IIA Project SB3 Final Report: Real Time Motion and Gesture Tracking Smart Gloves

Aidas Liaudanskas (al747)

July 29, 2017

Contents

1 Summary **2**

2 Introduction **2**

3 System Design Specifications **2**

4 Hardware **4**

4.1 Microcontroller 4

4.2 Flex Sensors 6

4.3 Accelerometers 6

4.4 Circuit boards 7

5 Microcontroller Software **8**

6 3D Simulation Software **9**

7 Conclusions and Possible Improvements **10**

8 Appendix A: Code **11**

9 Appendix B: Pictures **23**

10 Appendix C: How to Make a Cheap Flex Sensor **26**

First Report **27**

Marketing Datasheet **30**

1 Summary

This report covers the CUED IIA Data Logger project. The goal was to design a microcontroller powered device that would do something interesting. We worked in groups of 2. At the end of the project the team has produced a real time motion and gesture tracking smart gloves. The report includes design specifications, circuit descriptions, challenges faced, software flow, tests performed, application(s) and conclusions.

2 Introduction

Our initial idea was to make a pair of smart gloves. I wanted to make a BSL (British Sign Language) translator, Luke wanted to go with a 3D bomb defusion/soldering simulation. Since both me and Luke had quite a bit of previous experience with Arduino and electronics in general, we decided to go for something different, so both of us could learn something new. Since we both had some BLE Nano boards lying around from one of the past hackathons, we decided to go with this microcontroller(MCU).

As we started building the project, we have discovered more and more unforeseen barriers, so we had to adjust our goals slightly and ended up with something in between both ideas - smart gloves with real-time position and gesture tracking capabilities. In other words - the basis, on which both initial projects could be built.

3 System Design Specifications

After many iterations and feasibility checks we have came up with the following requirements for our system:

- Sense the position of individual fingers
- Sense the orientation of the wrist
- Be accurate enough, so that fairly-convoluted gestures could be recognised
- Provide real-time data:
- Minimum 30Hz refresh rate for the simulation;
- $\leq 150ms$ latency as anything above that becomes noticeably annoying for the user;
- Provide a realistic 3D simulation environment

These requirements resulted in the hardware and software block diagrams shown in Figures 1 and 2.

We have decided to make 3 boards to house all the components:

