Team Terminator Arm 2015/16 Final Report

Ayotunde A. Odejayi

Mark Chase

April 20, 2016

Executive Summary

The Terminator Arm project for designing, building and controlling a prosthetic arm was suggested to the department of Electrical and Computer Engineering at Howard University in 2015/2016 session by Mr. Ayotunde Odejayi. Upon approval of this request by Dr. Charles Kim, a student-based team was setup to work on this project.

The project was born to start a vertically integrated project that would seek to provide a prosthetic arm with comparable functionality to advanced ones that would go for a fraction of the price. It had been observed that although there exists significantly advanced prosthetics, these very prosthetics are virtually inaccessible to the people that need them the most: Amputees. Bridging this gap is the long term goal of project Terminator Arm.

Over the course of the 2015/16 academic session working in a cross-disciplinary team, we were able to progress to a prototype for a base working implementation to be improved over the years. Much of the work is also feature online on video hosting services like youtube and information sites including the course website at Howard University and the Cornell University Engineerinng Collabspace.

The final goal of this project is very attainable as for one, it has been verified that the cost criteria can be conveniently met. The high-functionality intended to be brought to the arm is also within reach based off the progress made in this years' experience alone. Over the next couple years, we expect the department at Howard University to have gained a foothold on this project and progress to supporting a major research for the advancement of affordable prosthetics for those who need them the most.

Table of Contents

Introduction and Background --- 4

Problem Statement-- 5

Current Status of Art-- 5

Design Requirements--- 6

Solution Approaches and Top Design--- 7

Project's Final Goal---13

Project's Spring 2016 Target Goal--- 14

Implementation, Testing and Evaluation-- 15

Conclusions--- 19

Recommendation for Future work-- 21

References --- 22

Source Code Listing-- 23

Introduction and Background

This project is primarily concerned with improving the quality of life for amputees worldwide. This includes amputees resulting from varying conditions including military warfare victims, disease inflicted victims including diabetes and of course those who were born amputated. The objective of the project is to design a relatively inexpensive, non-invasive prosthetic arm controlled by electrical pulses from the brain. In being relatively inexpensive, we intend to provide a functional prosthetic arm on the order of a few thousand dollars in comparison to the more expensive prosthetics which go for tens of thousands of dollars. The arm is also designed to be non-invasive as it requires no intrusion into the

Motivation:

As can be inferred, at the heart of this project is Amputees, particularly those without finances to support their situation and persons born amputated from the elbow. We hope to provide cheap assistance to amputees

Cost:

Traditional myoelectric prosthetics cost upwards of $3000. There exists state of the art prosthetics by Robotica that go for $8000, Terminator Arm would be assembled for less than one-tenth this price.

Problem Statement

Our problem statement verbatim is “Amputees deserve a normal life like everyone else. They require an inexpensive hand replacement that provides functionality comparable to the human hand. This should be light, comfortable and reliable for everyday use.”

We intend to produce a prosthetic arm utilizing electromyographic methods. Electromyography is the study of electrical activity in the muscles. We intend to use some modern technology including electromyography and 3D printing.

Current Status of Art

Traditional myoelectric prosthetics cost upwards of $3000, ours is to be assembled for one-tenth this price.

e-NABLE, described as a global network of passionate volunteers using 3D printing to give the world a helping hand has open sourced design for hand-prosthetics that are purely mechanical.

It is made up of a pair of hooks on the prosthetic arm held together by a rubber band. It also has tension cords with extensions that hang over the shoulder of the amputee. The “fingers” are either of a pair of claws or the regular 5 fingers. It is designed for amputees from the shoulder and to be controlled by mechanical shoulder bone motion. Hence, this goes for little to nothing as it is mostly given out free.

Advanced prosthetics made by major research groups at universities and companies including engineers and researchers at Johns Hopkins University. At Johns Hopkins, a team of neuroscientists, engineers

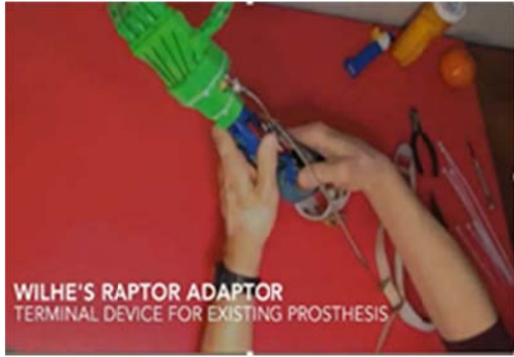


Fig. Purely mechanical and more advanced prosthetics

Design Requirements

First, as Intel Cornell Cup 2016 participants, we are required to use the DE2I-150 development board as a part of our design. In addition, we hope to produce a prosthetic that meets the following broad criteria. Physically, we intend for the arm to be as durable/strong as possible, such that it can withstand minor impact and still be compact. We also intend to produce a prosthetic that closely mirrors the human arm and hence has similar motion. In addition to wiring the individual fingers to be just taut, We decided to allow for a 60 degree rotation at the wrist of the arm. Something which is not available in several 3D prosthetic arms.

We also intend to produce an electrically stimulated functional prosthetic arm. As most 3D printed arms are purely mechanical, we seek to bridge the gap between the lower-end prosthetics which go for little to nothing and the more expensive and highly functional prosthetic arms. We are using the Myo Armband by Thalmic Labs as our sensor/listener for the purpose of electrical stimulation.

Although, not incorporated yet, we seek to provide a means of security/protection for the prosthetic arm and thereby extend the lifetime of the prosthetic arm significantly. The method we propose to protect the servo motors inside of the arm from severe external heat for instance.

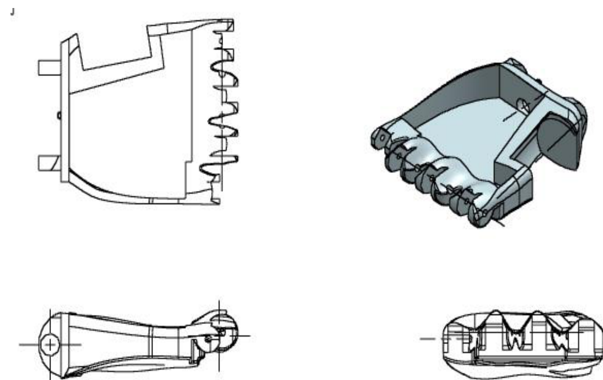
Solution Approaches and Top Design

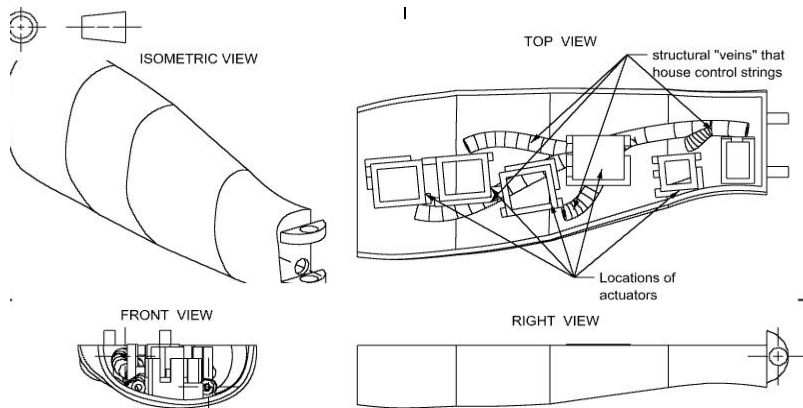
As far as the physical arm goes, our design intends to use veins channel cables from servos to finger joints. We chose to to have the servos housed in 3D printed pockets inside of the arm. We inevitably progressed from the initial idea where all servos were to be situated below the microcontroller inserted into the forearm to an external computing platform with the Intel Cornell cup.

