Gesture Control Interface

April 12th, 2016

Wearable gesture detection device for human-computer interfacing

Summary

A glove that allows the user to interact with a computer through gestures detected by embedded sensors.

Eric Smith

Rajan Jassal

Chris Chmilar

Abstract

The goal of this project was to design and build a working prototype of a glove that recognises a user's hand movements and uses them to send keyboard commands to a computer. The final design of the glove has a motion processing unit (MPU) attached to the back of the hand and four buttons attached to the tips of the fingers (one on each excepting the thumb). A perfboard worn on the forearm houses the power supply, wireless transmitter, and a programmable microprocessor. When the button on the index finger is held down, samples of acceleration data are gathered by the microprocessor and sent wirelessly to an FPGA development board (Altera DE2). When the button is released again a final character is sent signalling to the DE2 that it should begin analyzing the data. A defined "gesture" from a series of data points, from button press to button release, is collected. After the end signal is received by the DE2 it categorizes the gesture into one of five recognized gestures: up, down, left, right, or no movement. When one of these gestures is recognized, a set of key presses is sent across a USB connection to the host computer. The DE2 board is setup to be recognized as an HID keyboard on most modern operating systems, allowing key presses to be interpreted as they would be if typed normally. Using this design, users are able to execute functional keyboard shortcuts such as alt-tab using hand movements.

Table of Contents:

Functional Requirements

The requirements are to provide a way to intuitively interact with your computer using gestures. Fundamentally this means translating data gathered from movement sensors into commands the computer will recognize, and communicating these commands effectively. Gestures are primarily for convenience interactions such as scrolling and switching context.

Primary features included are:

- Scrolling on a webpage or document (or any focused window with a scroll bar)
- Switching between windows
- Provide a state in which users

Except for switching tabs, these features were all implemented successfully. Scrolling on webpages was not implemented by simulating a mouse wheel (as was anticipated) but through the use of gestures to simulate a page up and page down key press. Navigating through open windows was implemented as expected, by simulating alt-tab and alt-shift-tab key press with gestures. The locking mechanism was also implemented but in a much different way than initially planned. The concept for the locking mechanism of the glove involved two states: a locked state and an unlocked state. While in the unlocked state the glove would recognize all gestures that the user made until they entered the locked state in which no gestures would be recognized. This method involved continuous gesture recognition which led to an accumulation of errors in the recognition as time went on (i.e the longer the user stayed in unlocked state). The cause of this was the implementation of gesture recognition chosen. Due to this fact it was decided that the glove would be in locked state (no gesture recognition) by default and in order to implement a single gesture the user would hold down on a button, make a single gesture, and then let go of the button in order to symbolize the gesture was complete. This allowed for the gesture recognition algorithm used to process only on gesture at a time, which lead to less error in gesture recognition.

Design and Description of Operation

As mentioned in the requirements section, data will need to be collected and analyzed from a movement sensor. The movement sensor itself will be embedded on the top of the hand of a wearable glove with a wristband (hereafter referred to as the glove. See figure 2). Raw data collected from this accelerometer sensor will be sent to a DE2 board through a wireless communication device where it will be analyzed: i.e. the hand movement pattern represented by the data will be categorized into a gesture. A signal is sent to the computer through the board's USB interface that represents the action we want it to take based on the gesture performed.

There is an additional source of data collected and transmitted that is gathered by completing circuits using buttons on the fingers. These will be used to determine the 'state' of the device, which will modify what action each gesture is interpreted as. The glove will initially be in standby state (no acceleration data will be sent by the glove). When the button on the index finger of the user's hand is clicked the glove will go into collection state, where it will collect acceleration data from the MPU. The user will execute the gesture they desire while the index finger button is held down. Once the user is done making the gesture they desire they will release the button and glove component will send the collected data to the DE2 in order to process it with the gesture recognition algorithm. A microprocessor chip on the glove polls each source of data and schedules sending information to the transmitter.

States:

- Standby state - The glove will not send any data. No actions will be performed
- Collection state - Starts when the button on the index finger is held down. Acceleration data will be collected while the index finger button is held down and then sent when the index finger button is released. The gesture that is performed while the index finger button was held down is executed (performs window switch, page up or page down in current context)

The primary gestures implemented using the accelerometer will be the detection of vertical and horizontal movement. These will be used to perform document scrolling (page up and page down functionality) and tab changing (alt-tab and alt-shift-tab keyboard functionality) respectively.

Table 1) Primary gestures to be implemented

Gestures	Action (keyboard press) performed
Upwards vertical hand motion	Page up
Downwards vertical hand motion	Page down
Left Horizontal Motion	alt-shift-tab
Right Horizontal Motion	alt-tab

Figure 1) Example execution of an up gesture

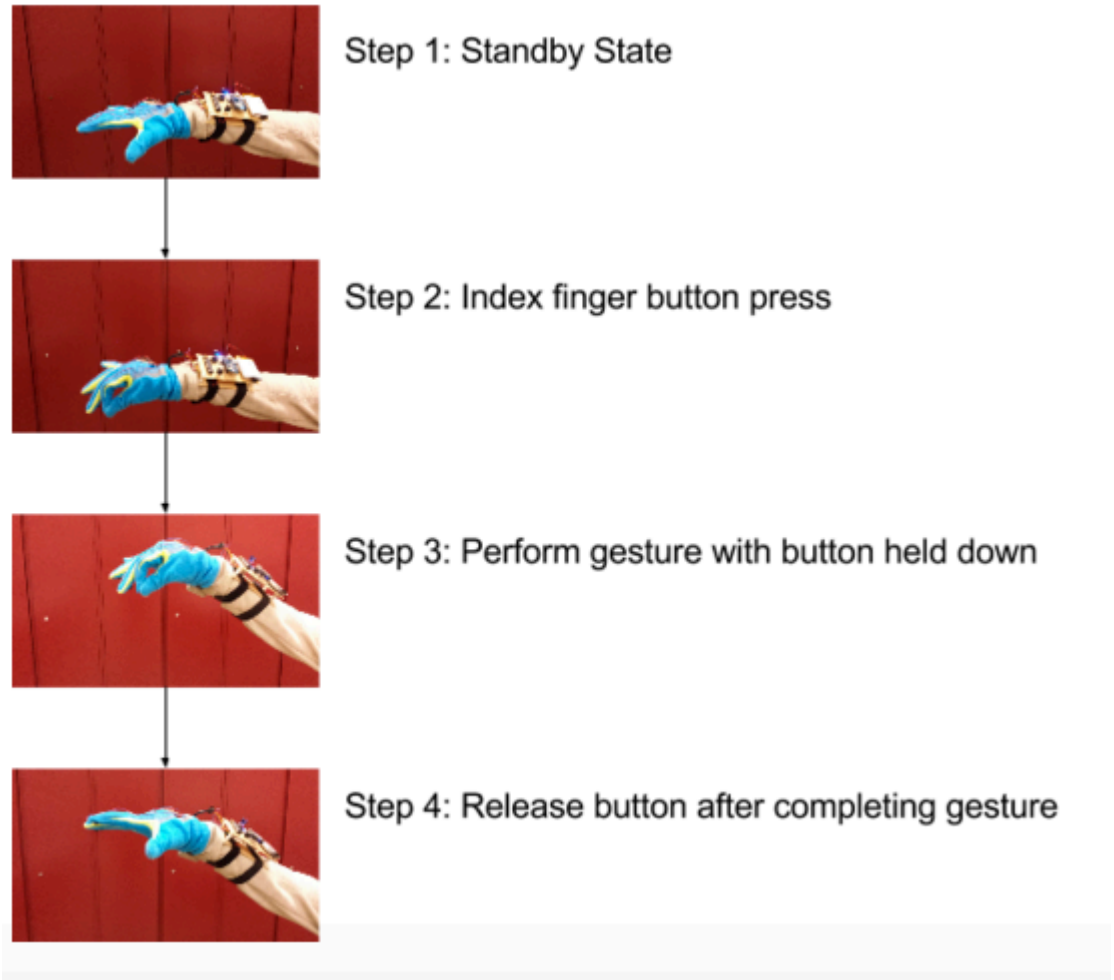
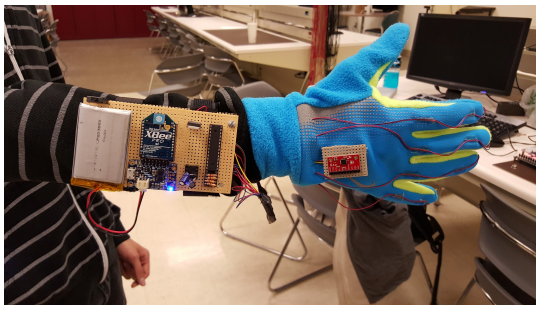
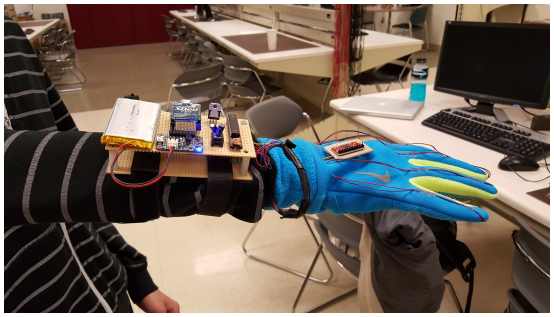


Figure 2) The glove in palm left orientation (left) and palm down orientation (right) and palm