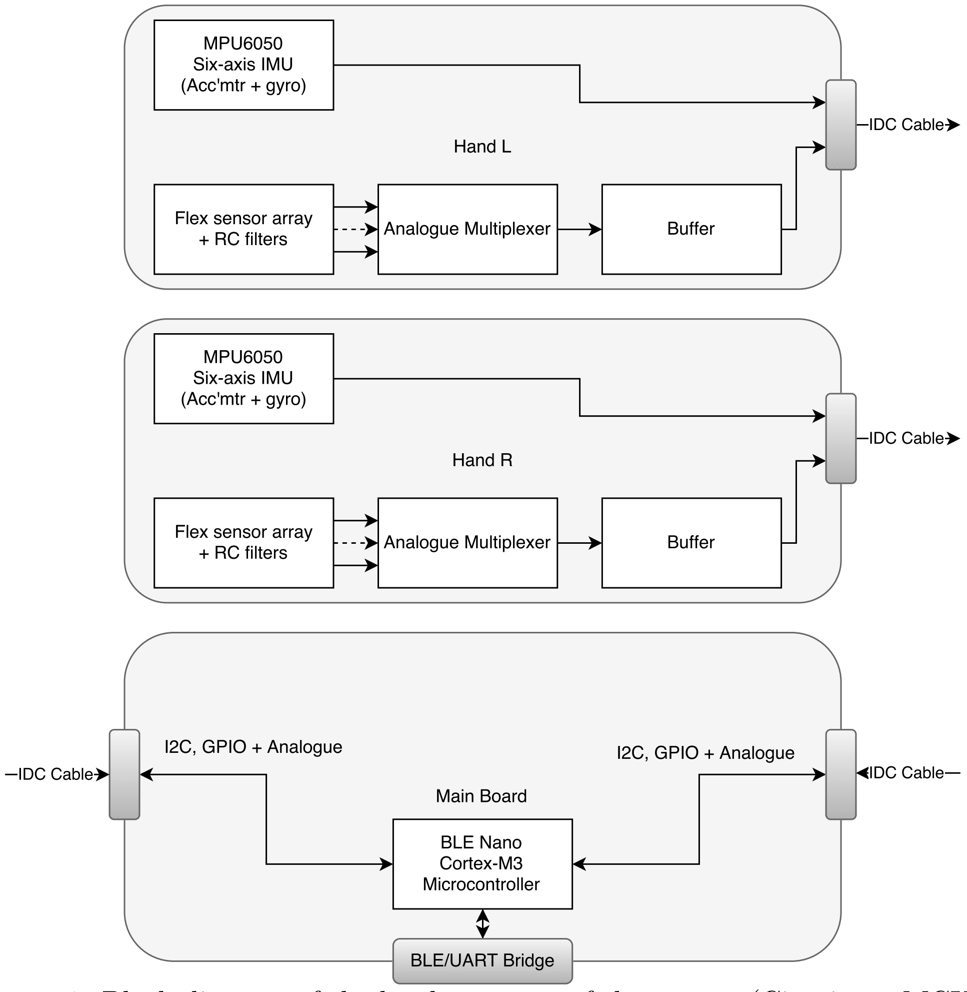


Figure 1: Block diagram of the hardware part of the system (Circuits + MCU).

- Left and Right sensor boards placed on user's hand. It houses the accelerometer board and flex sensors.
- Main board, where the MCU and the power supply are placed. It is intended to be worn around the neck, as a necklace.

Work was allocated as follows:

- Luke does 3D simulation and hand circuits
- Aidas does the mbed code for the MCU and the main board

The work was not strictly independent - we reached out to each other for suggestions/help if any of us got stuck and the team dynamics were really good.

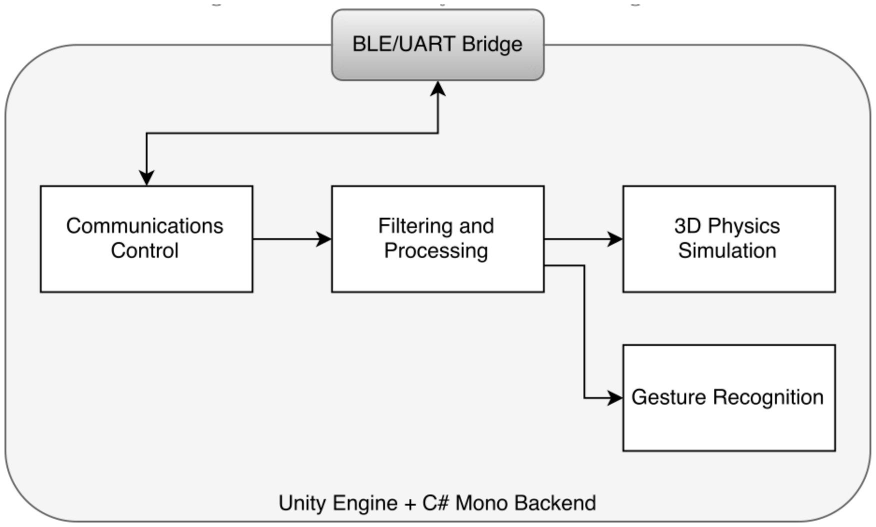


Figure 2: Block diagram of the software part of the system (runs on PC/Smartphone).

4 Hardware

4.1 Microcontroller

The full name of the microcontroller is BLE Nano development board by ReadBear-Labs. As shown in Figure 3, the board is tiny. Furthermore, it has a Bluetooth Low Energy (BLE) support in its Nordic nRF51822 SoC (System on a Chip). Both of these qualities make it a perfect choice for a wearable application.