Our Solution Approaches is split into 3 categories:

A) Physical Arm design Concepts

Design 1: Incorporates human-like wrist motion. Hinged wrist design allows 60 degrees of freedom





Assembly of forearm and hand without finger joints shown right

CAD Model is the right arm of an adult male. The forearm is also designed to house varying sized

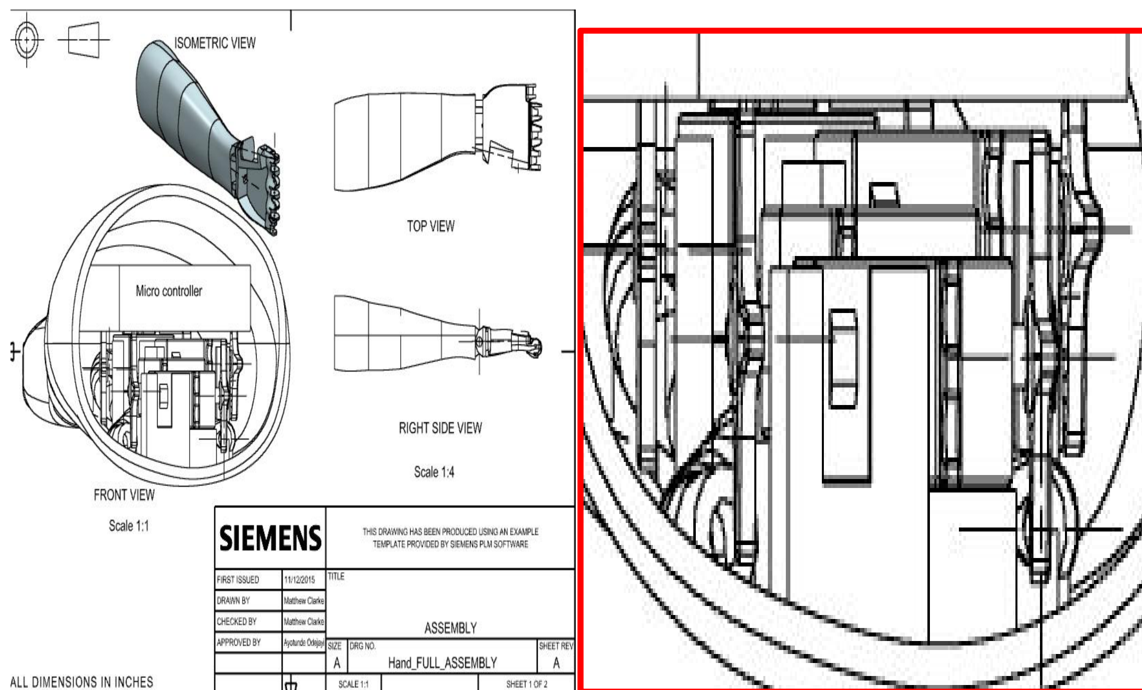
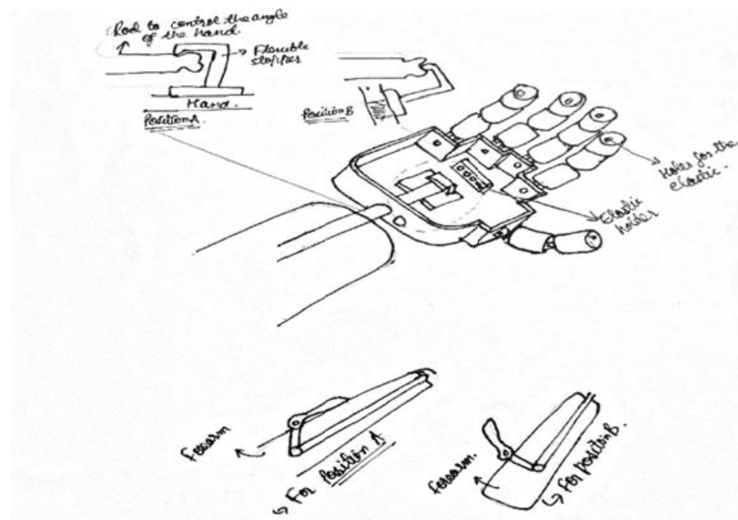


Fig. Fabrication & Assembly of 3D Printed Arm

Design 2:

This was to make use of much less agile material in metal and have a specific housing for a microcontroller in the palm of the Terminator Arm



B) Gesture Recognition concepts:

At the heart of this project, is being able to correctly and quickly identify whatever the user is doing at any moment with their arms so we can replicate it with the prosthetic arm. Here, we consider detecting gestures based on the data provide by the Myo armband from Thalmic labs pictured below. The data received from this sensor/listener is very hard to distinguish across a variety of persons hence gesture recognition is highly important.

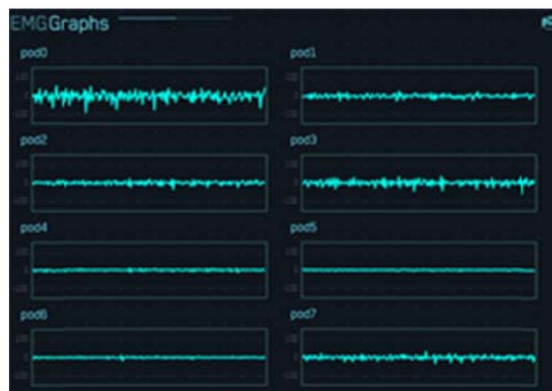


Fig. Myo armband and the data it provides

The conceptual designs for achieving this include:

1. **Gradient descent approach**:

This was going to include developing a multivariable equation and compute a solution. We thought trying to compute the line of best fit for the data set might give an insight into how the gestures could be accurately recognized.

2. **Characteristics approach**

   This method involves generating frequency plots based off the data and hardcoding expected percentage values for the separate gestures. This is a rather crude method of real-time data analysis.

	rest	thumb	index	middle	ring	pinky	wrist
pod 0	82.67504	52.77045	50.32498	52.55339	47.64151	39.86805	68.10345
pod 1	97.21724	83.64116	81.70845	86.16527	77.92453	76.24882	87.83525
pod 2	99.91023	93.66755	95.45032	95.72888	94.33962	93.1197	90.90038
pod 3	99.91023	92.70009	96.19313	92.85051	92.26415	92.83695	71.83908
pod 4	99.28187	91.38083	87.65088	80.87279	63.01887	83.03487	46.7433
pod 5	97.307	91.55673	76.97307	79.29434	49.43396	77.19133	52.20307
pod 6	98.9228	93.22779	85.23677	85.51532	79.62264	76.53157	61.2069
pod 7	94.52424	79.50748	74.65181	70.00929	65.9434	68.42601	64.94253

   Fig. Plot of data generated from frequency analysis

3. **Machine learning (Fast Artificial Neural Network, FANN)**

The method we almost defaulted to is Machine learning neural networks. After securing the source code behind the FANN, we explored training and testing this network of several layers of neurons with our data

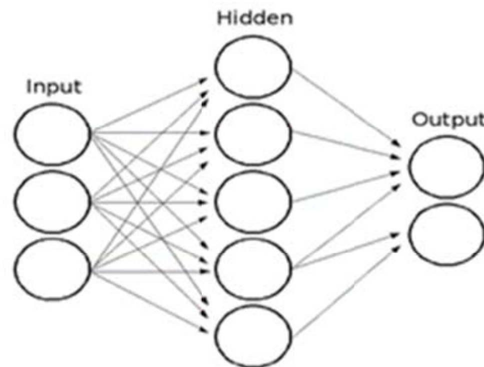


Fig. Neural network with 3 layers

C) Bluetooth low energy, BLE/serial communication implementation

Because the integrated graphics card that ships with the DE2I-150 board does not meet the requirements for the MyoConnect software (software provided by Thalmic labs to connect myo to pc/board). We had to think of a work around for this roadblock. The 2 concept designs were

1) **Serial communication**:

The first method was to create a 2-layer communication between the myo armband and the application code. The first step was to use an API (Application programmable interface) provided by Bluegiga, the company that manufactures the Bluetooth chip inside of the Myo armband to connect the myo to a physical COM port. Then, proceed to accessing data from the serial port (established as a virtual serial port because of Windows OS) for the application

2) External Graphics card

We also considered using an external graphics card (GPU) to run the MyoConnect software that connects the Myo armband to a computer as provided by Thalmic labs

Fig. student extracting graphics card from old desktop machine

Design Matrix													
	Weight		Unit score	Aggregate		Unit score	Aggregate		Unit score	Aggregate		Unit score	Aggregate
Ease	5	FANN	2	10	Char	3.5	17.5	GPU	3.5	17.5	BLE	2	10
Efficiency	4		2	8		5	20		3	12		4	16
Competition Constraints	4		5	20		5	20		2	8		5	20
Space overhead	3		5	15		5	15		2	6		5	5
Design Preference	5		2	10		1.5	7.5		2	10		5	25
Total				**59**			**77**			**43.5**			**73.5**

Fig. Top design selection

The final design consisted of physical design 1 for the hand design, Characteristics approach for gesture recognition and Serial communication for BLE communication

Project's Final Goal and Deliverable

This project's final goal is to create a perfectly functioning prosthetic arm. The arm will be used by amputees to aid them in their everyday lives. As such, this arm must be able to mimic any gesture displayed by the user with a significantly low response time. It must also be durable and have a sufficient battery life to support everyday use. With the ability to customize the physical arm to the specific user's needs, the functionality needs to be robust and work with a wide variety of users. Lastly, the arm needs to be amenable to daily use which means that it must be completely portable.

