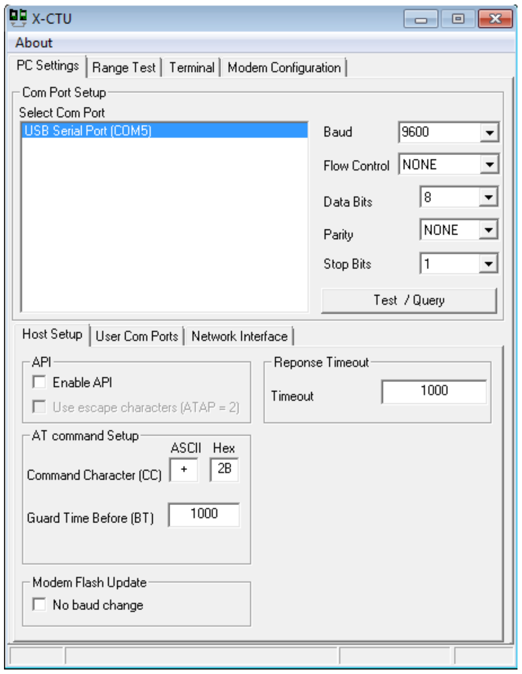
Though not a revolution in dance performance, this project is the first step in that direction, and a valuable exploration of the world of wearable electronics.

Appendix A – Configuring the XBees

Configuring the XBees for point to point communication requires both a method of connecting the XBees to a PC, and the X-CTU configuration software. SparkFun Electronics offers a USB Serial connector for XBees, and the X-CTU software can be downloaded for free from Digi International along with the drivers for the XBee devices of choice.

Figure A.1 shows what the X-CTU software looks like on startup. If more than one COM port is displayed, use the Test/Query button to determine on which COM port the XBee is operating. After the correct COM port has been selected, click the Modem Configuration tab.

Figure A.2 shows an example of the Modem Configuration tab. Click 'Read' to bring up the modem's current configuration. Select PAN ID to use for both XBees as well as a common channel. Then select an ID for each XBee – in the example shown, the IDs are 0 and 1. Enter the ID of the modem being configured into the MY field and the ID of the partner modem in the DL field. Leave the other fields unaltered. Verify the changes by checking the fields marked in yellow, and click 'Write' to commit them.



Com Port Setup
Select Com Port
USB Serial Port (COM5)

Baud 9600
Flow Control NONE
Data Bits 8
Parity NONE
Stop Bits 1

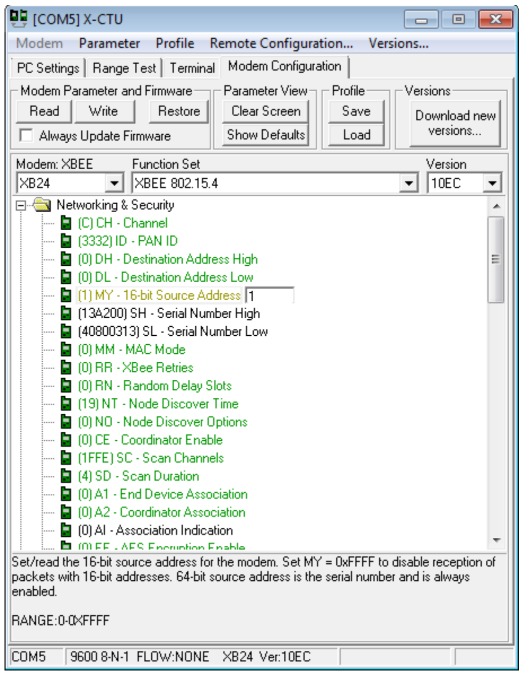
API
Enable API
Use escape characters (ATAP = 2)

Reponse Timeout
Timeout 1000

AT command Setup
ASCII Hex
Command Character (CC) + 2B
Guard Time Before (BT) 1000

Modem Flash Update
No baud change

Figure A1 – X-CTU Startup Screen



Modem: XBEE XB24
Function Set XBEE 802.15.4
Version 10EC

Networking & Security
(C) CH - Channel
(3332) ID - PAN ID
(0) DH - Destination Address High
(0) DL - Destination Address Low
(1) MY - 16-bit Source Address 1
(13A200) SH - Serial Number High
(40800313) SL - Serial Number Low
(0) MM - MAC Mode
(0) RR - XBee Retries
(0) RN - Random Delay Slots
(19) NT - Node Discover Time
(0) NO - Node Discover Options
(0) CE - Coordinator Enable
(1FFE) SC - Scan Channels
(4) SD - Scan Duration
(0) A1 - End Device Association
(0) A2 - Coordinator Association
(0) AI - Association Indication
(0) EE - AES Encryption Enable

Set/read the 16-bit source address for the modem. Set MY = 0xFFFF to disable reception of packets with 16-bit addresses. 64-bit source address is the serial number and is always enabled.