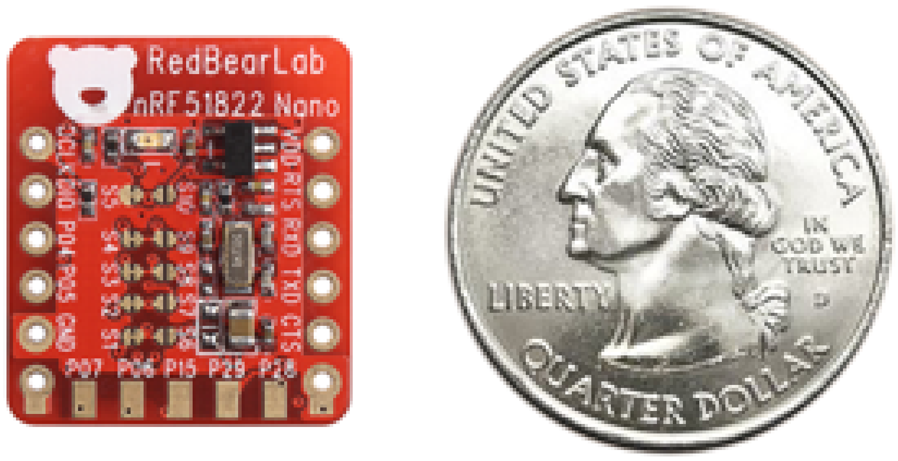


Figure 3: The development board is no bigger than a quarter (≈ 2cm by 2cm). Image courtesy of ReadBearLabs.

While the RBL BLE Nano is slightly cheaper than the Arduino, it houses an ARM Cortex-M3 32-bit processor, which is significantly more powerful than the 8-bit AVR used in Arduino. This results in a much faster execution of tasks at a similar active power consumption, meaning that the average power consumption is lower.

The development is done in C++ and the development environment can be chosen from:

- an online ARM mbed compiler;
- patched Arduino IDE
- native SDK using arm-gcc and Makefile.

Although it is much slower than compiling on my machine, I chose the first option as it was the easiest to set up.

Small board dimensions come at a price of certain functionality: the number of pins is very limited, when compared to an Arduino board. The pinout is shown in Figure 4.

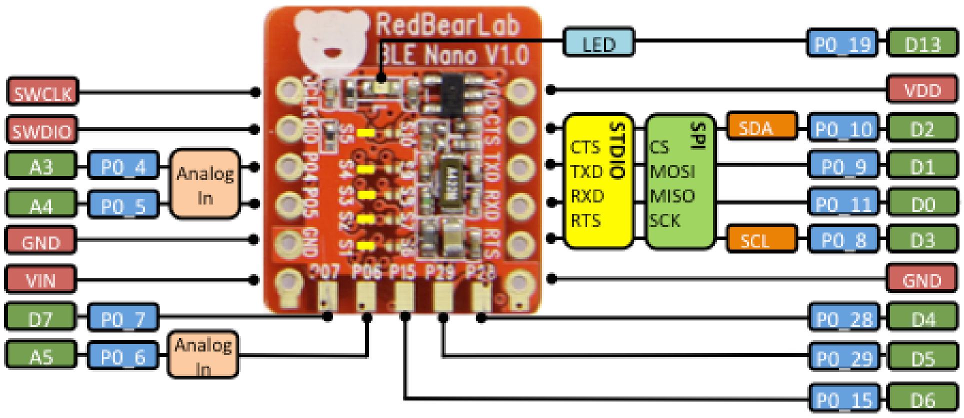


Figure 4: Block diagram of the software part of the system. Image courtesy of ReadBearLabs.

Everything sounds nice and perfect until this point. The fun part begins here...

The pinout was the first problem we had to overcome. Initially we have assigned every pin but two, because originally we intended to use the BLE to send the data to the computer. It took me around a week to get the Bluetooth peripheral (that is the equivalent for a slave in Bluetooth protocol terminology and it refers to the BLE Nano in this report) running. I was rather disappointed that there was no simple interface to get the BLE up and running: one either had to have an identical application as the ones in the examples, or had to read the whole BLE documentation and highly abstracted mbed code to understand how to write a new GATT service, to make the communication compatible with the specific application. I had to choose the latter way.