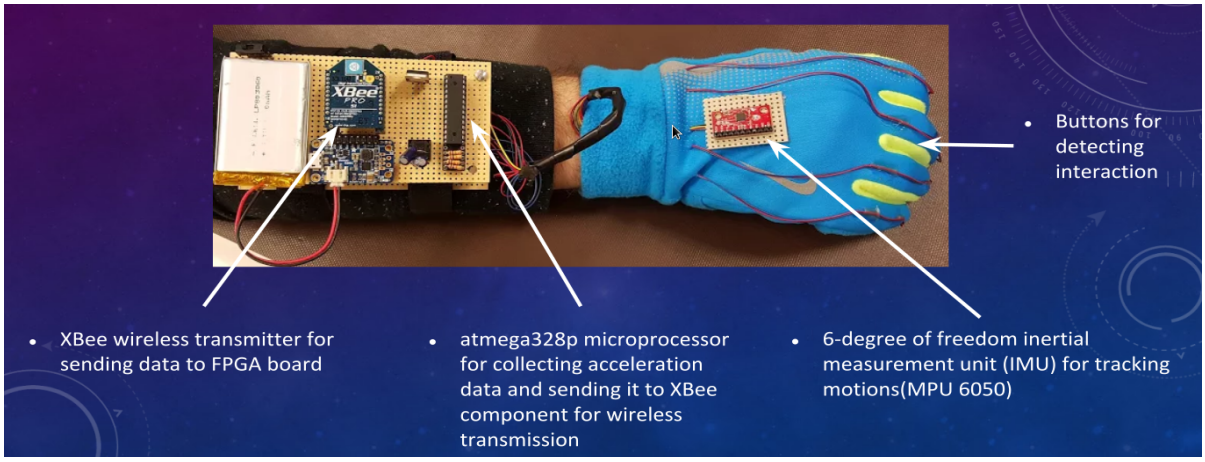


The design of the project can be split into two modules. The glove module is composed of all the hardware and software components that make up the gesture-tracking glove. The DE2 module is made up of the gesture recognition algorithm and the USB communication to the computer.

The Glove Module

The glove module is composed of all the hardware on the glove and is responsible for gathering all the acceleration data collected by the accelerometer and sending it to the DE2 module. An MPU 6050 accelerometer is mounted on the top of the glove and is connected to the circuit mounted on the wrist. Below is a labeled diagram of that circuit.

Figure 3) Labeled glove component



The circuit uses an ATmega328p microprocessor to gather incoming raw data from the MPU-6050 accelerometer (when index finger button is pressed down). This raw data is calculated into two forms of readable values: G's and rotational velocity. All three axes (X, Y, and Z) will provide values in these two forms, for a total of 6 values. For this project, only the Y and Z acceleration values will be used in gesture recognition, and thus the other 4 values will not be sent from the glove (except for the first initial position sent, which will send all 3 accelerations). The microprocessor will detect the index button being initially pressed, and will package a serial data stream of initial accelerations in X, Y, and Z. This information will allow the DE2 to figure out which orientation the palm is facing when operation begins, ensuring correct gesture movement in any palm orientation. The ATmega328p sends this flow of data serially using UART interfacing through its transmission pin, directed to the data in pin of the transmission XBee. The ATmega328p uses I2C to interface with MPU-6050 and UART to interface with the Xbee transmitter. The XBees used in this project uses the ZigBee standard, which provides a wireless, serial communication network that can be operated by up to 200 feet away, and accepts a UART serial stream of data. The ATmega328p must also process the data from the MPU and send it off to the UART in a form that will recognizable by the DE2 component. This process is discussed further in software design. The MPU-6050 provides data in a double format which is tough to send and requires a lot of resources in the DE2 module in order to execute floating point operations. In order to get around this, fixed pointed operations will be used. To implement this the microprocessor will multiply the acceleration data by 1000 (as to not lose data beyond the decimal point) and cast it to a integer before it is sent. The entire glove is powered by a 3.7V battery, power boosted to 5.1V, that is regulated down to 3.3V. This is due to the MPU and XBee being sensitive to voltages above the 3.3V threshold.

The DE2 module

The DE2 module is required to receive the data sent from the glove through an Xbee receiver, process the data with a gesture recognition algorithm, and send the corresponding keyboard shortcut (that is linked to the gesture on table 1) to the computer through USB.

The DE2 is connected to an Xbee receiver through the serial DB9 connector. To do this the Xbee receiver must be mounted on the XBIB-R-DEV XBee development board that will be serially connected to the DE2 through a null modem connection. To use the DB9 port on the

DE2 the university IP core for the serial module was used to create a Qsys system with access to the serial port. This IP core provided all the functions necessary to store incoming transmissions in a buffer(so no data is lost) and then read the transmissions character by character.

The gesture recognition section of the DE2 component would take the data that is read in serially and attempt to determine what gesture the data represented. The data that is transmitted from the glove would be sequences of accelerations in the two axes (y and z in units of G's) polled at a speed of 1MHZ(clock speed of the Atmega328p). Since the algorithm only has to detect gestures that were linear motions in one axis (the gestures specified in table 1) it was determine the most efficient method of gesture detection was approximating the integrals of the acceleration data(see figure 4). By integrating the acceleration data we were able to obtain the approximated velocity sequence of the gloves movement. By averaging this velocity data the direction of movement was determined (by the sign of the average velocity in a particular axis). This direction of movement mapped directly to a gesture (up, down, left, or right) because only linear movement needed to be analysed. This also indicated that only acceleration data from the z and y axes was needed to determine what just gesture was executed (z changes map to vertical movement and y changes map to horizontal movement).

Figure 4) The integration acceleration

$$v_i = \int a_i * dt$$

Figure 5) The integration approximation used

$$v_{i+1} = v_i + a_{i+1}T$$

Figure 5 shows the actual integration approximation that was calculated in our program. Here v_{i+1} was the new velocity value being calculate for the i+1 time interval, $v_i$is the old value from the last time interval i (when i = 0 v is assumed to be 0 because the glove should start at rest), $a_{i+1}$is the acceleration data that was sent over from the glove for time interval i+1, and T is the time interval between intervals i and i+1. Note because the glove polled the accelerometer at discrete time intervals, T was assumed to be 1 between time intervals that data was gathered. This integration approximation was done on the y and z-axis to create the velocity sequence of the glove in these axes (that represented the speeds it traveled in each axes at during the gesture). The average velocity of each axes was calculate and the largest absolute velocity

determined which axis and direction the glove moved in the most. This allowed the gesture the glove made to be determined based on the sign of the velocity with the highest absolute value. For example if on the completion of a gesture (with your palm facing down) the average velocities in the y and z axes were 900 and -1600 respectively, the gesture this movement would recognize would be the downward vertical hand motion because 1600 is the highest value indicating there was movement in the z axis and this value was negative in the z axis indicating the glove traveled down. The units of the velocity measurements are m/s(relative to the acceleration of gravity) but in the above example the magnitudes were large because as stated above all numbers were multiplied by a factor of 1000 in order to use fixed point arithmetic on the DE2. It should also be noted that velocity thresholds of 1500 were set in order to get rid of noise and slight movements of the glove. This means that any absolute value less than 1500 was seen as no movement by the gesture recognition and if no absolute average velocity was higher than 1500 then no keyboard shortcut would be executed.

The project also has functionality to work in multiple hand orientations. Figure 2 shows the glove in the two most tested positions, palm left and palm down. The glove will also work in the palm right and palm down orientations but as these are uncomfortable positions not much testing was done on these orientations. The orientation is determined by the initial measurement take when the index finger button is held down. Due to the fact accelerometer readings are based on gravity it was easy to determine the orientation of the glove from this initial measurement. For example in the palm down orientation Z will -1G because the force of gravity is pushing down on the z-axis will be 1G(9.81m/s^2). However for the palm left orientation the force of gravity on the Y-axis will be 1G. By comparing the initial conditions of the glove, orientation can be determined with a number from a number of cases of expected values. With the orientation by shifting the direction each axis represents the gesture in a specific orientation can be determined the same as above. An example is in the palm left orientation (figure 2) the accelerometer orientation would indicate that the y axis would translate to vertical motion and the z to horizontal motion.

The implementation of the gesture recognition lead to a number of unaccounted cases that would lower the success rate of our glove. Since the initial velocity is always assumed to be 0, starting from a non-rest state would lower the chance of the average velocity of each access to be calculated correctly. It was also found the accelerometer would work best when it lay at a 90 degree angle (in any orientation). The project would still work with slight bends in the palm but its performance would not be as great. Since the gesture recognition only looked for linear

motions, nonlinear motions (i.e diagonals or circles) were not accounted for and would simply translate to the gesture associated with the direction the glove traveled fastest in.

USB connection

A USB connection is established between the DE2 board and a host computer to which the keyboard commands are intended to be sent. The computer, as the host on the USB bus, interrogates the DE2 to determine what kind of device it is. The device is configured in such a way to inform the host that it has an HID keyboard interface so that a standard HID keyboard driver is loaded to interpret information sent by the device. The host is also asked to poll the device every 20 ms to see if there is an update available; If the device has information to send, a report containing the set of keys pressed is sent across to the host. The set of keys sent corresponds to either those mapped to a gesture, or if no gesture was recently made, to the default set of $keys_1$. Note the delay in receiving and interpreting a gesture from the glove make it impossible to override the last gesture by quickly making another before the information is sent to the host.

Next we will look at the USB connection in detail.

The DE2 microcontroller board has three USB ports labeled ‘blaster’, ‘device’ (USB B port), and ‘host’ (USB A port). The blaster is the connection that is used to program the device and to communicate between the board and the NIOS II console in Quartus. The device port, as its name suggests, is the port through which the connection to communicate with a host computer as a USB device is established. The host port is not used in this project.

The DE2 has a USB controller chip, the Phillips ISP1362, that controls the low level operation of the host and device ports (referred to hereafter as either the ISP1362 or the USB controller). The ISP1362 has 14 programmable endpoints that can serve as custom IN or OUT communication pipes and 2 fixed control endpoints (one IN and one OUT) that are used to send and receive USB requests from the host. In the USB specification, the IN direction always refers to data sent to the host and OUT always refers to data leaving the host. Only one of these programmable endpoints is used in this project which will be discussed in more detail later.