RANGE:0-0XFFFF

COM5 9600 8-N-1 FLOW:NONE XB24 Ver:10EC

Figure A2 – Modem Configuration Tab

Appendix B – Arduino Sketch Code

```
#include "HardwareSerial.h"

extern HardwareSerial Serial;

//The setup function is called once at startup of the sketch
void setup() {
  Serial.begin(9600);
  pinMode(13, 1);
  pinMode(2, 1);
  pinMode(10, 1); //Pins 9 and 10 provide power and ground to the left had accel
  pinMode(9, 1);
  digitalWrite(10, 0);
  digitalWrite(9, 0);
}

bool started = false;

// The loop function is called in an endless loop
void loop() {
  if (Serial.available() > 0) {
    char c = Serial.read();
    if (c == 1) {
      started = true;
      digitalWrite(13, 1);
      digitalWrite(10, 1);
      delay(2);
    }
    if (c == 0) {
      started = false;
      digitalWrite(13, 0);
      digitalWrite(10, 0);
    }
  }

  if (started) {
    for (uint8_t i = 0; i < 6; i++) {
      uint8_t val = (analogRead(i) >> 2);
      if (val == 0) {
        val += 1;
      }

      Serial.write(val);
    }

    Serial.print('\0');
    delay(40); //40 ms delay so computer can keep up
  }
}
```

Appendix C – Java Classes

Appendix C.1 – ADCDataReader

```
import gnu.io.CommPort;
import gnu.io.CommPortIdentifier;
import gnu.io.SerialPort;
import java.io.DataInputStream;
import java.io.OutputStream;

public class ADCDataReader {
  DataInputStream in;
  OutputStream out;

  //portName is the Serial port to use. EG: COM5.
  public ADCDataReader(String portName) throws Exception {
    CommPortIdentifier portIdentifier = CommPortIdentifier
        .getPortIdentifier(portName);
    if (portIdentifier.isCurrentlyOwned()) {
      throw new Exception("Error: Port is currently in use");
    } else {
      CommPort commPort = portIdentifier.open(this.getClass().getName(), 2000);

      if (commPort instanceof SerialPort) {
        SerialPort serialPort = (SerialPort) commPort;
        serialPort.setSerialPortParams(9600, SerialPort.DATABITS_8,
            SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);

        in = new DataInputStream(serialPort.getInputStream());
        out = serialPort.getOutputStream();
      }
      else {
        throw new Exception("Not a serial port");
      }
    }

    out.write(1); //Notify Arduino that the PC is ready
  }

  public int[] readData() throws Exception {
    byte[] data = new byte[7];
    int[] ret = new int[7];

    while (true) {
      in.readFully(data);
      if (data[6] != 0) {
        synchronize();
      } else {
        break;
      }
    }

    //Bytes are signed - convert to unsigned values 0-255
```

```
      for(int i = 0; i < data.length; i++) {
        ret[i] = (data[i] & 0xFF);
      }

      return ret;
    }

    private boolean synchronize() throws Exception {
      int bit;
      while ((bit = in.read()) != 0 && bit != -1)
        ;

      return bit != -1;
    }
  }
```