Furthermore, there was no simple way to parse the characteristic data from the Bluetooth controller in the PC because it was highly abstracted by the operating system. There was only one Python library that seemed to do almost what we wanted, but as the signal processing and Unity integrates through C#, this solution would require some rather difficult interprocess communication and it did not seem like something we would like to get involved in for a 4 week project. So we decided to move to an Android platform because bluetooth-app integration seemed to be better developed for this OS. We even found a library for Unity-Bluetooth integration and were very excited about it, just to find out that it had a bug in it - one transaction was limited to 3 bytes, the constant was hard-coded and to change that we would have to recompile quite a few Java libraries. It was at this point that our hope for a modern BLE-powered device was shattered - we realised that proper BLE integration would require too many man-hours to develop properly.

We decided to stick to good, old and simple Serial communication (UART), as it is well developed and quite easy to implement. This required two (we hoped it was two) pins, and those were the last two pins we had left on the boar. The hopes were high as the rest of the project depended on working serial link. Aaaand we were let down again. Nothing worked and we had no idea why. Even the I2C communication ceased to function. Little did we know that there was another bug in a hugely abstracted and **not open source** library for ARM MCUs.

We were fortunate enough to have a friend, who owned a logic analyser (shout out to Luke Shillaber, appreciate the help!). Only then were we able to grasp that, even though we thought we disabled the CTS and RTS pins for the serial communication link, they were reserved and could not be used for the I2C communication. Luke (my team-mate) has written some I2C communication code to try to override those pins. We were not successful again.

Fortunately, I have noticed a redundancy in the pin assignment, and was able to free up 3 pins by changing the connections a bit. Due to this lucky oversight, we were able to run I2C on these pins and finally got everything to work. More on that later.

4.2 Flex Sensors

For some reason **cough* patents *cough** a flex sensor on the market can cost anywhere between £6 and £15. As our budget was under £15, it was clearly unsuitable for anyone who had more than one finger... Unless, we made our own flex sensors. This is exactly what we decided to do. After some research and experimentation, we have decided that this strain-sensitive conducting plastic called Velostat is the most suitable for our application. The sheet cost us £5.5 for the amount that would be enough for $\approx$ 25 flex sensors and around 15 minutes of work per sensor. We were surprised by the results: the resistance changes between $\approx 600\Omega$ (finger closed) and $\approx 2000\Omega$ (finger open). Combined with a potential divider, this gave us a resolution of around $\frac{1}{6}$ of the full range of the 10-bit ADC: 170 bits.

Instructions on how to make a cheap flex sensor can be found in Appendix C.

4.3 Accelerometers

We have chosen to use MPU-6050 6-axis IMU (Inertial Measurement Unit). Used one on each hand circuit. We chose it mainly because it can be found for £1.40, already soldered on a breakout board with pins. You will notice the disparity between this price and the one submitted in the first report (attached at the end). This is because we bought it from a UK-based seller to have a next day delivery. The smaller price is for a month-long delivery from China.

It has internal digital low pass filter to smooth the signal. It is also capable of reading an external magnetometer sensor. Magnetometer would be useful as it would provide us with a reference direction and allow the measurement of the pitch. It has a sampling rate of 8kHz (1kHz after low-passing): fast enough for our application. I2C communication with adjustable address was also an important consideration for our pin-constrained design - we were able to have both devices connected to the same bus. I have written the software to setup the registers and sample the data. The accelerometer breakout board used is shown in Figure 5.

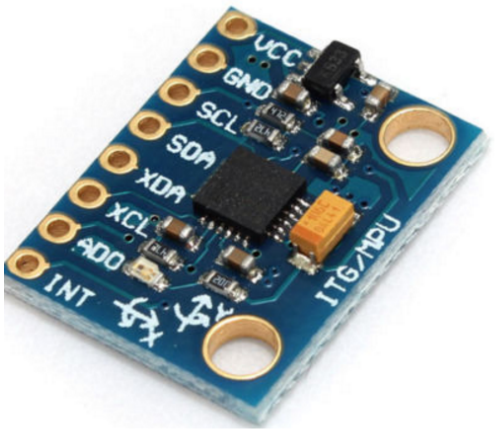


Figure 5: MPU 6050 breakout board.