The university IP core library for Quartus 13.0sp1 includes a “USB controller” component which is added to the design in Qsys. This provides an interface to the ISP1362 through which the software can send commands and write data to its various registers. It also provides a hardware interrupt that is triggered by the ISP1362 when one of its endpoints requires interaction. Setting

up the USB controller involves writing to several registers to configure whether it should try to connect, how hardware interrupts should be sent, settings of each endpoint etc. Buffer space is assigned to each enabled endpoint after all endpoints have been configured.

When a packet is sent to an OUT endpoint from the computer it is placed in the associated buffer and can be read; similarly, a packet is sent to the host computer when new data is written to an IN buffer. The actual timing of when the packets are sent is dependant on the endpoint type and scheduling of both the host and USB controller chip. This project uses the interrupt endpoint type for the keyboard interface, which is polled by the host periodically and the default bulk endpoint type for the control endpoints, which transfer packets with low priority but with high reliability of transmission (requires handshakes).

When the USB cable is connected from device to host the host interrogates the device. Through standard USB requests (see USB specification for details) a series of ‘descriptors’ are requested. These descriptors are standardized data structures transmitted byte by byte to the host that describe the details and configuration of the device. The first descriptor requested is the device descriptor; without going into details that are better covered in the USB specification, it describes a variety of details about the device including the vendor ID, product ID and importantly, the maximum packet size that can be sent and received from the control endpoints. It also normally communicates the device class (one of many specified in the USB standard), though for an HID device like this project, it is often left at 0, deferring the class selection to the next descriptor. The next descriptor is the configuration descriptor.

Each USB device has settings that are organized into a hierarchy; at a given time, a device may have one configuration, which can have multiple interfaces, and each of these can have multiple endpoints (recall from earlier that each represents a communication pipe between device and host). The host can request that a device change to a different configuration, but it is rare that a device has more than one. The configurations, interfaces, and endpoint settings are described and communicated to the host in the configuration descriptor. If the device has an interface of class “human interface device” (HID) than an additional descriptor is transmitted along with the configuration that describes some details about the HID interface and also notifies the host that a third descriptor, called the HID report descriptor, should be requested.

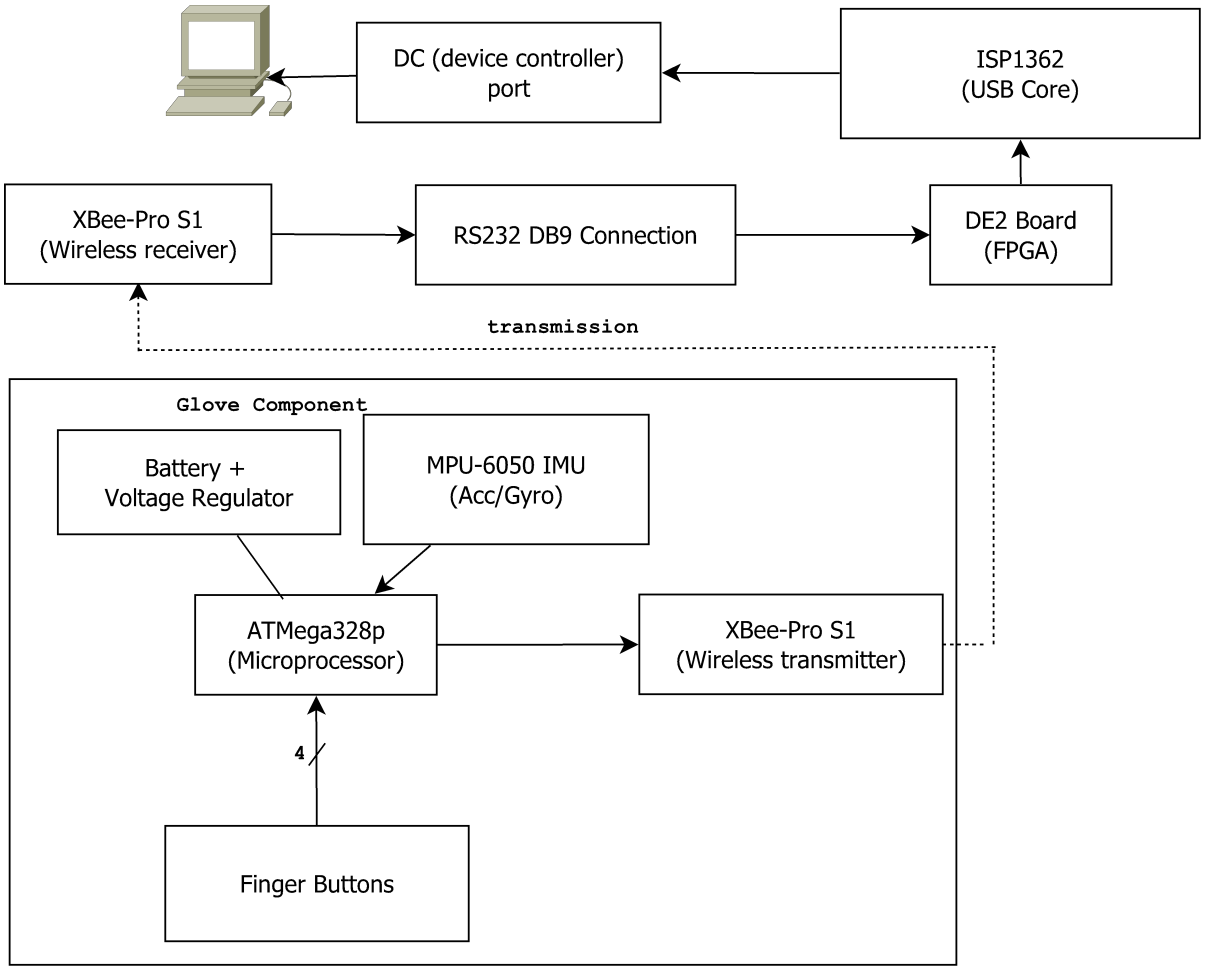
The device for this project only has one interface that has one endpoint. This interface specifies that it is an HID Keyboard and has one interrupt endpoint that should be polled at minimum every 20 ms. The host assigns an HID keyboard driver to receive and interpret packets sent from endpoints on this interface. If this project were to include the additional functionality of a mouse or a custom driver, this descriptor would include an additional (or multiple additional) interfaces. These interfaces would have a class that informed the host of the driver needed to handle information sent from their own endpoints. In the case of having multiple interfaces, like an HID mouse and an HID keyboard, two drivers would be loaded and the endpoints handled separately.

When the HID keyboard sends data across the USB connection it does so in the format of its HID report. After receiving the configuration descriptor that tells the computer that the device has an HID interface, it requests the HID report descriptor. This descriptor describes how the reports to be sent across the endpoint will be formatted. Our reports are formatted the way a typical HID keyboard is formatted as suggested in the HID 1.11 specification. Eight bytes long: the first is a bitmap of modifier keys held down (includes cntl, alt, and shift on each side of they keyboard plus two reserved bits); the second is a reserved byte for padding; and the last six bytes are each for a key code that represent keys on a typical keyboard. The encoding of these keys is also described to the host in the HID report descriptor. This means that only six keys can be pressed at the same time, though this is well above the maximum possible used by this device.

When the computer receives a report containing key presses from the interrupt endpoint, it sends an acknowledgement packet back to the DE2, which causes the USB controller to trigger a hardware interrupt on endpoint 1 (the interrupt endpoint associated with the HID keyboard interface). After receiving an interrupt on interrupt one the device sends another report with no keys pressed (signifying that all keys were released) so that every gesture only sends one command rather than the many you would get from holding down a key on a conventional keyboard. The modifier keys are not released after a gesture automatically; in this way we remain in the alt tab menu until another gesture is made that clears the alt, such as the no movement gesture.

Block Diagrams:

Figure 6) Data Flow and Hardware Resources Block Diagram



Bill of Materials

5V 2A power supply required, not included in total

Part Name	**Cost**
Altera/Terasic DE2 Development Board	$ 517.72
MPU-6050 Accelerometer Breakout Board	$ 55.74
2x XBee Pro S1	$ 83.62
XBIB-R-DEV XBee Development Board	$ 97.66
Null Modem DB9 Cable	$ 5.99
DB9 Gender Changer	$ 4.99
AVR ATMega32P-PN 28-Pin Microprocessor	$ 5.79
Pocket AVR USB Programmer	$ 19.29
Adafruit Powerboost 500c Charger	$ 19.42
803860 3.7 V Lithium-Polymer Battery	$ 19.42
Perfboard	$ 7.50
Glove	$ 15.00
Velcro Strips	$ 3.49
Toggle Switch	$ 0.70

4x Button Switches	$ 2.00
4x 10KOhm Resistors	$ 0.60
2x 10uF 50V Electrolytic Capacitors	$ 0.90
7x Screws	$ 0.14
4x Plastic Spacers	$ 1.32
Total Cost	$ 861.29

Available Sources

- xbee_ansic_library from Digi. GitHub repository made by Digi[1]
- AVR Ultimate Driver Pack. Used for the MPU6050 libraries, as well as i2cmaster library. GitHub repository made by Martin K. Schroder[2]
- Previous ECE 492 projects involving Glove Interfacing as well as XBee Communication.
- Various tutorials on manufacturers websites
- University IP cores provided by Altera

[1] Digi (2012, November 26), xbee_ansic_library[Online], Available: https://github.com/digidotcom/xbee_ansic_library

[2] M.K.Schroder (2014, November 26). AVR Ultimate Driver Pack [Online], Available: https://github.com/mkschreder/avr-ultimate-driver-pack

Data Sheet

Operating Conditions

The DE2 board must be operated within a temperature of 15 to 32 degrees Celsius. Avoid rapid temperature change to prevent the introduction of condensation to the board.

Before operation of the glove, users must be relatively grounded to avoid static shock of components. The MPU-6050 chip must maintain a voltage of 2.375-3.46 V, and any static shock could cause a voltage spike, which could damage the accelerometer.