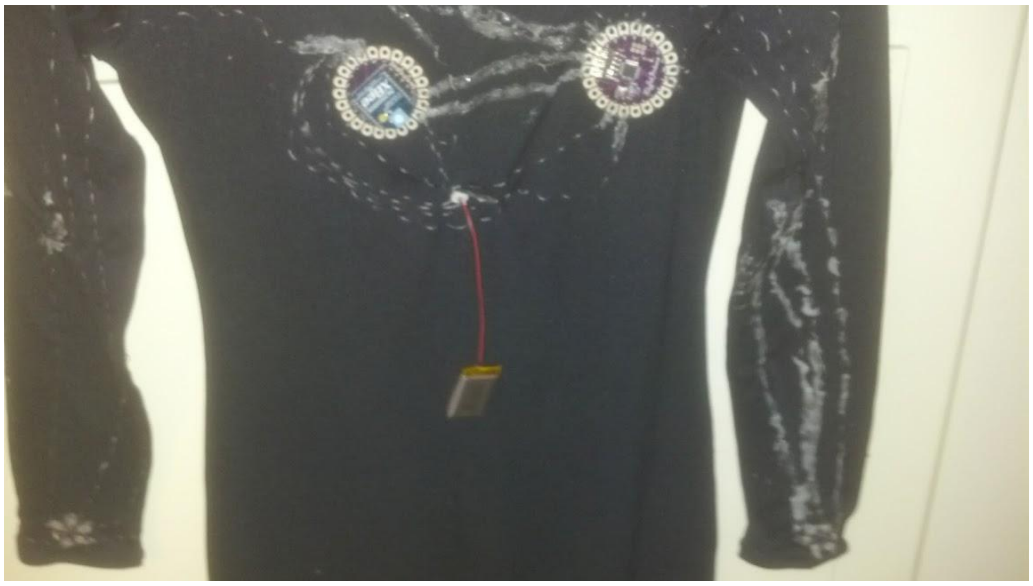
Appendix C.2 – GestureDetector

```
import java.util.LinkedList;

public class GestureDetector {
  public enum Gesture {
    XY_CLOCKWISE, //Hand moving to the right
    XY_COUNTER, //Hand moving to the left
    ZY_DOWN, //Downards hand "slam"
    ZY_UP, //Upwards hand "raise"
  }

  int[] minThresholds;
  int[] maxThresholds;
  int timeThreshold;

  //Java linked list has queue behavior
  LinkedList<Integer>[] dataPoints = new LinkedList[3];

  boolean anyTrigger = false;
  long maxTrigger[] = new long[3];
  long minTrigger[] = new long[3];

  long time = 0;
  int safeTime = 0;

  public GestureDetector(int[] minThresholds, int[] maxThresholds, int timeThreshold)
{
    if (minThresholds.length != 3 || maxThresholds.length != 3) {
      throw new IllegalArgumentException("Gesture detector requires exactly " +
              "3 max and min thresholds, one each per axis");
    }

    if (timeThreshold < 0) {
      throw new IllegalArgumentException("timeThreshold must be greater than" +
              "or equal to 0");
    }

    this.minThresholds = minThresholds;
    this.maxThresholds = maxThresholds;
    this.timeThreshold = timeThreshold;

    for (int i = 0; i < 3; i++) {
      dataPoints[i] = new LinkedList<Integer>();
      for (int j = 0; j < 3; j++) {
        dataPoints[i].push(128);
      }
    }
  }

  public Gesture processData(int[] data) {
    Gesture ret = null;
    time++;
    if (data.length != 3) {
      throw new IllegalArgumentException("Data must be 3 in length");
    }

    calculateExtrema(data);

    if (anyTrigger && safeTime > timeThreshold) {
      int lateral = -1;
      int centripetal = -1;

      for (int i = 0; i < 3; i++) {
        if (maxThresholds[i] != 0 && minThresholds[i] != 0) {
          lateral = i;
          break;
        }
      }

      for (int i = 0; i < 3; i++) {
        if (i == lateral) {
          continue;
        }

        if (maxThresholds[i] != 0 || minThresholds[i] != 0) {
          if (maxThresholds[i] == 0 || minThresholds[i] == 0) {
            centripetal = i;
          }
        }
      }

      if (lateral != -1 && centripetal != -1) {
        ret = determineGesture(lateral, centripetal);
      }

      reset();
    }

    return ret;
  }

  private Gesture determineGesture(int lateral, int centripetal) {
    if (lateral == 0 && centripetal == 1) {
      if (minThresholds[0] > maxThresholds[0]) {
        return Gesture.XY_CLOCKWISE;
      }
      else {
        return Gesture.XY_COUNTER;
      }
    }
    else if (lateral == 2 && centripetal == 1) {
      if (minThresholds[2] > maxThresholds[2]) {
        return Gesture.ZY_DOWN;
      }
      else {
        return Gesture.ZY_UP;
      }
    }

    return null;
  }

  private void reset() {
    anyTrigger = false;
    safeTime = 0;
    time = 0;
    for (int i = 0; i < 3; i++) {
      maxTrigger[i] = 0;
      minTrigger[i] = 0;
    }
  }

  private void calculateExtrema(int[] data) {
    boolean allSafe = true;

    for (int i = 0; i < 3; i++) {
      dataPoints[i].pop();
      dataPoints[i].add(data[i]);

      if (dataPoints[i].get(1) > maxThresholds[i]) {
        allSafe = false;
        if (dataPoints[i].get(1) > dataPoints[i].get(0) &&
            dataPoints[i].get(1) > dataPoints[i].get(2)) {
          maxTrigger[i] = time;
          anyTrigger = true;
        }
      }
      else if (dataPoints[i].get(1) < minThresholds[i]) {
        allSafe = false;
        if (dataPoints[i].get(1) < dataPoints[i].get(0) &&
            dataPoints[i].get(1) < dataPoints[i].get(2)) {
          minTrigger[i] = time;
          anyTrigger = true;
        }
      }
    }

    if (allSafe) {
      safeTime++;
    }
  }
}
```