4.4 Circuit boards

The flex sensors are connected to a potential divider circuit as shown in Figure 7. It has a RC low-pass filter connected to it with a cut-off frequency of 160 Hz. The mid-point of this potential divider circuit is taken into an analogue multiplexer (AMUX), which helps us save a lot of pins on the MCU. The sampling frequency (50Hz) is quite low compared to MCU capabilities, so cycling through all of them does not increase latency at all. This is were the previously mentioned redundancy was found: the AMUX select pins (8-channel AMUX has 3 select pins) can be driven by the same digital output pins of the MCU if we used two different analogue inputs. Fortunately the MCU did have two analogue inputs, and so exactly two digital pins were freed up: exactly what we needed to make the I2C communication work.

The ADC should generally be supplied with a low impedance input, so a buffer is added between the output of the AMUX and the input of the ADC. The circuit also sets the address pins of the accelerometer boards differently (low for LH; high for RH).

Hand boards are connected to the main board using IDC connectors. They make the design tidy, reliable and easy to connect/disconnect many times over. As they are symmetric, the notch position is indicated on the boards. Luke produced these. An excellent design. We used perfboards and point-to-point soldering to wire up the boards.

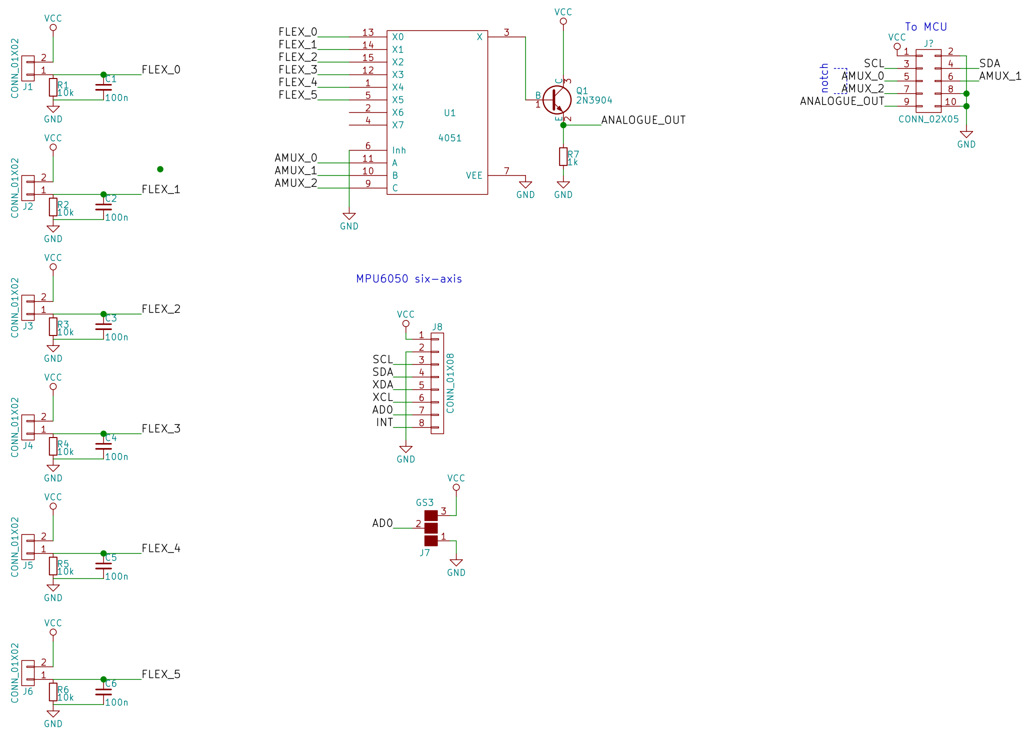


Figure 6: Hand circuit schematic.

The main board had only two connectors that were soldered point-to-point onto the MCU pins. It also had two pins brought out for the serial connection to the PC and two pins for 5V DC input and GND. A really trivial circuit from the top of my head.