Putting the glove on, please ensure that no wires leading from the wrist board to the glove are unseated or disconnected. Improper connections to the glove will lead to improper/absent readings, or damage to the accelerometer.

IO signals

ATMega328p	
Inputs Pins	**Connection Description**
VDD (pin 7)	3.3V from ‘Power Source’ component below
AVCC (pin 20)	3.3V from ‘Power Source’ component below
GND (pin 8)	Common to all glove components
AGND (pin 22)	Common to all glove components
SDA to I2C bus (Digital Pin) [Bidirectional] (pin 4)	Connects to IMU (Inertial Measurement Unit). This component will always be master.

SCL to I2C bus (Digital Pin) [Bidirectional] (pin 5)	Connects to IMU (Inertial Measurement Unit)
Finger Contact Inputs (Analog to Digital Pins) (pin 23-28)	Will poll pins for signal (metal contacts complete circuit to pass high or low value) *Note: This is probably not the way it will end up being in the end, but for now it's all we have.*
XBee Clear_to_send (Digital Pin) (pin 2)	Clear to send signal received from XBee Pro Module
XBee_data_in (N/A)	Not sure if this will be used in our implementation, however it is included in the interface diagram
Outputs	**Connection Description**
Data_out to Xbee Module (Digital Pin) (pin 3)	Main UART data line to XBee Pro Module
Request_to_send (Digital Pin) (pin 11)	RTS input to XBee Pro Module

DE2 Board	
Inputs	**Connection Description**
VDD	Power from socket
GND	Relative to socket, common to all components internal to DE2
XBee Clear_to_send(Digital Pin 3)	Clear to send signal received from XBee Pro Module
PIO input from XBee_data_in	Stream of data received by XBee receiver

Outputs	**Connection Description**
USB Connection to Computer	
XBee_data_out	Unsure if necessary, unless used for ACK

XBee-Pro	
Inputs	**Connection Description**
VDD (Pin 1)	3.3V from ‘Power Source’ component below
GND (Pin 10)	Common to all glove components
DIN (Pin 3)	UART Data In, will be connected to TX pin on ATMega328p
Outputs	**Connection Description**
DOUT (Pin 2)	UART Data Out, will be used on DE2 connected XBee to provide flow of serial data coming from glove

MPU-9150 GYRO ACCEL IMU 9DOF	
Inputs	**Connection Description**
VDD	3.3 Volts. *note: max voltage does not have a high tolerance above this value. Fluctuations in voltage regulator should be watched.*

GND	Common ground to all components on glove
SDA to I2C bus [Bidirectional]	Connects to atMega328p, will always be slave
SCL to I2C bus [Bidirectional]	Connects to atMega328p, will always be slave
I2C Slave Address (LSB)	GND. Since it is the only IMU does not matter.

Power Consumption

Note: DE2 was found to operate in equal current draw and voltage when idle and receiving signal/sending USB

Device	**Current (mA)**	**Voltage (V)**	**Power (W)**
DE2 Development Board	430	8.95	3.89
Glove Component (Idle)	60	5.15	0.031
Glove Component (Transmitting)	280	5.15	1.442

Background Research

Peer Reviewed Sources

A high-performance training-free approach for hand gesture recognition with accelerometer[3]-x From this paper we learned an approach to gesture recognition that did not involve learning(which in resource intensive). While we did not end up using this the method in this paper it still taught us important features of gesture recognition that we had to account for.

Other Sources

Writing device drivers in linux: A brief tutorial[4].- Learned how a computer identifies a USB device
Linux Device Drivers, Third Edition[5] - Learned about HID drivers and their functionality.
SparkFun beginning embedded electronics tutorials[6] - Taught us how to program the ATmega328p microcontroller.

[3] L. Yin et al., "A high-performance training-free approach for hand gesture recognition with accelerometer," *Multimed Tools Appl*, vol. 72, no. 1, pp. 843-864, Sep. 2014.
[4] X. Calvet (2006, April 26). Linux: A Brief Tutorial [Online]. Available: http://www.freesoftwaremagazine.com/articles/drivers_linux
[5] *Linux Device Drivers*, 3rd ed, O'Reilly Media, Inc., Sebastopol, CA, 2005
[6] N. Seidle, (2008, June 19). Beginning Embedded Electronics [Online]. Available:https://www.sparkfun.com/tutorials/category/1

Software Design

The software design of this project involves splitting up the software into three distinct parts. Each is listed below with the sub-tasks each is responsible for:

1. The USB communication module
    a. Setup and configure the USB controller on the DE2
    b. Establishing a connection to a host computer being recognized as a USB device
    c. Configuring the DE2 to be recognized as a HID device by an HID keyboard driver
    d. Sending sets of key presses to the computer whenever a gesture is recognized
2. The glove module
    a. Is programmed on the ATmega328p on the glove
    b. Involves programming the ATmega328p to accept data from all the components on the glove and correctly send it to the DE2 board through the Xbee transmitter
3. The gesture recognition module
    a. Is handled exclusively by the DE2 board
    b. Involves reading in data from the receiver through the DB9 connector on the DE2
    c. Using a gesture recognition algorithm to determine what gesture the read data represents

The USB Communication Module

This section will describe the flow of the software as it relates to the USB communication. The interactions between the USB communication module and the gesture recognition module will not be overlooked, as the run in the same main loop on the DE2, however USB communication will be the focus. The sections will closely follow the layout of the software flow diagram in ***figure 7*** so it may be useful to refer to it now.

When the DE2 is powered on or reset it first has to configure the USB controller chip, the Philips ISP1362. First the ISP1362 is reset. It sets the mode and hardware configuration to reset values and very importantly sets the address that the chip thinks it is on the USB bus to zero. Then the device is configured to connect on the USB bus; the main things here are to set the DcMode register to enable softconnect, to enable power and the setup clock in the DcHardwareConfiguration register, then finally to set up the endpoints on the chip by writing to the DcInterruptEnable register, and the DcEndpointConfiguration registers (one for each

endpoint)[7]. At this point the main interrupt service routine is registered for the hardware interrupt from the USB controller. One of the components in our design requires that we use the legacy interrupt API which did not work when using microC OS, as a consequence it is not used and all code is run in a single C main while(1) loop.

The layout and flow of the program is based off of an example project included on the Altera DE2 installation CD[8]. The 'Connect USB' and 'Handle USB Requests' sections of the software flow diagram are right at the beginning of the main while loop. Although they could easily be considered the same part, they are separated to emphasize that a connection is established once but each iteration continues to check for and handle unexpected USB requests.

The program flow is controlled primarily by interrupts which set the program state. When the ISP1362 chip fires its hardware interrupt, a main interrupt service routine is called; this routine reads from interrupt register in the controller chip to find out where the interrupt came from, then calls the interrupt service routine function related to the source. In our software there are three possibilities of where the interrupt came from, either it was from one of the IN or OUT control registers or from the IN interrupt register. The interrupt service routine branches set the state in a global variable, as the program iterates through the main loop it calls handler functions depending on the state. The state then unwinds to the next state depending on what was read from the packet being analyzed in the previous handler. Then the next appropriate handler is called in the next iteration. Each packet of a USB transaction with the host is handled in this process and it branches to various functions to handle tasks requested in the request.

There are too many different possible branches and requests to go over individually but there are three main categories of requests: Standard requests, class requests, and vendor requests. The standard requests are used to interrogate the device for various descriptors, set the configuration, or set the device's address on the USB bus. They are outlined in the USB specification and can be read about in more detail there. The class requests handled in this program are those required by the HID keyboard interface they are detailed in the HID 1.11 specification. Vendor requests are not applicable to this project.

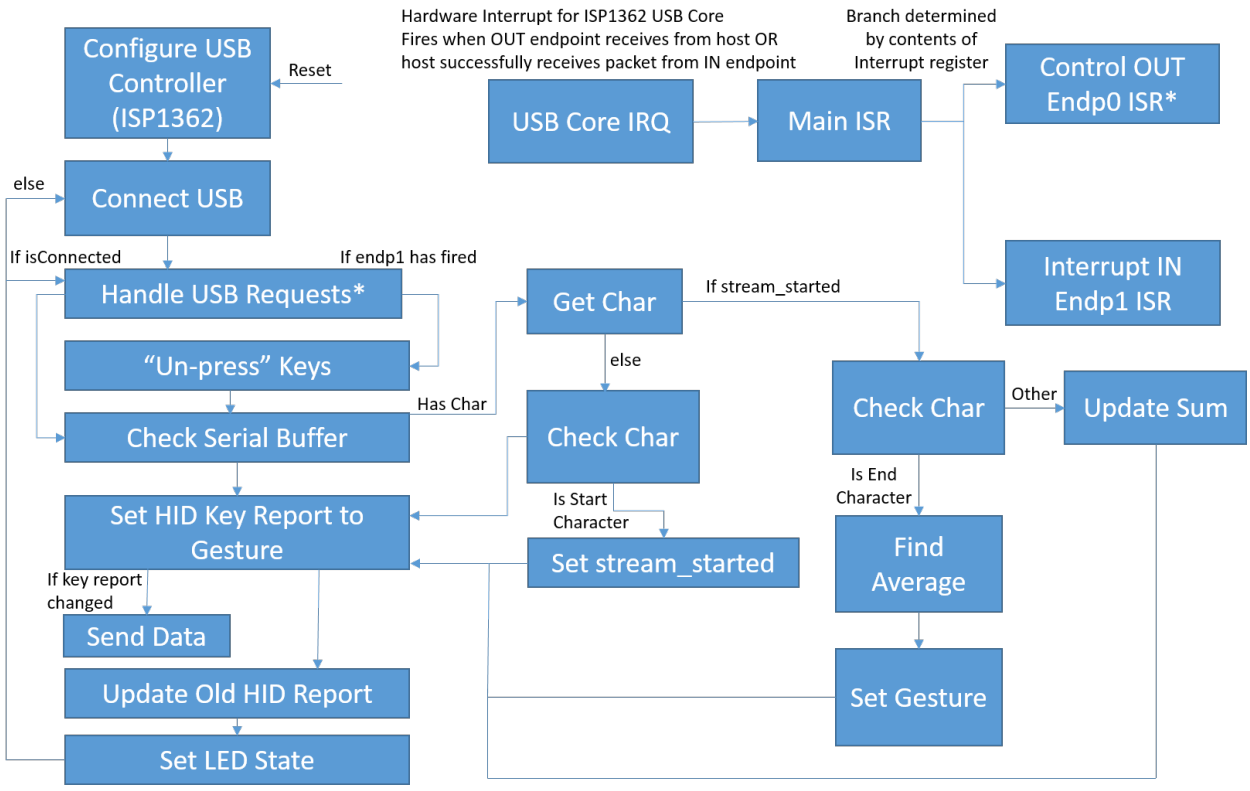
[7] For details on these registers or specific values written to them see the ISP1362 datasheet and project source code respectively in the appendices.
[8] From the Altera DE2 Installation CD under demonstrations