We also seek to implement this entire system with a significantly low response time. This issue is a little complex and proves difficult to solve as we have several integral components to our system which all increase the overall latency of the system.

For practical application purposes, it is important for the prosthetic to be durable and provide sufficient battery life. It would be near disastrous to produce an arm that requires new batteries or battery charging every now and then. For instance, if a user was driving or doing some other delicate task and experienced a dead battery.

It is also desirable to produce a robust system that works for a wide variety of users irrespective of the width of a person's forearm or the amount of hair on the surface of the skin or even the amount of fat beneath the skin surface. We want to produce an arm that works generally across a large variety of users.

Finally, we desire for terminator arm to be amenable to daily use. In this sense, we want it to be highly portable and lightweight for the best user experience.

These serve as the basis for what we intend to achieve with the Terminator Arm in the future. Further design specifics may be introduced in the future.

Project's Spring 2016 Target Goal

As far as Spring 2016 is concerned, the features we set out to incorporate into the Terminator Arm are as follows:

We intend to produce a prosthetic arm that can respond accurately to a select number of input gestures. These are 6 in number including a rest state, thumb finger contraction/relaxation, index finger contraction/relaxation, middle finger contraction/relaxation, ring finger contraction/relaxation, index finger contraction/relaxation and wrist finger/relaxation.

As in the images below, we intend for the arm to replicate these gestures remotely with the armband close to the user's elbow.

Fig. Assembling the Terminator Arm

We also desire to limit the lag time to the order of seconds for now (A 1.5 second response time is an ideal start). We are focused on Implementing the arm functionality centered around the Intel platform in the spring semester.

Implementation, Testing and Evaluation

The major tasks in completing the project were split in the following manner:

Physical Hand Design (Matthew Clarke, Bibek Ramdam), Software development (Ayotunde Odejayi), Electrical and Computer hardware (Ayotunde Odejayi), Bluetooth Low Energy, BLE (Mark Chase, Cory Bethrant), External GPU (Ayotunde Odejayi, Mark Chase), Data Science (Ezana Dawit), Project Management/Admin (Bibek Ramdam).

We progressed from theory to implementation with the production of the 3D printed assembled arm. In the photo below, three mechanical engineering students namely Taylor Hines, Matthew Clarke and Bibek Ramdam.

Fig. Assembling the Terminator Arm

The very next step after printing and assembly was to prototype the servo motors inside of the forearm with FPGA controls. We were unable to easily and quickly implement PWM for controlling the motors through the FPGA GPIO so we opted to introduce a microcontroller between the DE2I-150 board and the physical arm for easier control of the arm.

Stage 1: 3D printing

FDM process using ABS plastic which involves the removal of support material from model material using acid bath

Why 3D printing?

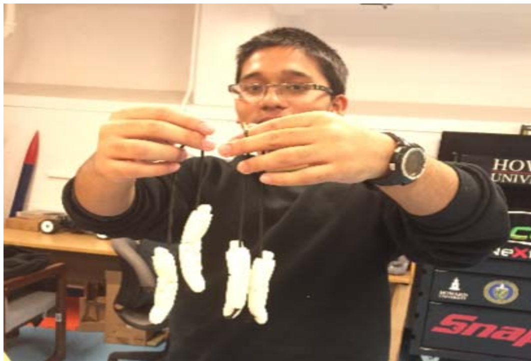
3D printing was selected because it offers short time-to-market and its easily customizable

We also realized that 3D printing enables hardware open-sourcing of our terminator arm. The specific strength of ABS for this concept design is 0.073 and the specific strength of the stainless steel is approximately 0.07.

Stage 2: Assembly of arm, hand and finger joints

Assembly took upwards of 2 days and 8 hours

Utilized additional material purchased



Data Collection:

To make sense of data and establish working concepts across all users, we had to collect data from test subjects. We were able to gather data from 31 subjects over the course of the spring semester. The Myo armband has 8 pods that provided 8 EMG data readings around forearm for each user for 3 seconds with each gesture. The data was determined by location and intensity of contraction.

Data Analysis:

In the process of data analysis, we used the characteristics approach to compare pod values and pod relationships to predetermined trends from frequency plots.

We then proceeded with designing the Terminator arm by prototyping with FPGA controls for servo motors

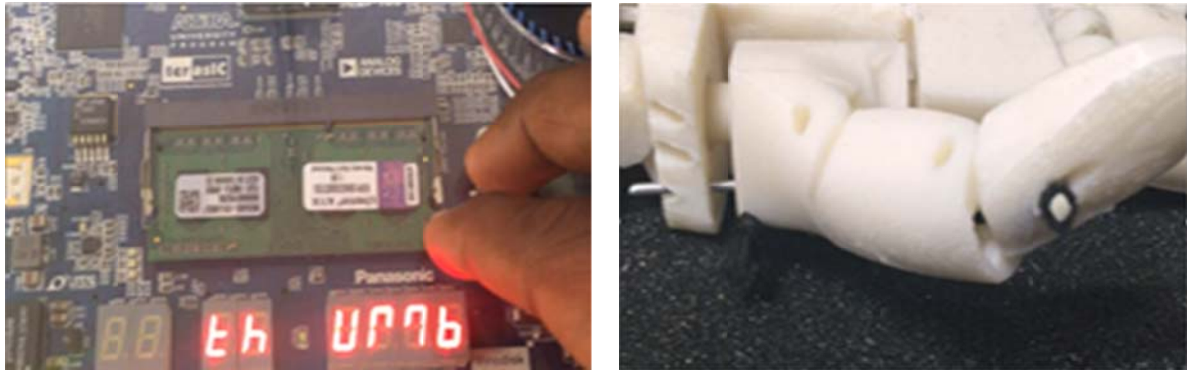


Fig. Prototyping the motors with FPGA through microcontroller

Implemented PCIe bus communication using a BYTE (eg. 0100 0000) in one-hot mode to encode data for all gestures.

In completing the arm at this pointj, our four point approach was:

Connect MYO with Intel Atom

Translate MYO data into accurate gesture (Classifier)

Atom/FPGA connection through PCIe bus

Interface FPGA with Intel Galileo for servo motor control and other I/O

We then proceeded to set up the development board by installing Windows OS and Visual Studio IDE on Intel Atom.

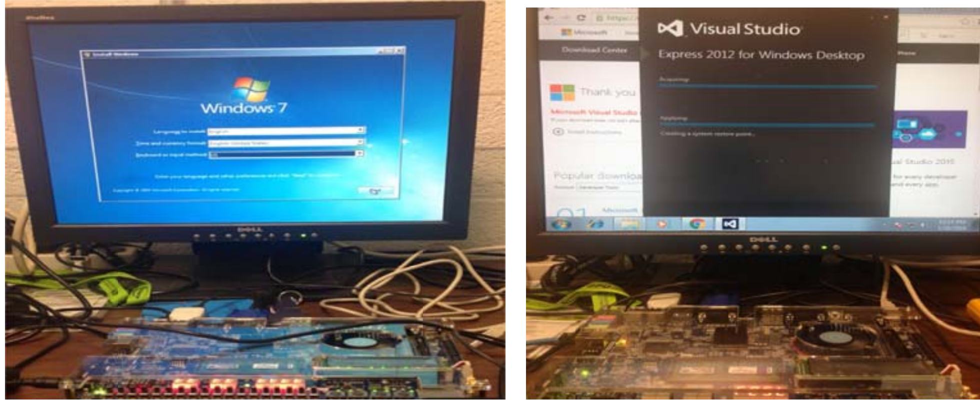


Fig. Installing Windows OS and Microsoft Visual studio on atom

After this, we were able to establish a basic communication between the Atom and FPGA