5 Microcontroller Software

This was my responsibility. I was hoping to finish everything in a week, however in the end it took three weeks in total. As mentioned previously, a lot of reiterations were needed because some unknown bugs just kept showing up every time I have tried something new. The code is written in C++ using some of the mbed libraries provided by ARM. The code itself can be found in the Appendix A.

A more detailed software diagram of the communication between the MCU and PC is given below:

The code is mostly self explanatory, well-commented and the functions are named appropriately, so that the software flow would be easy to follow. The outline of the contents of the files:

- GlovesBT.cpp - main function and the BLE service setup. Sensor data is sent wirelessly

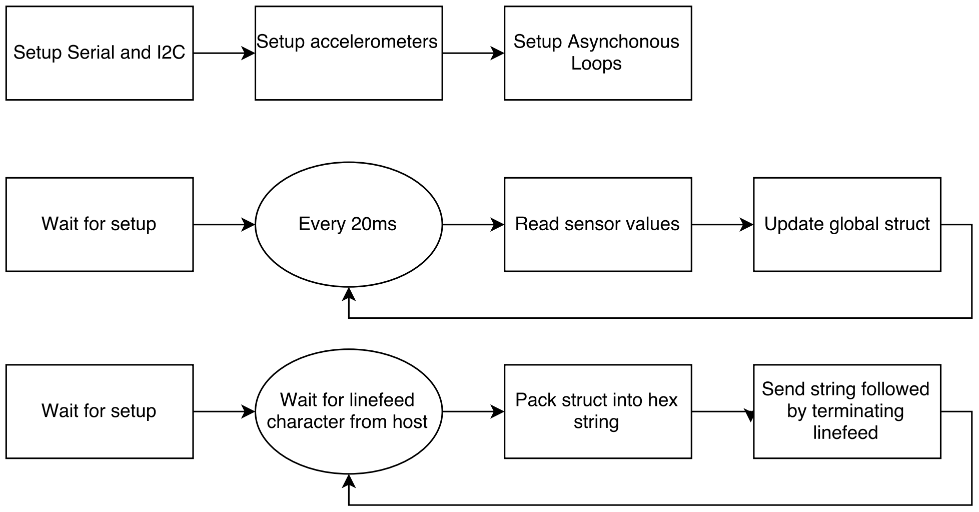


Figure 7: Detailed software flowchart.

- gloves.cpp - main function and serial port setup, data is sent over 2 wires at at a rate of 115200 bauds per second. The link is currently using around 50% of its maximum capacity
- softi2c.cpp - Luke's code to drive I2C. It did not work due to a high level abstraction of the ARM mbed library, so an inbuilt I2C library is used in practice.
- The MCU collects the data in structure called Datapacket readings, reverses all data to be big-endian and sends it over to the PC/Smartphone.

6 3D Simulation Software

This was Luke's responsibility and he did this entirely on his own. I have just helped him to come up with gestures and some of the desired functionality. It is written in C# using Unity. It does the following:

- data processing - uses quaternions to work out the Euler angles;
- maps flex sensor readings onto the finger positions;
- renders 3D graphics;

It is described in-depth in Luke's report. Some pictures of the 3D environment created can be found in Appendix B: Figures.

7 Conclusions and Possible Improvements

Some main things we learned from these fun four weeks can be summarised below:

- If you want to get up and running quick, without having to understand and debug a lot of highly abstracted code, use a widely used platform like Arduino. Plethora of online tutorials is a really valuable asset for the community.
- If, however, you decide to work with an unexplored platform, estimate the time it will take to do something first. Then triple it. This will give a more realistic estimate.
- At the moment, the device can perform as a virtual whiteboard or as means of telepresence during conference calls.
- To move on to more specific applications, like sign language translation, we would need to connect the elbow flex sensors and magnetometers. Magnetometers are more expensive and did not fit within our budget, but the circuit is ready to be upgraded - the connections are made open.

I would like to thank the organising team for setting up such an interesting project, and all the supervisors for their help - especially the guy with the MIT shirt!