A data structure is stored that keeps track of the current set of keys and the current gesture. Every iteration of the main loop the keys are changed to match the current gesture. On most iterations of the main loop the gesture does not change and so the keys do not change either. Only two things change the current gesture: either the gesture recognition module completes analysis of a gesture (receives an end character), or the interrupt from the HID keyboard interface endpoint fires. If the state is changing due to recognising a gesture the current gesture is set to whatever action the user was interpreted to have taken was. The key set is changed in the report, and then, because the report is different from the last iteration it is written to the endpoint buffer to be sent. If the state is changing due to the interrupt firing, it means that a keyboard report was just received by the computer, so the gesture is changed back to the default state to send the key releases across in the same manner.

Each iteration of the loop the gesture recognition module moves forward one “tick”. Which is to say that it reads in one character from the serial buffer. Though this happens in the middle of the USB loop it can be condensed to a function that only returns a value every 50-100 iterations after user interaction and so it doesn’t interfere with the flow of the USB module.

At the end of the loop the old report is updated to the new report so that the comparison to see if the report has changed is accurate in the next iteration. Then if the most recent key press sent across was not a key-release, the key code is displayed by lighting up the green LEDs on the DE2.

Figure 7: Software Flow Diagram



Gesture Control Module

The glove module involves the ATMega328P polling the components (finger button and MPU-6050) on the glove and sending the received data to the transmitter. As specified in design, the ATmega328p will only send data when the index finger button is pressed down.All data transmitted from the microprocessor is converted into a print stream, therefore the data will be taken in from the DE2 as a buffer array of characters. When the button is pressed, The microprocessor will first take in one poll of converted acceleration data . This data is initially in a double format, but for the sake of processing intensity that comes alongside double value, all acceleration values are multiplied by 1000 and explicitly cast as floating point values. These initial X, Y and Z values will be used by the DE2 to determine what is the initial orientation of the glove. An integer value called count will be reset to a value of 0. This value will be used to limit the number of sent accelerations (the limit is set to the first 100 values). As long as the button is still held, and as long as the count is less than 100, The next set of accelerations are pulled from the MPU. The Y and Z acceleration values will be held in a double array until the count limit is reached or the button is depressed. Once one of these check are reached, the ATMega328P will printf() the character value ‘C’. This will be the character signal used by the DE2 to specify the start of data polling. The ATMega328P will now transmit all double data points within the array (in the form of floating points, multiplied by 1000). Each acceleration point is delayed by 5ms, in order to avoid any communication timing issues between the microprocessor and the XBee wireless transmission. When the array is empty, the character ‘E’ will be sent, which signifies the end of the gesture. The following code shows this process:

```
count = 0;
while(CHECK_BIT (i, 5) != 0){
	if(count < 100){
		mpu6050_getNextData(&daxg, &dayg, &dazg, &dgxds, &dgyds, &dgzds, ax, ay, az,
gx, gy, gz);
		accz[count] = dazg;
		accy[count] = dayg;
		count++;
	}
	i = PIND;
}
```

```
printf("C%d\n",count);
delay_ms(5);
printf("%d\n", initax);
delay_ms(5);
printf("%d\n", initay);
delay_ms(5);
printf("%d\n", initaz);
delay_ms(5);

for(iterator = 0; iterator < count ;iterator++){
        printf("%d\n", (int16_t) (accy[iterator]*1000));
        delay_ms(5);
        printf("%d\n", (int16_t) (accz[iterator]*1000));
}
printf("E");
```

To communicate with MPU the publicly available libraries mpu6050.c and i2cmaster.c were used.

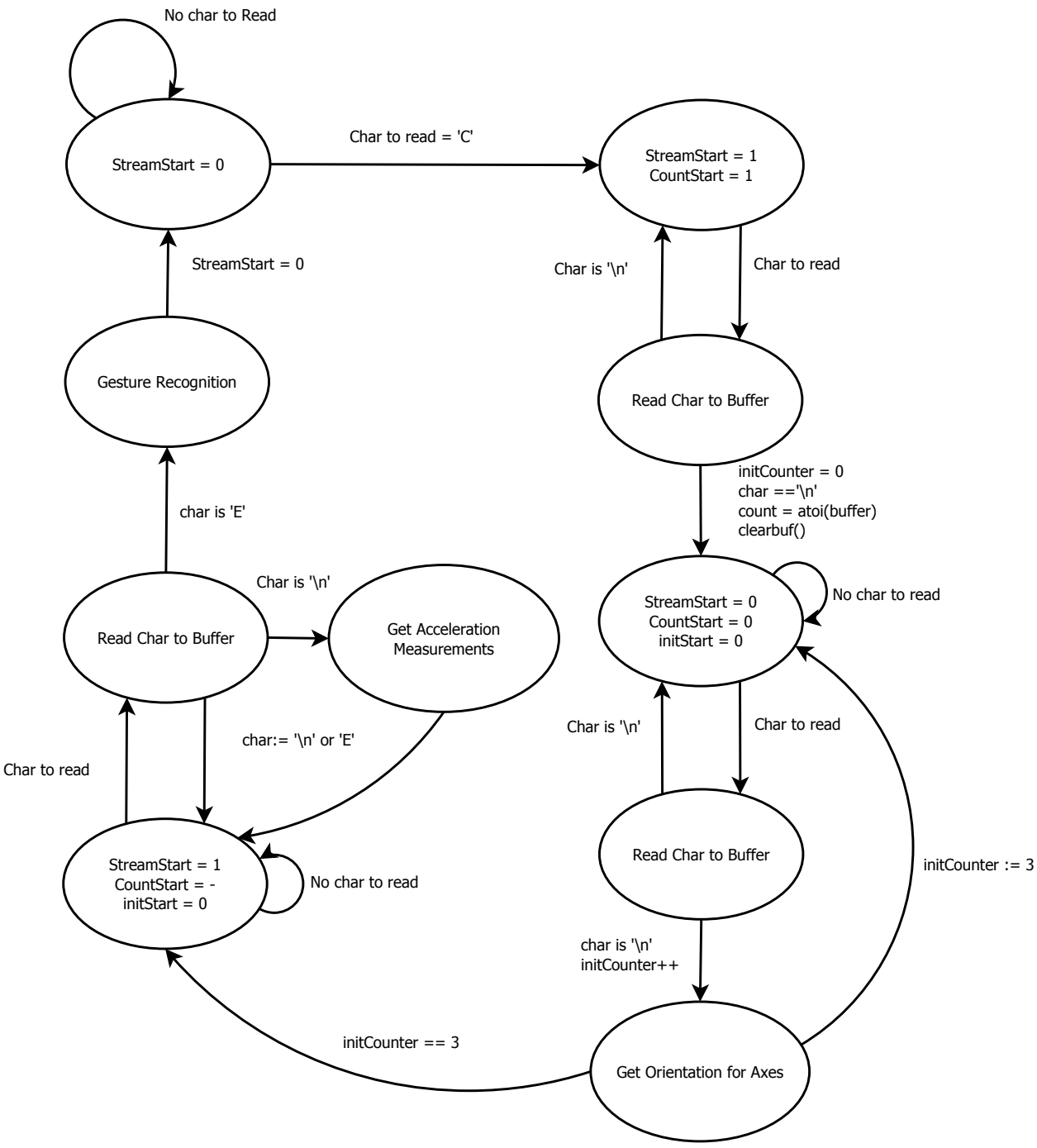
The gesture recognition module is where a gesture recognition algorithm will run in order to process the data received from the glove. This module is run on the DE2 and first must receive data from the glove through the Xbee receiver connected to the DB9 serial port. The Xbee receiver is mounted onto a development board which will handle all the communications specifics, but the DE2 is required to read data that the Xbee development board sends over the serial connection. To do this the university IP core for the serial port will be used. This IP core provides all the functions that are needed to read and store data from the serial port of the DE2 board. These are contained in the library `altera_up_avalon_rs232.h` and `altera_up_avalon_rs232_regs.h`. Using the `alt_up_rs232_dev` structure provided by the university core, a structure representing the serial port is created. The structure has a FIFO buffer that will hold all the data that is read from the serial port (so no data is lost when it is not read as it comes in). A provide function `alt_up_rs232_get_used_space_in_read_FIFO` is used to determine if there is data available to be read. If there is data available, another provided function called `alt_up_rs232_read_data` will be used to read a single character from the FIFO buffer. Note that even though the data needed is numeric, the serial connection sends characters that must be converted correctly. Each number can be multiple characters so a newline is sent after each number so that the DE2 can read all the digits of a number into a buffer and convert it into this

character buffer into a number with the function `atoi` when a newline is read. Below is example code using the university core libraries to read from the serial port.

```
alt_up_rs232_dev* rs232_dev;
char data_R8;
rs232_dev = alt_up_rs232_open_dev("/dev/rs232_0");
alt_up_rs232_enable_read_interrupt(rs232_dev);
while(1){
        read_FIFO_used = alt_up_rs232_get_used_space_in_read_FIFO(rs232_dev);
        if (read_FIFO_used > READ_FIFO_EMPTY){

                //Read a single character
                alt_up_rs232_read_data(rs232_dev, &data_R8, &p_error)
                /* Process the character
                …
                */
        }
}
```

There are various pieces of information that the gesture recognition algorithm needs in order to determine what gesture the glove executed. It needs the count of the number of measurements that are going to be sent, the orientation data for initial orientation, and all the acceleration data for the gesture sequence (in the y and z axes). Since what the DE2 reads is only a series of characters, the pseudo-protocol of characters sent by the microprocessor (seen above) will help the DE2 determine what information it reads. As it is a pretty lengthy process to get all the data below is a state diagram to help describe this process (following page).