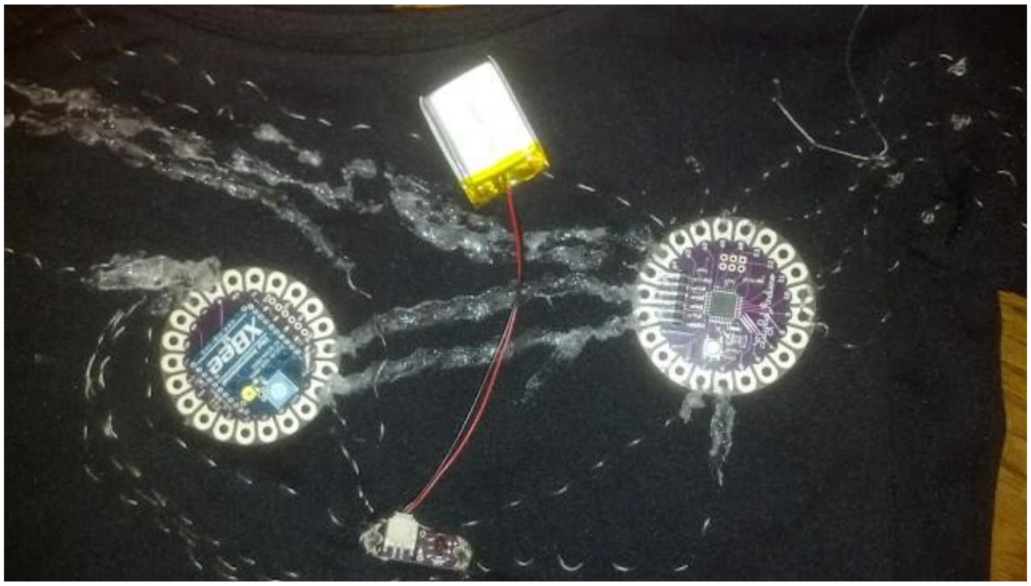
Appendix C.3 – MIDIExample

```
import javax.sound.midi.Instrument;
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.Synthesizer;

//Note these examples are independent blocks of code - some code is redundant
public class MidiExample {
  //List all instruments by name and number on the console
  static void ListInstruments() throws Exception {
    Synthesizer synth = MidiSystem.getSynthesizer();
    Instrument[] instruments = synth.getAvailableInstruments();

    for (Instrument instrument : instruments) {
      System.out.println(instrument.getPatch().getProgram() + ": " +
          instrument.getName());
    }
  }

  //Play a note for half of a second
  //Pitch and volume must be between 0 and 127, inclusive
  static void PlaySound(int instrument, int pitch, int volume) throws Exception {
    Synthesizer synth = MidiSystem.getSynthesizer();
    synth.open(); //Call only once in a program

    //Channels 0-7 are for general instruments
    //Channel 8 is special for percussion - every pitch is a unique
    //percusison instrument when channel 8 is used.
    MidiChannel channel = synth.getChannels()[0];

    channel.programChange(instrument);
    channel.noteOn(pitch, volume);
    Thread.sleep(500);
    channel.noteOff(pitch, volume);
  }
}
```