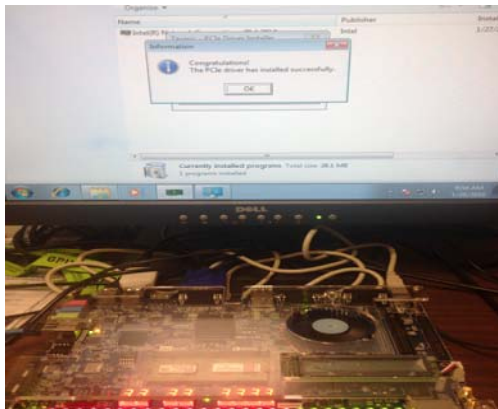
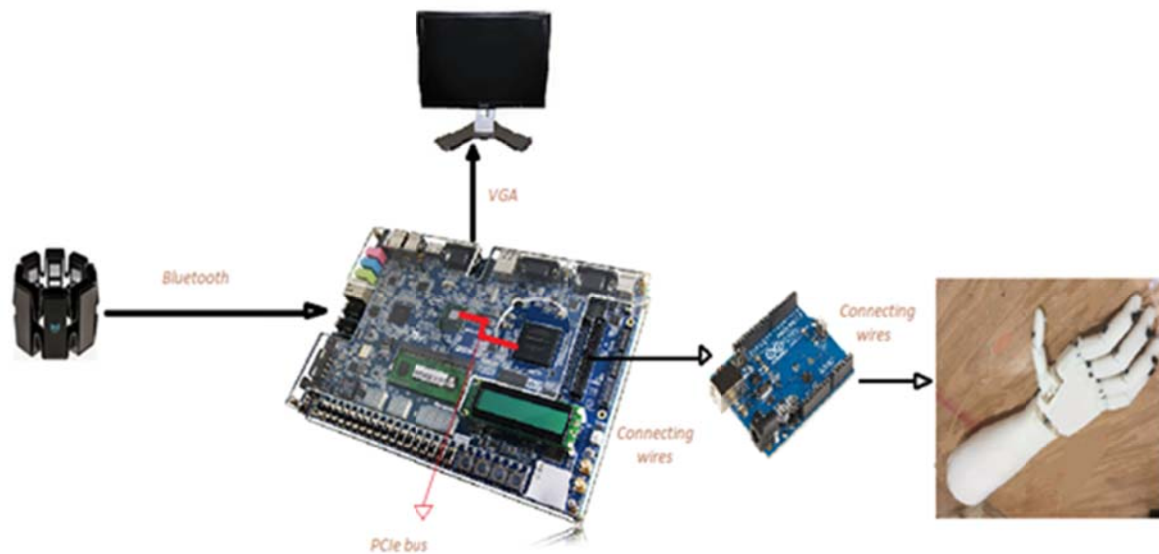


Fig. Interfacing FPGA with Intel Galileo for better servo motor control

Design Architecture

The final design architecture for the project is as shown:



Conclusion

As intended, this project served as a great engineering design and learning experience for the students involved. The budget for the resources used came just under $320 with the Myo armband at $199 being the biggest expense. We also learned about the common pitfalls of cross-departmental, dynamic teams. It proved important to set important team goals and always update everyone on the progress made by one unit as the nature of the project demands interdependence. For instance, we were unable to proceed with major portions of the electrical/computer hardware of the system until the mechanical engineering students building the physical arm were finished.

We also learned firsthand about unforeseen design problems/constraints which are a common part of the engineering design process. The base of the project would have been completed quite early with continuous improvement to gesture recognition, the only concern if the required development machine, the Terasic DE2I-150 board had better (any modern computer level) integrated graphics. We stumbled into a major roadblock with the Bluetooth communication implementation because of this.

We produced as final results a basic, working implementation of the 2 sides of the Terminator Arm. Namely, gesture recognition with the Myo amband and CPU control of the servo motors in the arm through the PCI express bus communication. One the gesture recognition side, we were able to recognize 3 of 6 hand gestures reliably including rest, ring and wrist gestures while on the Arm control side, we were able to control the individual gestures (although not perfectly reliably) for the 6 gestures.

We also learned to be very agile in the engineering design process. Our entire system crashed in less than 48 hours before our final presentation/demo while implementing external GPU. However, we were able to get back up and running before the presentation.

Recommendation for Future work

As the project is handed on to future design groups, the areas that stand out and could use some improvement include gesture recognition, diminishing lag time/latency, creating portability, incorporating new features, increasing battery life/durability. As can be seen, there is more than ample room for improvement and further work on the Terminator Arm. The method of gesture recognition used Characteristics classifiers needs continuous improvement to come to a competitive standard. Furthermore, a really advanced form of prosthetic would be practically impossible without the use of machine learning further down the road for all gesture recognition. It remains to be seen why the FANN, fast artificial neural network library implemented in this project did not produce telling results. Future teams are therefore encouraged to actively explore machine learning techniques in making major progress in gesture recognition for the Terminator Arm.

Optimal power means is also an issue of concern. Supplying power to Servos and Arduino via batteries.

Ensuring the wiring of the Terminator Arm servos is taut and perfect would also be a good idea. Servo handles need adjustment. As the terminator arm progresses towards a marketable product, completing the arm packaging would be a necessity. The arm would also probably improve in response time if latency is reduced.

References

i) dongkikim.com/project/data/FINAL ECE Spring Documentation v6.pdf

ii) http://www.mwftr.com/SoCs14/DE2i-150_FPGA_System_manual.pdf

iii) http://developerblog.myo.com/myo-bluetooth-spec-released/

iv) https://developer.thalmic.com/forums/topic/2903/

v) https://www.safaribooksonline.com/library/view/getting-started-with/9781491900550/ch04.html

vi) https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt

Source Code Listing

The major blocks of code (the .cpp files application) are included. Source is also available on github at [TerminatorArm-Project Github]

1. **GatherData.cpp (Used in gathering data from subjects)**

GatherData.cpp | Infile.txt
emg-data-sample-VisualStudio2013 | (Global Scope)

```cpp
// Modified Version of emg-sample-data.cpp (Copyright (C) 2013-2014 Thalmic Labs Inc. Distributed under the Myo SDK license agreement, LICENSE.txt)
// By:
// Howard University Terminator Arm Senior Design 2015.
//

// Dependencies:
// 1. MyoConnect
// 2. Infile.txt (Containing dynamic final part of output data file)

// Description:
// This is used solely for gathering our EMG classifier data. The rest of the code for the project is separate from this.
// EMG streaming is only supported for one Myo at a time
// Terminator Myo MAC address: db-fa-0c-69-14-78 (Can be used alternatively instead of MyoConnect to attach our Myo to a Hub)
// http://diagnostics.myo.com/ provides diagnostic data of Myo (connected in MyoConnect) on a pc
// NB: Replacing system("cls") is recommended and advised.


// #define _USE_MATH_DEFINES
#include <cmath>
#include <iomanip>
#include <algorithm>
#include <array>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include <fstream>
#include "windows.h"    // contains Sleep library
#include <time.h>
```