No char to Read
StreamStart = 0
Char to read = 'C'
StreamStart = 1
CountStart = 1
StreamStart = 0
Char is '\n'
Char to read
Gesture Recognition
Read Char to Buffer
initCounter = 0
char =='\n'
count = atoi(buffer)
clearbuf()
char is 'E'
Char is '\n'
StreamStart = 0
CountStart = 0
initStart = 0
No char to read
Read Char to Buffer
Get Acceleration
Measurements
Char is '\n'
Char to read
char:= '\n' or 'E'
Char to read
StreamStart = 1
CountStart = -
initStart = 0
No char to read
Read Char to Buffer
initCounter := 3
char is '\n'
initCounter++
initCounter == 3
Get Orientation for Axes

The received character that starts off the stream is a ‘C’. The next number received (terminated with a newline) will be the total count of acceleration measurements that make up the gesture sequence. After that the next three measurements received will the initial acceleration measurements of x, y and z used to determine the initial orientation. The next numbers received will Y and Z accelerations for a certain point in time. These numbers will be stored so that they may be integrated to find the velocity data. Once the character’ E’ is read the DE2 knows it has all the data needed and may begin the gesture recognition.

The gesture recognition has been described extensively in the design so for the sake of brevity some features will be omitted here. As the data is being received and read, an integration approximation (figure 5) will be done on both the acceleration data from the y and z axis in order to get the velocity data for that time point. A running sum is of the velocity on a certain axis is also kept in order to be averaged out at the end. Below is code that represents reading a y acceleration and velocity data for a certain time point. Note that this step occurs after a newline is read for a y-axis acceleration number.

```
ay[counti] = atoi(buffer);
vNewy = vOldy + (ay[counti] );
VAveragey += vNewy;
vOldy = vNewy;
```

Once all the data has been read (an ‘E’ has been received) the program will average the running sum of velocities to get the average velocity in each axes. As described in the description, the orientation will be determined from the initial measurements. This is done with the orientation function shown below:

```
//function to get the orientations. 0 means palm towards the floor. 1 is palm towards the
left
//2 is palm towards the right. 3 is palm towards the roof
int orientation(int x, int y, int z){
	if(abs(z) >= abs(y) && abs(z) >= abs(x)){
		if(z>=0){
			return 0;
```

```
                } else{
                        return 3;
                }
        }else if(abs(y) >= abs(x)){
                if(y>=0){
                        return 1;
                } else{
                        return 2;
                }
        }else {
                return 0;
        }
}
```

This function works by finding which acceleration has the highest absolute magnitude. This will directly translate to a certain orientation if starting at rest. Once the orientation is determined the average velocities are passed into a function to determine which has the highest absolute magnitude to determine the direction the glove as traveled. Each orientation has a different function that will treat axes differently depending on motion translates to corresponding movement on the axes (as described in design) . Below is the function used for the palm left orientation.

```
//largrstPalmLeft is used to determine which direction the glove in
//the palm left orientation traveled based average velocity of each axes
//returned 0 means forward, 1 mean backwards,2 means left, 3 means right, 4 means up,
//5 means down
int largestPalmLeft(int x,int y,int z){
        if(abs(x) >= abs(y) && abs(x) >= abs(z)){
                if(x>=0){
                        return 0;
                } else{
                        return 1;
                }
        }else if(abs(y) >= abs(z)){
                if(y>=0){
```

```
                        return 4;
                } else{
                        return 5;
                }
        }else {
                if(z>=0){
                        return 3;
                } else{
                        return 2;
                }
        }
}
```

There also was a function to check if there was no movement (to account for noise and small movements as described in the description)

```
int noMovement(int x, int y, int z){
        if((abs(x) < 1500) && (abs(y) < 1500) && (abs(z) < 1500) ){
                return 1;
        }
        return 0;
}
```

Putting it all together the data read can be successfully analysed to determine what gesture the glove executed. The following code shows all the checks coming together to determine the gesture the acceleration data represents.

```
//VAverage is a running sum before the division
VAveragey = VAveragey/count;
VAveragez = VAveragez/count;
gloveOrientation = orientation(initax,initay,initaz);
switch (gloveOrientation){
        //case orientation is palmdown
        case 0:
                movementDirection = largest(0, VAveragey, VAveragez);
                Break;
```

```
        /*Other cases
        ...
        */
}
```

Note the main function can be found in main.c under DE2Component_Flash/Software/FinalFlashedDE2 in our github repository.

Test plan

Hardware testing consisted of:

1. AVR microprocessor connected to AVR Programmer (chip used to program the AVR in circuit. See application notes for more detail) .
2. XBee receiver connected to XBee development board. Board attached to DE2 through DB9 serial port. Wireless connection established for serial streaming.
3. Initial glove components connected with DC power supply. Breadboarded microprocessor, accelerometer, XBee transmitter and 3.3V voltage regulator providing proper voltage and power regulation. Testbenched with voltage probing.
4. Battery powered hookup of glove components. Breadboarded microprocessor, accelerometer, xBee transmitter, 3.3V voltage regulator and battery supply providing proper voltage and power regulation. Testbenched with voltage probing.
5. Components wired to glove. Voltage probing used to ensure proper voltage regulation.
6. USB connection of DE2 to Windows.

Software testing consisted of:

1. Microprocessor programmed by use of WINAVR through AVR programmer board. Microprocessor is properly programmed with a simple testbench program, which will blink LED's on and off if successful (shows that code is properly running, and internal clock is working).
2. USB connection of DE2 to Windows platform - recognized as a keyboard (HID compliant keyboard driver selected by sending correct packets). No necessary input is required; only that the DE2 is seen as a keyboard.
3. Wireless connection of AVR microprocessor to DE2 - recognized button movements. Can emulate scrolling of pages by movement of accelerometer on breadboard.
4. Memory leak checks to ensure freespace on DE2 board is not heavily used during operation.

5. Calibration of accelerometer. Determine minimum and maximum gesture recognition. Ensured sporadic movement is not interpreted by the driver as actual gesture.
6. Programmed a mock-data test on gesture recognition algorithm, to determine if algorithm could correctly identify velocity values as gestures..
7. Programmed gesture-to-key interpretation on DE2. Have Up/Down/Left/Right movement on glove perform PageUp/PageDown/Alt+Tab/Alt+Shift+Tab on PC.

Results of experiments and characterization

AVR microcontroller required programming through flashing code on the chip. A pocket programmer was used as a means to program the chip; the programmer was attached via wired ISP pins on the ATMega328p and to the PC via Mini-USB. WinAVR was used as the IDE to create C file to compile, and AVR-GCC was used to compile code into flashable HEX files. The files were flashed onto the AT via AVRDUDE. Initial tests concluded that the chip held a default 1 MHz internal clock. Button inputs were set as the “finger press” unit test. A series of LED’s were set to outputs, corresponding to button presses. Testing showed an instantaneous I/O reaction between buttons and LED’s.

XBee components were calibrated and programmed by using an XBIB-U-Development board attached to PC via USB (spoofed through driver use to act as COM port). XBee’s were set to the same PAN ID, and were provided MAC addresses to prevent illicit transmissions. Baud rate was set to 9600 bps, and XBee-to-XBee wireless network established as a one-way communication link.

XBee transmitter and 3.3V regulator attached to breadboard, and XBee receiver attached to XBIB-U-Development board still connected to PC via USB. Each breadboarded button was wired to one of four digitial I/O pins on transmission XBee. When powered on, and button pressed, an LED on the development board turned on instantaneously. This proved that the XBee network was communicating properly.

Accelerometer was soldered to header pins and attached to the breadboard, with proper voltage and ground pins, and 2 lines directing I2C lines (SDA, SCL) to AVR. When transmission (index)

button was pressed, probing showed a 3.28V high to low signal be passed through the SDA line.

X-CTU program was used to receive serial information from the receiving XBee to PC. Four button tests were set. The pinky button would send a constant stream of the characters “2\n” to ensure the communication of the XBees was not compromised by baud rate or clocking errors. The ring button was used to send “Test.” via a printf stream established on the AVR to ensure that it was working properly. The middle button was used to send the return case on the mpu6050_testConnection method provided by the mpu6050 library, and would output a “1” if the mpu was available on the I2C line. Finally, the index button was used to send in raw data received through the MPU. This data, in the form of raw acceleration and rotational velocity values, would ensure that a constant stream can be continuously sent. It was determined that 5ms delays would need to be applied before each value was passed, to ensure no incorrect serial information would be passed.

The accelerometer would send correct raw data wirelessly, but would contain incorrect gyroscopic values while stationary and flat on the board. Offsets were set in the mpu6050 header file, with GX set to -42, GY set to 9, and GZ set to -29. With the board flat, acceleration in the X and Y plane would output a very small number (close to 0), with Z displaying close to -1G (the acceleration of the chip due to gravity).

Raw data was recalculated into understandable units (G’s and Deg/s). Pressing down the index button. The 5V 2A power supply was replaced with the 3.7V LiPo battery wired to 500c power boost charger. Voltage probing showed an output of 5.12V. This output was then run through the 3.3V regulator. Output from the regulator was probed at 3.28V. Glove was ready to be constructed, with most major components wire-wrapped to a perfboard (wrist board) and accelerometer/buttons attached and wired to the glove.