8 Appendix A: Code

GlovesBT.cpp:

```
#include "mbed.h"
#include "ble/BLE.h"
#include "sensors.h"
#include <Serial.h>




DigitalOut led(LED1);
uint16_t customServiceUUID  = 0xABCD;
uint16_t readCharUUID       = 0xABCE;
uint16_t writeCharUUID      = 0xABCF;

static volatile bool triggerSensorPolling = false;

Serial pc(USBTX, USBRX);



const static char DEVICE_NAME[]        = "WorkGloves";
static const uint16_t uuid16_list[]        = {0xFFFF}; //
   Custom UUID, FFFF is reserved for development

/* Set Up custom Characteristics */
static uint16_t readValue[26] = {0x55, 0x33};
ReadOnlyArrayGattCharacteristic<uint16_t, sizeof(readValue)>
   readChar(readCharUUID, readValue);

static uint8_t writeValue[8] = {0x00};
WriteOnlyArrayGattCharacteristic<uint8_t, sizeof(writeValue)>
   writeChar(writeCharUUID, writeValue);


/* Set up custom service */
GattCharacteristic *characteristics[] = {&readChar, &writeChar
   };
GattService customService(customServiceUUID, characteristics,
   sizeof(characteristics) / sizeof(GattCharacteristic *));
```

```
void datapackettoarray(Datapacket todump, uint16_t *dest)
{       //Need a uint16_t array[26] input for correct
   functioning
        uint8_t j,k;

        for (j=0; j<12; j++)
        {
                *dest=todump.flex[j];
                dest++;
        }
        for(k=0; k<7; k++)
        {
                *dest=todump.acc[0][k];
                dest++;
        }

        for(k=0; k<7; j++, k++)
        {
                *dest=todump.acc[1][k];
                dest++;
        }



}


/*
 *  Restart advertising when phone app disconnects
 */
void disconnectionCallback(const Gap::
   DisconnectionCallbackParams_t *)
{
        BLE::Instance(BLE::DEFAULT_INSTANCE).gap().
           startAdvertising();
}

void periodicCallback(void)
{
        led = !led; /* Do blinky on LED1 while we're waiting
           for BLE events */

        /* Note that the periodicCallback() executes in
```

```
            interrupt context, so it is safer to do
           * heavy-weight sensor polling from the main thread.
             */
          triggerSensorPolling = true;
}

/*
 *  Handle writes to writeCharacteristic for screen data from
    phone
 */
void writeCharCallback(const GattWriteCallbackParams *params)
{
          /* Check to see what characteristic was written, by
             handle */
          if(params->handle == writeChar.getValueHandle()) {
                    /* Update the readChar with the value of
                       writeChar */
//                     BLE::Instance(BLE::DEFAULT_INSTANCE).
   gattServer().write(readChar.getValueHandle(), params->data,
    params->len);
          }
}
/*
 * Initialization callback
 */
void bleInitComplete(BLE::
   InitializationCompleteCallbackContext *params)
{
          BLE &ble           = params->ble;
          ble_error_t error = params->error;

          if (error != BLE_ERROR_NONE) {
                    return;
          }

          ble.gap().onDisconnection(disconnectionCallback);
          ble.gattServer().onDataWritten(writeCharCallback); //
             TODO: update to flush screen !!!

          /* Setup advertising */
          ble.gap().accumulateAdvertisingPayload(
             GapAdvertisingData::BREDR_NOT_SUPPORTED |
             GapAdvertisingData::LE_GENERAL_DISCOVERABLE); //
             BLE only, no classic BT
          ble.gap().setAdvertisingType(GapAdvertisingParams::
             ADV_CONNECTABLE_UNDIRECTED); // advertising type
          ble.gap().accumulateAdvertisingPayload(
             GapAdvertisingData::COMPLETE_LOCAL_NAME, (uint8_t
             *)DEVICE_NAME, sizeof(DEVICE_NAME)); // add name
          ble.gap().accumulateAdvertisingPayload(
             GapAdvertisingData::COMPLETE_LIST_16BIT_SERVICE_IDS
             , (uint8_t *)uuid16_list, sizeof(uuid16_list)); //
             UUID's broadcast in advertising packet
          ble.gap().setAdvertisingInterval(250); // 250ms.

          /* Add our custom service */
          ble.addService(customService);

          /* Start advertising */
          ble.gap().startAdvertising();
}


void chararrraytoUART(uint8_t *array, uint8_t length){
          if(pc.writeable())
                    for(int i=0; i<length; i++) {
                            pc.putc(*array);
                            array++;

                    }

}

/*
 *  Main loop
 */
int main(void)
{

          pc.baud(115200);
          pc.printf("IIC␣Demo␣Start␣\r\n");
          // For debugging into PC terminal
          /* initialize stuff */
//         printf("\n\r********* Starting Main Loop *********\n
   \r");
          Datapacket readings;
          setupI2C();
          setupacc(ACC_LEFT);
          setupacc(ACC_RIGHT);
          pc.printf("Setup␣complete␣\r\n");
          led = 1;
          Ticker ticker;
          ticker.attach(periodicCallback, 1); // blink LED every
              1 second
          BLE& ble = BLE::Instance(BLE::DEFAULT_INSTANCE);
          ble.init(bleInitComplete);
          uint16_t array[26];
          /* SpinWait for initialization to complete. This is
             necessary because the
           * BLE object is used in the main loop below. */
          while (ble.hasInitialized()  == false) { /* spin loop
             */ }
          /* Infinite loop waiting for BLE interrupt events */
          while (1) {
                    // check for trigger// from periodicCallback()
                    if (triggerSensorPolling) {
                            triggerSensorPolling = false;
//                        data[0] = data[0]+1;//
                            readflexs(&readings);
                            readacc(&readings, ACC_LEFT );
                            readacc(&readings, ACC_RIGHT );
                            datapackettoarray(readings, array);
                            pc.printf("Values␣read:␣\r\n");
                            chararrraytoUART((uint8_t*) &array,
                               52);
                            BLE::Instance(BLE::DEFAULT_INSTANCE).
                               gattServer().write(readChar.
                               getValueHandle(), (uint8_t*)array,
                               52 );

                    }
                    else {
                            ble.waitForEvent(); // low power wait
                               for event
                    }
          }
}
```