100 %

```cpp
#include <myo/myo.hpp>            // Myo default header library

// DataCollector class inheriting member functions from myo::DeviceListener
class DataCollector : public myo::DeviceListener {
public:
    DataCollector() : emgSamples()
    {}

    // onUnpair() is called whenever the Myo is disconnected from Myo Connect by the user
    void onUnpair(myo::Myo* myo, uint64_t timestamp)
    {
        // We've lost our Myo.
        // Zeros are seen in output file if there was a disconnect so test can be repeated
        emgSamples.fill(0);
    }

    // onEmgData() is called whenever a paired Myo has provided new EMG data, and EMG streaming is enabled
    // Classifier output data
    void onEmgData(myo::Myo* myo, uint64_t timestamp, const int8_t* emg)
    {
        for (int i = 0; i < 8; i++) {
            emgSamples[i] = emg[i];
        }
    }

    void stayUnlocked(myo::Myo* myo)
    {}

    // The values of this array is set by onEmgData() above
    std::array<int8_t, 8> emgSamples;
```

```
    // There are other virtual functions in DeviceListener that we could override here, like onAccelerometerData()
    // For this application, the functions overridden above are sufficient
    std::string SerialIndex, update = "gesture";
    std::ofstream TerminatorFile;
    std::string filepath = "C:\\Users\\Ayotunde\\Google Drive\\Team Terminator\\Data Analysis\\EMGClassifierData\\TeamTerminatorData";
    int NewLineflag = 0;

    // We define this function to write the current values that were updated by the on...() functions above
    void writeData(std::string gesture)
    {
        // Create and open dynamic outfile
        std::string filename = filepath + SerialIndex + ".txt";
        TerminatorFile.open(filename, std::ofstream::app);

        // Write current gesture to outfile
        if (update != gesture){
            TerminatorFile << "\n" << gesture << "\n"; update = gesture;
        }

        // Clear the current line
        std::cout << '\r';
        // write out the EMG classifier data
        for (size_t i = 0; i < emgSamples.size(); i++) {
            std::ostringstream oss;
            oss << static_cast<int>(emgSamples[i]);        // convert 8-bit array into int
            std::string emgString = oss.str();

            if (NewLineflag == 1){
                TerminatorFile << "\n";
                NewLineflag = 0;
```

```
            }

            // Write to outfile
            TerminatorFile << emgString << " ";
        }
        NewLineflag = 1;
        std::cout << std::flush;
        TerminatorFile.close();       // flush buffer to outfile
    }
};


int main()
{
    // Declare & open infile
    std::ifstream Serialfile;
    Serialfile.open("Infile.txt");

    // Define test gestures
    std::string gestures[12] = { "*REST/RELAX* position", "*(THUMB)* contraction", "*REST/RELAX* position", "*(INDEX fing.)* contraction", "*REST/RELAX* position",
        "*(MIDDLE fing.)* contraction", "*REST/RELAX* position", "*(RING fing.)* contraction", "*REST/RELAX* position",
        "*(PINKY fing.)* contraction", "*REST/RELAX* position", "*(HAND)* contraction" };

    // We catch any exceptions that might occur below -- see the catch statement for more details
    try {

        // First, we create a Hub without any application identifier (I deemed it unnecessary), the Hub provides access to one or more Myos
        myo::Hub hub("");

        std::cout << "Attempting to find HU Terminator Myo..." << std::endl;
```

```
// Next, we attempt to find a Myo to use. If a Myo is already paired in Myo Connect, this will return that Myo immediately
// waitForMyo() takes a timeout value in milliseconds. We try to find Terminator Myo for 10 seconds, and
// if that fails, the function will return a null pointer
myo::Myo* myo = hub.waitForMyo(0); //  Times-out until a Myo is found!

////////////////////////// To-do: CONNECT TERMINATOR MYO BY MAC ADDRESS AND KEEP UNLOCKED //////////////////////////////////////////////////////
hub.setLockingPolicy(hub.lockingPolicyNone);      // Keep Terminator Myo unlocked
//Hub::a               attachByMacAddress(String macAddress)
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// If waitForMyo() returned a null pointer, we failed to find our Myo, so exit with an error message
if (!myo) {
    throw std::runtime_error("Unable to find HU Terminator Myo! \nCheck MyoConnect!");
}

// We've found our Myo
std::cout << "Connected to HU Terminator Myo!" << std::endl << std::endl;

// Next we enable EMG streaming on the found Myo
myo->setStreamEmg(myo::Myo::streamEmgEnabled);

myo->unlockHold;

// Next we construct an instance of our DeviceListener, so that we can register it with the Hub
DataCollector collector;

// Grab index of subject for file-writing purposes
while (!Serialfile.eof())
    Serialfile >> collector.SerialIndex;
```

```
            // Get data for 3 seconds
            while (currentTime <= 3000)
            {
                // In each iteration of our main loop, we run the Myo event loop for a set number of milliseconds
                // In this case, we wish to update our display 50 times a second. (Myo provides EMG at 200Hz and IMU d
                hub.run(1);

                // After processing events, we call the writeData() function to write new data to our outfile
                collector.writeData(gestures[i]);

                // Update time for iteration purposes
                currentTime = GetTickCount() - startTime;
            }

            while ((GetTickCount() - startTime) <= 5000) {}; // wait for 1 extra sec for user change

        }

        // Tidy up & End program
        std::cout << "Saving Data... " << std::endl;
        system("cls");
        std::cout << "\n       *****  Thank you for helping out HU Team Terminator!  *****\n\n";
        std::cout << "                          From Ayo & Mark!           \n\n";
        Sleep(3000);
        return 0;

    }
    catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        std::cerr << "Press enter to continue.";
```

```
        // Hub::addListener() takes the address of any object whose class inherits from DeviceListener, and will cause
        // Hub::run() to send events to all registered device listeners
        hub.addListener(&collector);

        // Set console font parameters (Make easier to follow instructions)
        CONSOLE_FONT_INFOEX cfi;
        cfi.cbSize = sizeof cfi;
        cfi.nFont = 0;
        cfi.dwFontSize.X = 0;
        cfi.dwFontSize.Y = 16;
        cfi.FontFamily = FF_DONTCARE;
        cfi.FontWeight = FW_NORMAL;
        wcscpy_s(cfi.FaceName, L"Consolas");
        SetCurrentConsoleFontEx(GetStdHandle(STD_OUTPUT_HANDLE), FALSE, &cfi);

        std::cout << "Please follow the instructions to provide gestural data!" << std::endl;
        std::cout << "Allow a couple seconds while Terminator Myo warms up to arm... " << std::endl << std::endl;
        Sleep(5000);            // suspend execution of current/active thread for time-argument

        // Finally we enter our main loop.
        for (int i = 0; i < sizeof(gestures) / sizeof(*gestures); i++){
            system("cls");

            std::cout << std::endl; std::cout << "\n\n\n \t\t Perform:  " << gestures[i] << " for (5) secs" << std::en

            // Get current CPU time
            double startTime = GetTickCount();
            double currentTime = 0;

            while ((GetTickCount() - startTime) <= 1000) {}; // wait for 1 extra sec for user change
```

```
        std::cin.ignore();
        return 1;
    }
}
```

2. **UserApp (Main application code including PCIE and Characteristics gesture recognition)**

```
cfi.dwFontSize.X = 0;
cfi.dwFontSize.Y = 16;
cfi.FontFamily = FF_DONTCARE;
cfi.FontWeight = FW_NORMAL;
wcscpy_s(cfi.FaceName, L"Consolas");
SetCurrentConsoleFontEx(GetStdHandle(STD_OUTPUT_HANDLE), FALSE, &cfi);

std::cout << "\n\n\n\n\n\n\t\t\t\t\t   Welcome to Terminator Arm Gen. (1.0) !" << std::endl;
std::cout << "\t\t\t\t\t  Allow a couple seconds while Terminator Myo warms up to arm... " << std::endl << std::endl;
Sleep(5000);            // suspend execution of current/active thread for time-argument


// Finally we enter our Real-time loop.
system("cls");
std::cout << "\n\n\n\t\t\t\t\t\t\tMode:  " << mode << "\n\n\t\t\t\t\t\tGR_method:  " << GR_method << " \n";
std::cout << "\t\t\t\t\t\tTerminator Myo is now ready to listen for gestures! \n"; Sleep(2000);
system("cls");
Listen:
while (collector.onArm){

    // Get current CPU time
    double startTime = GetTickCount();
    double currentTime = 0;

        //while ((GetTickCount() - startTime) <= 3000)
        //{
        // Run Myo hub
        hub.run(1);

        //} // wait for 1 extra sec for user change

        //collector.writeCalData(gestures[0]);

        // After processing events, we call the writeData() function to write new data to our outfile/array
        collector.writeRunData();

        // Parse input data stream
        TerminatorGR.ParseData(GR_method);
```

```
try {
    system("mode 650");   // Full screen mode
    // First, we create a Hub without any application identifier (I deemed it unnecessary), the Hub provides
    myo::Hub hub("");

    std::cout << "\t\t\t\t\t\t Attempting to find HU Terminator Myo..." << std::endl;

    // Next, we attempt to find our Myo. If it's already paired in Myo Connect, this will return our Myo imm
    // waitForMyo() takes a timeout value in milliseconds
    myo::Myo* myo = hub.waitForMyo(0);              // Times-out until Myo is found!

    hub.setLockingPolicy(hub.lockingPolicyNone);      // Keep Terminator Myo unlocked

    // If waitForMyo() returned a null pointer, we failed to find our Myo, so exit with an error message
    if (!myo) {
        throw std::runtime_error("Unable to find HU Terminator Myo! \n Please Check MyoConnect!");
    }

    // We've found our Myo!
    std::cout << "\t\t\t\t\t\t Now connected to HU Terminator Myo! " << std::endl << std::endl;

    // Next we enable EMG streaming on the found Myo
    myo->setStreamEmg(myo::Myo::streamEmgEnabled);
    myo->unlockHold;

    // Next we construct an instance of our DeviceListener, so that we can register it with the Hub
    DataCollector collector;
    GestureRec TerminatorGR;    // Instance of Terminator Myo_FANN

    // Hub::addListener() takes the address of any object whose class inherits from DeviceListener, and will
    // Hub::run() to send events to all registered device listeners
    hub.addListener(&collector);

    // Set console font parameters (change some default cmd properties)
    CONSOLE_FONT_INFOEX cfi;
    cfi.cbSize = sizeof cfi;
    cfi.nFont = 0;
```