Receiving XBee was attached to the DE2 through DB9 serial connector. “RXD” LED on the development board would light when finger was held down on glove, proving that wireless transmission from glove to board was successful.

Calibration of accelerometer was necessary to deduce actual movement to sporadic/passive movement of the user. Threshold values needed to be implemented to eliminate noise and

unnecessary movement, wherein velocities would be rejected before compared. Threshold values were set to 1.5 meters per second. Performing a series of glove gestures showed an accuracy rate of 95% (19 out of 20 gestures successful) in palm-down orientation, and 74% (69 out of 93) in palm-left orientation.

Largest delay from end of button press to gesture performed was just over half a second. This was due to the buffered data only sending from the glove to the board after the button is released, with a maximum data buffer of 100 accelerations. Each acceleration data point is separated by 5ms of delay time due to the 7% clock error of the AVR (which could cause discrepancies when sending data through to the XBee), which summates to 0.5s from glove to board. The moment the end character is detected, the DE2 would send the gesture out. In similar fashion, the shortest delay was just over 5ms (one acceleration point of data).

Testing for the USB connection occurred alongside debugging efforts. Each piece completed by monitoring tools in linux and windows. Error reports from windows device manager and dmesg in linux helped substantially. When the connection was actually established key codes written to the reports to be sent across were shown on the DE2 board in green LEDs which worked every time even if the keys were not successfully sent across to the computer. Another test run was to send over alternating characters to be printed to a word processor. The intention was to send keys that were easily visible and that could be alternated rapidly without needing user input to test. The results were not ideal and revealed a timing issue. Frequently (more than half) of the characters printed multiple on the screen before alternating to the other. The number of characters printed was also unpredictable, being between 1 and 5 additional. This was a problem for our purposes since it was important to only send a single command per gesture performed by the user. This problem was fixed by changing the keys pressed to the released state in the interrupt triggered by the computer successfully receiving the first key report.

Safety

This project involves multiple electronic components interacting with each other(in some cases in close proximity of a human) and as such safety is an important concern that must be taken into consideration.

Component Specifications

Component	Max Voltage	Operating Temperature	Power Consumption
AVR 28 Pin ATMega328P[9]	5.5V	-40C to 85C	Active mode: 0.2 mA Power save: 0.75µA
XBee-PRO® S1[10]	3.6V	-40C to 85C	Transmit: 250 mA Receive: 55 mA
MPU-6050[11]	3.465V	-40C to 85C	Gyro+ACC: 3.9mA
DE2 GPIO Pins	3.3V		

The electronic components specified above will all be contained on a glove the user will wear. In order to avoid overloading any of the components a voltage regulator will be used to maintain the input voltage of all components at 3.3V. The circuit on the glove will also have to be implemented so it is relatively free of static electricity. To achieve this the components on the glove may need to be slightly elevated above the glove on a breakout board. The glove will also be manufactured with the most static free material available. Materials such as wool will be avoided. The 3.7 V Li-polymer battery that will act as the power source for all the components will also be contained on the glove. This battery represents a fire hazard, especially due to the fact it is placed close to the user's body, and as such it must be implemented in a secure manner. The battery has an operating temperature of -20 to 60 degrees Celsius and a discharge current of 0.2 A. It is important to keep these factors in mind when using the battery in order to prevent it from causing a fire. If possible a small enclosure will be made that will provide a relatively static free and temperature safe environment for the battery in order for it to function as safely as possible. Capacitors will also be used in order to keep the voltage from the battery steady. In order to be assured of a properly functioning circuit, the glove component will be first implemented on a breadboard. Since there is wireless communications associated with the glove RF exposure is something that has to be taken into account. Due to the fact the XBee

[9] *8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash*, rev. 8025I, Atmel Corp.,San Jose, CA, 2009.
[10] *XBee®/XBee-PRO® RF Modules*, 1st edition, Digi International, Minnitonka, MN, 2009.
[11] *MPU-6050 Product Specification*, rev. 4.3, Invensense Inc., Sunnyvale, CA, 2013.

Pro's rate of transfer was relatively low it was determined that the RF exposure associated with the component was inconsequential.

Regulatory and Society

This project presents very few security threats as we deal with no private user information and does not use wifi for wireless communications (thus avoiding any IoT or remote vulnerabilities). The wireless standard used to communicate between the glove and the computer is ZigBee (an IEEE 802.15.4 standard). It operates on an unlicensed frequency band (in our case 2.4 GHZ) so there are no legal issues in Canada relating to our wireless communications.

A valid security concern is that someone in close proximity (around 100 feet) could send commands which the user did not request to the receiver, which is attached to the DE2 board and by extension the computer. The XBee wireless protocol has the transmitter and receiver programmed to a predetermined personal area network identification (or PAN ID). The receiver is also programmed to only allow incoming data to be processed based on the receiver's MAC address, which is passed along as part of the packet transmission header. Should someone find a way to send XBee information using a spoofed MAC, data could be falsely sent.

Environmental Impact

The glove component of the project is composed of various electronics that have varying degrees of environmental impact. The main electronic components of the glove (IC, MPU and voltage regulator) were required components that are all ROHS compliant. They will all have to be disposed after their functionality ceases but this was unavoidable. To dispose of the electronic components of the project a standard E-waste program should be used to minimize environmental impact. The material used for the finger contacts is copper, which is not one of the materials restrained under the ROHS directive. The battery used to power the glove component is the component that has the most environmental impact in our project. It is a Li-polymer battery (model number LP803860) that is ROHS compliant. It must be disposed of in a way compliant to the local regulations in order to have as little of an environmental impact as possible. The initial design of our project used 3 AA batteries to power the glove component however the wasteful nature of this method caused the power source to be switched to the Li-

polymer battery. This battery is rechargeable with a guarantee of over 300 charge cycles[12] and as such is much less wasteful than the previous method.

[12] Shenzhen PKCELL Battery Co., Ltd, “Li-Polymer Battery Technology Specification,” 803860 datasheet, Jun. 2004.

Sustainability

Power consumption:

IMU = Gyro+ACC: 3.9mA * 3.456 V = 0.0134784 W

ATMEGA328p = Active mode: 0.2 mA * 5.5 V = 1.1 mW

Power save: 0.75μA * 5.5 V = 4.125 μW

XBEE trans/rec = Transmit: 250 mA * 3.6 = 0.900 W

Receive: 55 mA *3.6 = 0.198 W

DE2 = Total Thermal Power Dissipation 126.96 mW

Power Estimation Confidence Low

(user provided insufficient toggle rate data)

This is an estimate; the actual total power consumption will likely be higher as it does not take into account data rate for USB component (uses default data rate which could be inaccurate). Nevertheless it provides insight into the power consumption of the DE2 board based on our FPGA design.

Device	**Max Power Consumption [W]**	**Duty Cycles**
MPU-6050 (IMU)	Gyro+ACC: 0.0135	100% while glove is on
ATMEGA328p (microprocessor)	Active mode: $1.1 * 10^{-3}$ Power save: $4.125 * 10^{-6}$	In active mode for 100% of the time while device on
XBee-Pro (wireless communication)	Transmit: 0.900 Receive/Idle: 0.198	Not transmitting while device is in ‘locked’ mode: Active (60%) * (Trans. + Rec.) + Idle (40%) * (Idle + Idle)

DE2 / FPGA	0.12696	Estimated using Quartus' PowerPlay Power Analyzer
Max Total	1.24076	
Average	0.95996	

Energy Cost:

As of February 1st in Edmonton regulated rates are 4.753¢ /kWh[13]

Annual Consumption:
(Average Power Consumption) * 7 days/week * 24 hr/day * 52 weeks/year = 8.386 kWh

Cost using February rate as average:
39.89 ¢/year

Carbon footprint:

(2.17 lbs. / kWh) / (2.4 lbs. / kg) = 0.90416 kg / kWh

8.386 kWh * 0.90416 kg / kWh = ***7.582 kg***

[13] Alberta Government. 2016 Rates (Online). Available: http://www.ucahelps.alberta.ca/historic-rates-2016.aspx

References

[1] *XBee®/XBee-PRO® RF Modules*, 1st edition, Digi International, Minnitonka, MN, 2009.

[2] *MPU-6050 Product Specification*, rev. 4.3, Invensense Inc., Sunnyvale, CA, 2013.

[3] *8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash*, rev. 8025I, Atmel Corp.,San Jose, CA, 2009.

[4] Linx Technologies. (2011, Aug 8) *Considerations for Sending Data over a Wireless Link* [Online]. Available: http://www.digikey.ca/en/articles/techzone/2011/aug/considerations-for-sending-data-over-a-wireless-link

[5] M.K.Schroder (2014, November 26). AVR Ultimate Driver Pack [Online], Available: https://github.com/mkschreder/avr-ultimate-driver-pack

[6] USB Implementers' Forum. (2001, June 27) *Device Class Definition for Human Interface Devices (HID)* [Online]. Available: http://www.usb.org/developers/hidpage/HID1_11.pdf

[7] Terasic DE2 CD-ROM, rev. 2.0.4

[8] Altera (2015, October 14). University Program IP Cores [Online], Available: https://www.altera.com/support/training/university/materials-ip-cores.html

[9] X. Calvet (2006, April 26). Linux: A Brief Tutorial [Online]. Available: http://www.freesoftwaremagazine.com/articles/drivers_linux

[10] *Linux Device Drivers*, 3rd ed, O'Reilly Media, Inc., Sebastopol, CA, 2005

[11] N. Seidle, (2008, June 19). Beginning Embedded Electronics [Online]. Available:https://www.sparkfun.com/tutorials/category/1

[12] Shenzhen PKCELL Battery Co., Ltd, "Li-Polymer Battery Technology Specification," 803860 datasheet, Jun. 2004.