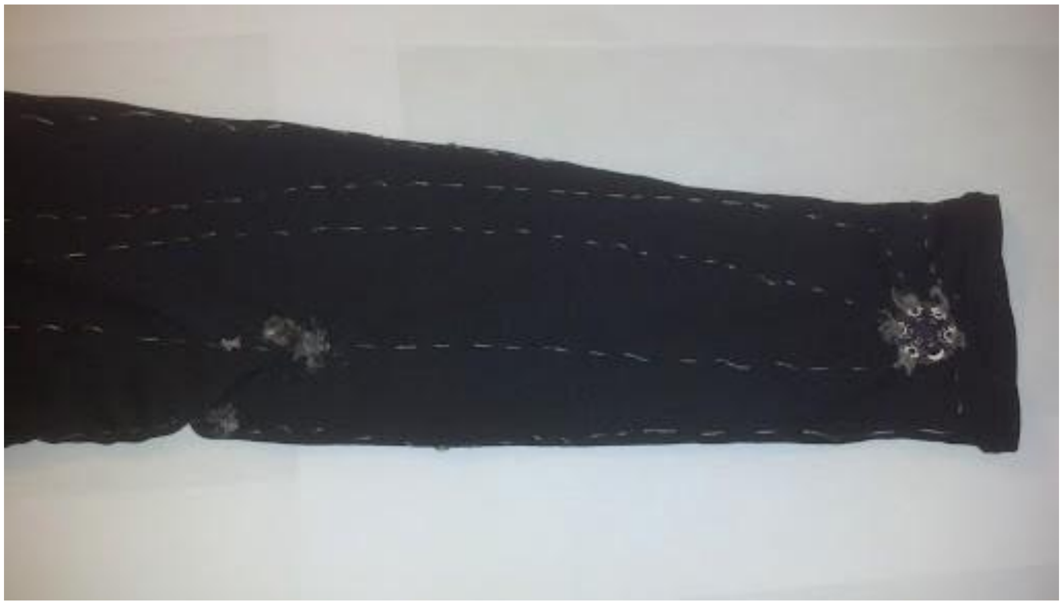
Appendix D – Project Images



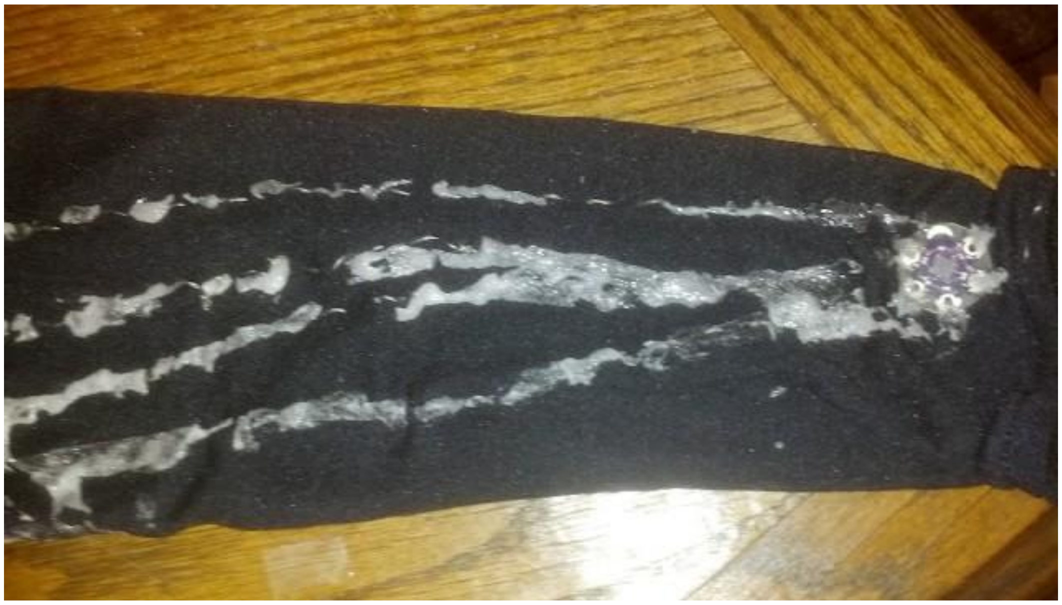
Complete System Photograph



Close-up of system "heart." Note small profile of LiPo battery and connector.



Close-up of Right Arm – Relatively clean sewing job



Close-up of Left Arm – Excessive fabric glue applied

References

[1] Buechley, Leah. Lilypad Website. http://web.media.mit.edu/~leah/LilyPad/

[2] Goh Chun Fan, Fitriani, Wooi-Boon Goh. “Generic Motion Gesture Detection Scheme Using Only a Triaxial Accelerometer.” 2011 IEEE 15th International Symposium on Consumer Electronics.