I'm Cortana. Ask me anything.

```
    float temp = 0.0; int largestindex = 0;
    finalvalue[0] = Rest; finalvalue[1] = Thumb; finalvalue[2] = Index; finalvalue[3] = Middle; finalvalue[4] = R
    for (int i = 0; i < 7; i++)
    {
        if (temp < finalvalue[i]){
            temp = finalvalue[i]; largestindex = i;
        }
    }

    if (largestindex == 0) gesture = "Rest"; if (largestindex == 1) gesture = "Thumb"; if (largestindex == 2) ges
    if (largestindex == 3) gesture = "Middle"; if (largestindex == 4) gesture = "Ring"; if (largestindex == 5) ge
    if (largestindex == 6) gesture = "Wrist";

    // Print gesture

    // Display weights
    //std::cout << "\n\n\n      Final values:";
    //for (int i = 0; i < no_gestures; i++)
    //std::cout << "   " << _gestures[i] << ": " << finalvalue[i] << " |";
    system("cls");
    std::cout << "\n\n\n\n\n\n\n\n\t\t\t\t\t\t\tGesture detected: " << gesture << "!\n";
    Sleep(2000);



ain(int argc, char** argv)

/ Define Calibration(Cal.) mode test-gestures
td::string gestures[12] = { "*REST/RELAX* position", "*(THUMB)* contraction", "*REST/RELAX* position", "*(INDEX
    "*(MIDDLE fing.)* contraction", "*REST/RELAX* position", "*(RING fing.)* contraction", "*REST/RELAX* position
    "*(PINKY fing.)* contraction", "*REST/RELAX* position", "*(HAND)* contraction" };
td::string mode, GR_method = "Characteristics";
onst static bool toCalibrate = 0;  // Set user mode i.e calibration w/user data or run on Stored data
onst static int datasize = 240;    // datasize for training and testing
```

```
    //std::cout << "\t\t\t\tRest: " << Rest << " |Thumb: " << Thumb << " |Index: " << Index << " |Middle: "

    // Branch 3 (Ring, Wrist) ** Methinks this oscillates Ring/Wrist result a little!
    /*if (Percents[4][5] <= 61)
    {
        if (Percents[4][0] > 58) Wrist += 1.0;
        else
            Ring += 1.0;
    }*/

    // ** Print Branch 3 results **
    std::cout << "\n\n\t\t\tBranch 3: \n";
    //std::cout << "\t\t\t\tRest: " << Rest << " |Thumb: " << Thumb << " |Index: " << Index << " |Middle: "

    // Branch 4 (Thumb, Index, Middle, Ring, Pinky, Wrist)
    bool PodFlag = 0;
    for (int i = 1; i < no_pods; i++){
        if (Percents[4][0] > Percents[4][i])
            PodFlag = 1;    // Flag set if Pod 0 is not the smallest
    }
    if (!PodFlag){
        if (abs(Percents[4][5] - Percents[4][0]) <= 10) Ring += 1.0;
        if (abs(Percents[4][5] - Percents[4][0]) > 10)Wrist += 1.0;
    }

    // ** Print Branch 4 results **
    //std::cout << "\n\n\n\t\t\tBranch 4: \n";
    //std::cout << "\t\t\t\tRest: " << Rest << " |Thumb: " << Thumb << " |Index: " << Index << " |Middle: "


    // Insert more Branches here


    // 4. Select based on final Weight values
```

```
        Pod0 = 0, Pod1 = 0, Pod2 = 0, Pod3 = 0, Pod4 = 0, Pod5 = 0, Pod6 = 0, Pod7 = 0;
    }


    // 3. Weight/Decision Branches
    bool Flag = 0;
    // Branch 1 (Rest)
    for (int i = 0; i < 6; i++){
        if (Percents[4][i + 1] < 90.0) Flag = 1;
    } if (!Flag) Rest = 4.0;


    // Branch 2  (Thumb, Index, Middle, Ring, Pinky, Wrist)
    if (Percents[4][3] >= 84.0){

        if (Percents[4][5]> 85.0) Thumb += 1.0;

        else if (Percents[4][5] < 65.0) Ring += 1.0;

        else{
            if (Percents[4][0] > 45.0 && Percents[4][6] > 80.0){       // Changed up stuff in this nest
                    if (Percents[4][4] > 84.0) {
                        if (Percents[4][1] < 84.0) Index += 1.0;
                    }
                    else
                        if (Percents[4][1] > 84.0) Middle += 1.0;
            }
            else{
                Pinky += 0.8; Middle += 0.1; Index += 0.1;
            }
        }
    }
    else{
        Wrist += 1.0;
    }

    // ** Print Branch 2 results **
    //std::cout << "\n\n\t\t\tBranch 2: \n";
```

```
int checkvalues[] = { 4, 5, 6, 7, 8 };
float Percents[5][no_pods]; float finalvalue[7];

// GR Approach
// 2. Compute percentages for values between 4 through 8
for (int i = 0; i < sizeof(checkvalues) / sizeof(checkvalues[0]); i++){
    for (int j = 0; j < datasize; j++){            // Score no_times data is less than 8
        if (input_Char[j] < checkvalues[i]){
            if (j == 0 || j % 8 == 0)  Pod0++;                   // Pod 0
            else if (j == 1 || (j - 1) % 8 == 0)  Pod1++;        // Pod 1
            else if (j == 2 || (j - 2) % 8 == 0)  Pod2++;        // Pod 2
            else if (j == 3 || (j - 3) % 8 == 0)  Pod3++;        // Pod 3
            else if (j == 4 || (j - 4) % 8 == 0)  Pod4++;        // Pod 4
            else if (j == 5 || (j - 5) % 8 == 0)  Pod5++;        // Pod 5
            else if (j == 6 || (j - 6) % 8 == 0)  Pod6++;        // Pod 6
            else Pod7++;                                         // Pod 7
        }
    }

    // Compute percent based off no_times
    Percents[i][0] = (Pod0*no_pods * 100.0) / datasize; Percents[i][1] = (Pod1*no_pods * 100.0) / datas
    Percents[i][3] = (Pod3*no_pods * 100.0) / datasize; Percents[i][4] = (Pod4*no_pods * 100.0) / datas
    Percents[i][6] = (Pod6*no_pods * 100.0) / datasize; Percents[i][7] = (Pod7*no_pods * 100.0) / datas

    // ** Print Percents **
    //system("cls");
    //std::cout << "\n\n\t\t\t\t";
    //for (int i = 0; i < no_pods; i++)
        //std::cout << "Pod" << i << ": " << Percents[4][i] << "|";

    // Reset Pod values
```

```
    }

    // These values are set by onArmSync() and onArmUnsync() above.
    bool onArm;
    myo::Arm whichArm;
};

class GestureRec {
public:
    std::ifstream RunData;
    std::string prev_gesture, gesture;
    static const int datasize = 8;
    int input_Char[datasize];

    void ParseData(std::string GR_method)
    {
        RunData.open("C:\\Users\\Ayotunde\\Documents\\Visual Studio 2013\\Projects\\fann-master\\fann-master\\e

        // Grab test data
        for (int i = 0; i < datasize; i++){
            RunData >> input_Char[i];           //Here, using .txt files           // input_Net[i] = Data[i]; /
            if (input_Char[i] < 0) input_Char[i] = sqrt(input_Char[i] * input_Char[i]);     // compute absolute
        }

        // Delete RuntimeData.txt
        RunData.close();
        Sleep(2000);
        std::remove("C:\\Users\\Ayotunde\\Documents\\Visual Studio 2013\\Projects\\fann-master\\fann-master\\ex
    }

    void Translate_Characteristics(int Data[datasize])
    {
        // Function-specific Variables
        //const int dataSize = 8;
        const int no_pods = 8; std::string gesture = "Rest";

        float Rest = 0.0, Thumb = 0.0, Index = 0.0, Middle = 0.0, Ring = 0.0, Pinky = 0.0, Wrist = 0.0;
```

```
            std::cout << '\r';

            // write out the EMG classifier data
            for (size_t i = 0; i < emgSamples.size(); i++) {
                std::ostringstream oss;
                oss << static_cast<int>(emgSamples[i]);        // convert 8-bit array into int
                std::string emgString = oss.str();

                if (NewLineflag == 1){
                    DataFile << "\n";
                    NewLineflag = 0;
                }

                // Write to outfile
                DataFile << emgString << " ";
            }
            NewLineflag = 1; no_lines++;
            std::cout << std::flush;
            DataFile.close();       // flush buffer to outfile
//      }

            /* Use array mode to write data, here!
           int j = 0;    // Reset Array starting write-index

            while (j < no_testDatalines*emgSamples.size()){
            // write out the EMG classifier data
            for (size_t i = 0; i < emgSamples.size(); i++) {
            std::ostringstream oss;
            oss << static_cast<int>(emgSamples[i]);        // convert 8-bit array into int
            std::string emgString = oss.str();
            DataArray[j] = static_cast<int>(emgSamples[i]); j++;
            }
            }*/
    }

    void DeleteCalFile()
    {
        std::remove("C:\\Users\\Ayotunde\\Documents\\Visual Studio 2013\\Projects\\fann-master\\fann-master\\ex
```

```
emg-data-sample-VisualStudio2013                DataCollector
                if (NewLineflag == 1){
                    DataFile << "\n";
                    NewLineflag = 0;
                }

                // Write current gesture value to outfile
                if (update != gesture){
                    no_lines = 0;   // Reset no. of lines
                    if (update == "*REST/RELAX* position")      gesture_label = 0;
                    else if (update == "*(THUMB)* contraction")   gesture_label = 0.2;
                    else if (update == "*(INDEX fing.)* contraction") gesture_label = 0.4;
                    else if (update == "*(MIDDLE fing.)* contraction") gesture_label = 0.6;
                    else if (update == "*(RING fing.)* contraction")   gesture_label = 0.8;
                    else if (update == "*(PINKY fing.)* contraction")  gesture_label = 0.9;
                    else if (update == "*(HAND)* contraction") gesture_label = 1.0;
                    DataFile << gesture_label << "\n"; update = gesture;
                    if (gesture == "*(HAND)* contraction") lastdataval = 1.0;
                }

                // Write to outfile
                DataFile << emgString << " ";
            }
            NewLineflag = 1; no_lines++; title_flag = 1;
            if (lastdataval == 1.0 && no_lines == no_testDatalines) DataFile << "\n" << lastdataval;
            std::cout << std::flush;
            DataFile.close();       // flush buffer to outfile
        }
    }

    // We define this function to write the current values that were updated by the on...() functions ab
    void writeRunData()
    {
        NewLineflag = 0; no_lines = 0;
    //  while (no_lines != no_testDatalines){       // Check to ensure we're only writing how many data
            // Create and open outfile
            DataFile.open(Run_filepath, std::fstream::app);

            // Clear the current line
```

```
    {
        for (int i = 0; i < 8; i++) {
            emgSamples[i] = emg[i];
        }
    }

    // There are other virtual functions in DeviceListener that we could override here, like onAccelerometerD
    // For this application, the functions overridden above are sufficient

    // The values of this array is set by onEmgData() above
    std::array<int8_t, 8> emgSamples;
    int no_testDatalines = 30, DataArray[240], no_lines = 0;    // ** SET the desired no. of training & test

    std::string update = "*REST/RELAX* position";
    std::fstream DataFile;
    std::string Cal_filepath = "C:\\Users\\Ayotunde\\Documents\\Visual Studio 2013\\Projects\\fann-master\\fa
    std::string Run_filepath = "C:\\Users\\Ayotunde\\Documents\\Visual Studio 2013\\Projects\\fann-master\\fa
    double gesture_label, lastdataval;
    bool NewLineflag = 0, title_flag = 0;

    // We define this function to write the data values from Cal. in calibration format required by the Myo_F
    // ** NB ** : This function could use a lot of optimization but it works as is.
    void writeCalData(std::string gesture)
    {
        if (no_lines != no_testDatalines || update != gesture){       // Check to ensure we're only writing h
            // Create and open outfile
            DataFile.open(Cal_filepath, std::fstream::app);
            if (!title_flag) DataFile << "12 80 1\n";

            // Clear the current line
            std::cout << '\r';

            // write out the EMG classifier data
            for (size_t i = 0; i < emgSamples.size(); i++) {
                std::ostringstream oss;
                oss << static_cast<int>(emgSamples[i]);       // convert 8-bit array into int
                std::string emgString = oss.str();
```

```
class DataCollector : public myo::DeviceListener {
public:
    // Device listener functions
    DataCollector() : onArm(true), emgSamples()
    {}

    void attachByMacAddress(std::string macAddress)
    {

    }

    void onArmSync(myo::Myo* myo, uint64_t timestamp, myo::Arm arm, myo::XDirection xDirection, float rotati
    {
    system("cls"); //std::cout << "ARM SYNC!!" << std::endl;
    onArm = true;
    whichArm = arm;
    }
    // onArmUnsync() is called whenever HUTerminator Myo has detected that it was moved from a stable positi
    // it recognized the arm. Typically this happens when someone takes it off of their arm, but it can also
    // when it's moved around on the arm.
    void onArmUnsync(myo::Myo* myo, uint64_t timestamp)
    {
        system("cls"); std::cout << "\n\n\n\n\t\t\t\t\t\tTerminator Myo UN-SYNC!!" << std::endl;
        onArm = false;
        Sleep(5000);
    };

    // onUnpair() is called whenever the Myo is disconnected from Myo Connect by the user
    void onUnpair(myo::Myo* myo, uint64_t timestamp)
    {
        // We've lost our Myo.
        // Zeros are seen in output file if there was a disconnect so test can be repeated
        emgSamples.fill(0);
    }

    // onEmgData() is called whenever a paired Myo has provided new EMG data, and EMG streaming is enabled
    // Classifier output data
    void onEmgData(myo::Myo* myo, uint64_t timestamp, const int8_t* emg)
```

```
// Entry point for Terminator Arm UserApplication
// By:
// Howard University Terminator Arm Senior Design 2015/2016.
//

// Dependencies:
// 1. MyoConnect
// 2. Myo_Fann.h

// Description:
// This is used in a larger system. This module provides EMG data streaming and gesture recognition based on 7
// EMG streaming is only supported for one Myo at a time
// Terminator Myo MAC address: db-fa-0c-69-14-78 (Can be used alternatively instead of MyoConnect to attach ou
// http://diagnostics.myo.com/ provides diagnostic data of Myo (connected in MyoConnect) on a pc
// NB: Replacing system("cls") is recommended and advised.


// #define _USE_MATH_DEFINES
#include <iostream>
#include <stdio.h>
#include <cmath>
#include <string>
#include <iomanip>
#include <iomanip>
#include <algorithm>
#include <array>
#include <sstream>
#include <stdexcept>
#include <fstream>
#include "windows.h"    // contains Sleep library
#include <time.h>
#include <process.h>
#include <math.h>

#include <myo/myo.hpp>            // Myo default header library


// DataCollector class inheriting member functions from myo::DeviceListener
```

```
                // Call GR_method
                TerminatorGR.Translate_Characteristics(collector.DataArray);
    }

        while (collector.onArm == false){
            system("cls");  std::cout << "\n\n\n \t\tPlease re-sync Terminator Myo for best results!";
        }
        goto Listen;

        // Tidy up & End program
        system("cls");
        std::cout << "\n       *****  Thank you from HU Team Terminator!  *****\n\n";
        Sleep(3000);
        return 0;

    }
    catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        std::cerr << "Press enter to continue.";
        std::cin.ignore();
        return 1;
    }
}
```