[13] Digi (2012, November 26), xbee_ansic_library[Online], Available: https://github.com/digidotcom/xbee_ansic_library

[14] Alberta Government. 2016 Rates (Online). Available: http://www.ucahelps.alberta.ca/historic-rates-2016.aspx

Appendices

Quick Start Manual

Program the FPGA with DE2Component.sof. The SOF and POF files are provided in the Github repository located under References

The software files are found in the repository at DE2Component_Flash/Software/FinalFlashedDE2. These are the files that must be programed to the board. The FinalFlashedDE2_bsp must also be used because a generated bsp will not work correctly without a little tweaking.

Connect the DE2 and XBIB-R-DEV Development board to wall-socket power. Connect a serial null modem DB9 line to the XBIB-R-DEV Development board. The other end of the line needs to be connected to a DB9 gender changer. The gender changer can now be attached to the DB9 port on the DE2. Connect a type B USB wire to the “Device” port on the DE2. The other end of the cable (assumed to be type A) is then connected to one of the Windows or Linux computer’s USB port. MyKeyboardB driver may be displayed as being installed. This is the provided HID name of the DE2.

The “Good” LED will flicker and light when a USB connection has been established. In the event that it appears as though no more USB signals are being sent to the PC, press the KEY0 button (most bottom-right button on the DE2). This is the USB restart button, and will attempt to reconnect the USB device. This process takes between 3 and 10 seconds.
Wrist board and glove are still constructed. Connect a 3.7V LiPo Battery to the 5V power boost charger. Turn the glove on by the toggle switch attached to the wrist board; a blue LED should turn on on the power boost charger.

Future Work

Future features which could have been added with time permitting:

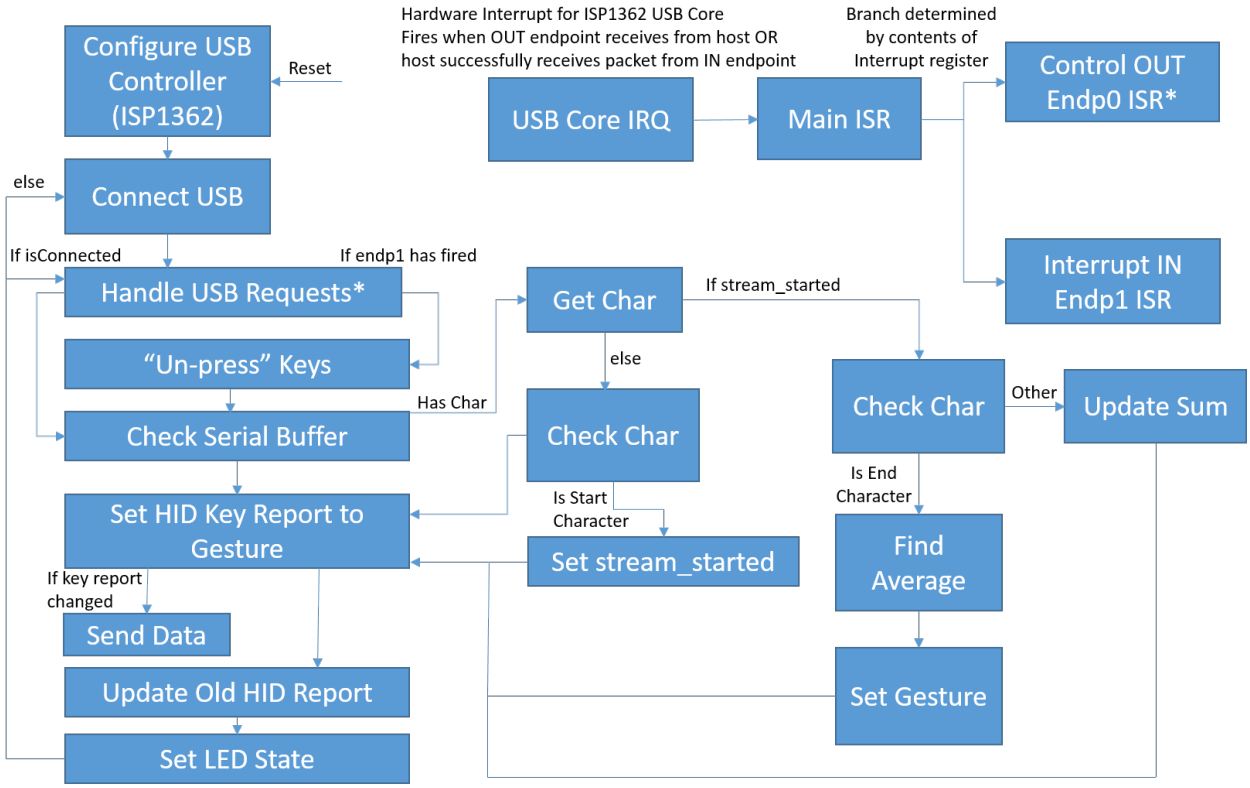
- Features for interacting with videos (play/pause, ffw/rw, volume)
- Mouse-like interaction (a pointer or other generic selection mode)
- Zoom in / out
- Personalized shortcuts

A backup plan was also implemented in the event of scope failure or time constraints:

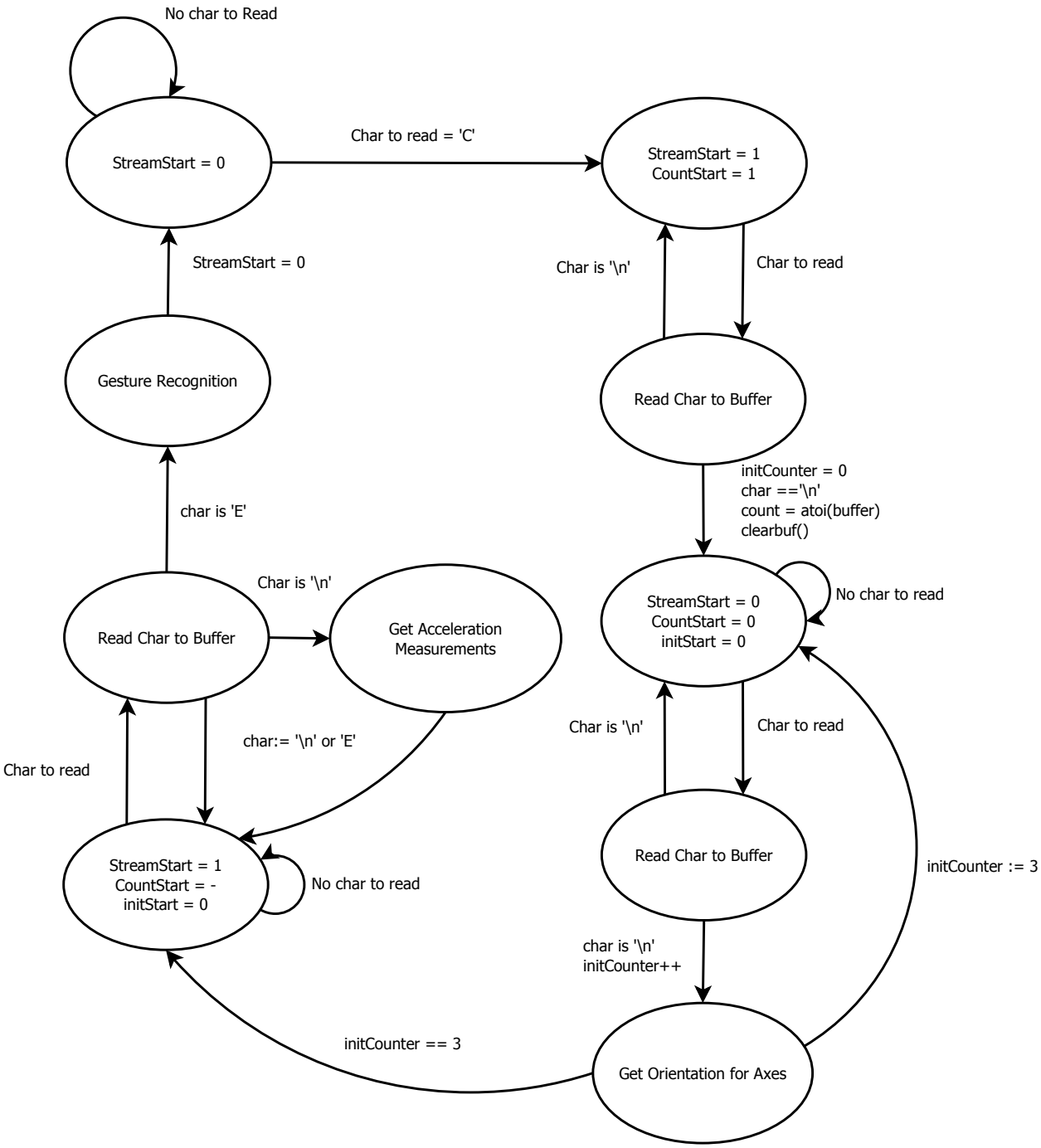
- The primary objective of the project was to achieve reliability in the basics. If multiple features were cut due to time constraints, the remaining base features would be the ability to switch windows and tabs.
- If recognizing the DE2 as a HID compliant device over USB was not achieved then a custom USB driver (in the Linux environment) would have been implemented instead. This driver would only accept simple data from the DE2 board (e.g. numbers) that would correspond to actions to perform. These actions (e.g. keyboard press or mouse wheel movement) would be simulated through a program running on the computer.
- If glove-to-DE2 communication was being recognized, but the DE2 failed to correctly communicate as a USB device over to the operating system, a backup would have been set up to display the action performed by the glove onto the DE2's LCD display panel and LED lights.

Hardware Documentation

Software flow diagram



Serial Transmission and Gesture Recognition State Diagram



Source code files can be found at

https://github.com/smithe0/GestureControlInterface

Video Project Overview

Video of the project is available at https://youtu.be/HFpMwudaUh8