3. Arduino Code for servo motor control listing

```
    midServo.write(pos);
    delay(25);}
  }
  else if (ringinput > 500){
        Serial.println(F("(Ring finger Gesture detected!)")); Serial.println(F("(Moving Termina
  for(pos = 0; pos < 180; pos += 1){    // goes from 0 degrees to 180 degrees in steps of 1 degr
    ringServo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(25);}                        // waits  for the servo to reach the position
  for(pos = 180; pos>=1; pos-=1){      // goes from 180 degrees to 0 degrees
    ringServo.write(pos);
    delay(25);}
  }
  else if (pinkyinput > 500){
     Serial.println(F("(Pinky finger Gesture detected!)")); Serial.println(F("(Moving Terminato
  for(pos = 0; pos < 180; pos += 1){    // goes from 0 degrees to 180 degrees in steps of 1 degr
    pinkyServo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(25);}                        // waits  for the servo to reach the position
  for(pos = 180; pos>=1; pos-=1){      // goes from 180 degrees to 0 degrees
    pinkyServo.write(pos);
    delay(25);}
  }
  else if (wristinput > 500){
    Serial.println(F("(Wrist Gesture detected!)")); Serial.println(F("(Moving Terminator Wrist!
  for(pos = 0; pos < 180; pos += 1){    // goes from 0 degrees to 180 degrees in steps of 1 degr
    wristG.write(pos); delay(25);}
  for(pos = 180; pos>=1; pos-=1){      // goes from 180 degrees to 0 degrees
    wristG.write(pos); delay(25);}
  }
```

```
  Serial.print("RingPose: "); Serial.println(ringinput);
  Serial.print("PinkyPose: "); Serial.println(pinkyinput);
  Serial.print("WristPose: "); Serial.println(wristinput);

}

void ServoController()
{
  if (thumbinput > 500){
        Serial.println(F("(*** Thumb finger Gesture detected! ***)")); Serial.println(F("(Moving
  for(pos = 0; pos < 180; pos += 1){   // goes from 0 degrees to 180 degrees in steps of 1 degre
    thumbServo.write(pos); delay(25);}   // Tells servo to go to position & waits  for the servo
  for(pos = 180; pos>=1; pos-=1){     // goes from 180 degrees to 0 degrees
    thumbServo.write(pos); delay(25);}
  }
  else if (indexinput > 500){
        Serial.println(F("(Index finger Gesture detected!)")); Serial.println(F("(Moving Termina
  for(pos = 0; pos < 180; pos += 1){   // goes from 0 degrees to 180 degrees in steps of 1 degre
    indexServo.write(pos); delay(25);}
  for(pos = 180; pos>=1; pos-=1){
    indexServo.write(pos); delay(25);}
  }
  else if (midinput  > 500){
        Serial.println(F("(Middle finger Gesture detected!)")); Serial.println(F("(Moving Termin
  for(pos = 0; pos < 180; pos += 1){   // goes from 0 degrees to 180 degrees in steps of 1 degre
    midServo.write(pos);               // tell servo to go to position in variable 'pos'
    delay(25);}                       // waits  for the servo to reach the position
  for(pos = 180; pos>=1; pos-=1){     // goes from 180 degrees to 0 degrees
```

```
  // attaches the servo on pin# to the servo object
  thumbServo.attach(3); indexServo.attach(5); midServo.attach(6); ringServo.attach(9); pinkyServ
}

// Main Control code
void loop()
{
  thumbinput = analogRead(thumbpose);indexinput = analogRead(indexpose);midinput = analogRead(mi

  Serial.println(F("-------------------------"));

  // Debug purposes
  AnalogReadPrint();
  //TestwLEDs();

  ServoController();

  delay(1000);
}

void AnalogReadPrint()
{
// Print Analog reads
  Serial.print("ThumbPose: "); Serial.println(thumbinput);
  Serial.print("IndexPose: "); Serial.println(indexinput);
  Serial.print("MidPose: "); Serial.println(midinput);
  Serial.print("RingPose: "); Serial.println(ringinput);
```

Control_Servo

```
// Controls a Servo Motor
// Terminator Arm senior design project.
// Howard University


// Terminator Intel galileo code
// Warning: It is NOT advised to uncomment all of the print commands, as you can run out of hea



#include <Servo.h>
#include <Wstring.h>
// Keep a string inside of PROGMEM and prevent it from consuming RAM at run-time
#define F(string_literal) (reinterpret_cast<const __FlashStringHelper *>(PSTR(string_literal)))

int thumbpose = A0, indexpose = A1, midpose = A2, ringpose = A3, pinkypose = A4, wristpose = A5
Servo thumbServo, indexServo, midServo, ringServo, pinkyServo, wristG;        // Servo object
int thumbinput, indexinput, midinput, ringinput, pinkyinput, wristinput;
//int thumbLED = 3, indexLED = 5, midLED = 6, ringLED = 9, pinkyLED = 10, wristLED = 13;
int pos;
bool isLEDTest = 1;

void setup()
{
  Serial.println(F("Seting up!"));
  Serial.begin(9600);          // set baud-rate
  SetpinMode();
```

```
  else {Serial.println(F("(No gesture detected!)"));}         // do nothing if no input pose is
}

void SetpinMode()
{
  pinMode(thumbpose, INPUT); pinMode(indexpose, INPUT); pinMode(midpose, INPUT);
  pinMode(ringpose, INPUT); pinMode(pinkypose, INPUT); pinMode(wristpose, INPUT);
  //pinMode(thumbServo, OUTPUT); pinMode(indexServo, OUTPUT); pinMode(midServo, OUTPUT);
  //pinMode(ringServo, OUTPUT); pinMode(pinkyServo, OUTPUT); pinMode(wristServo, OUTPUT);
}

/*void TestwLEDs()
{
  if (thumbinput > 700){
    digitalWrite(thumbLED, HIGH);    digitalWrite(indexLED, LOW);     digitalWrite(midLED, LOW)
        digitalWrite(wristLED, LOW);     digitalWrite(ringLED, LOW);     digitalWrite(pinkyLED,
  }
  else if (indexinput > 700){   digitalWrite(thumbLED, LOW);    digitalWrite(indexLED, HIGH);
        digitalWrite(wristLED, LOW);     digitalWrite(ringLED, LOW);     digitalWrite(pinkyLED,
  }
  else if (midinput > 700){   digitalWrite(thumbLED, LOW);    digitalWrite(indexLED, LOW);
        digitalWrite(wristLED, LOW);     digitalWrite(ringLED, LOW);     digitalWrite(pinkyLED,
  }
  else if (ringinput > 700){   digitalWrite(thumbLED, LOW);    digitalWrite(indexLED, LOW);
        digitalWrite(wristLED, LOW);     digitalWrite(ringLED, HIGH);     digitalWrite(pinkyLED
  }
  else if (pinkyinput > 700){   digitalWrite(thumbLED, LOW);    digitalWrite(indexLED, LOW);
```

4. Major part of FPGA PCIE implementation (Rest is available on github)

```
wire [7:0]gpiobus;
wire reset_n;

//=======================================================
//  Structural coding
//=======================================================

assign reset_n = 1'b1;


    de2i_150_qsys u0 (
        .clk_clk                                  (CLOCK_50),                              //
        .reset_reset_n                            (reset_n),                            //
        .pcie_ip_refclk_export                    (PCIE_REFCLK_P),                    //              pcie_ip_r
        .pcie_ip_pcie_rstn_export                 (PCIE_PERST_N),                 //          pcie_ip_pcie_rst
        .pcie_ip_rx_in_rx_datain_0                (PCIE_RX_P[0]),                //              pcie_ip_rx_in
        .pcie_ip_tx_out_tx_dataout_0              (PCIE_TX_P[0]),              //          pcie_ip_tx_out.t
        .led_external_connection_export           (LEDG[3:0]),           //     led_external_connection.export
        .gpio_external_connection_export          (gpiobus),
        .button_external_connection_export        (KEY)          // button_external_connection.export
    );

assign PCIE_WAKE_N = 1'b1;  // 07/30/2013, pull-high to avoid system reboot after power off

assign GPIO[7:0] = gpiobus[7:0];
```