gloves.cpp:

```
#include "mbed.h"
#include "sensors.h"


DigitalOut led(LED1);

static volatile bool triggerSensorPolling = false;

Serial pc(USBTX, USBRX);


void datapackettoarray(Datapacket todump, uint16_t *dest)
{
        uint8_t j,k;

        for (j=0; j<12; j++)
        {
                *dest=todump.flex[j];
                dest++;
        }
        for(k=0; k<7; k++)
        {
                *dest=todump.acc[0][k];
                dest++;
        }

        for(k=0; k<7; j++, k++)
        {
                *dest=todump.acc[1][k];
                dest++;
        }



}

char hex_from_nibble(uint8_t nibble)
{
        nibble &= 0xf;
        if (nibble < 0xa)
                return '0' + nibble;
        else
                return 'a' + (nibble - 0xa);
}

void put_byte(uint8_t byte)
{
        pc.putc(hex_from_nibble(byte >> 4));
        pc.putc(hex_from_nibble(byte));
}
```

```
void periodicCallback(void)
{
        led = !led; /* Do blinky on LED1 while we're waiting for events */

        /* Note that the periodicCallback() executes in interrupt context, so it is safer
         * heavy-weight sensor polling from the main thread. */
        triggerSensorPolling = true;
}

void chararrraytoUART(uint8_t *array, uint8_t length){
        if(pc.writeable()) {
                pc.printf("\r\n");
                for(int i=0; i<length; i++) {
                        put_byte(*array);
                        array++;
                }
        }
}


/*
 *  Main loop
 */
int main(void)
{
        pc.baud(115200);
        pc.printf("IIC Demo Start \r\n");
        Datapacket readings;
        setupI2C();
        setupacc(ACC_LEFT);
        setupacc(ACC_RIGHT);
        pc.printf("Setup complete \r\n");
        led = 1;
        Ticker ticker;
        ticker.attach(periodicCallback, 0.02); // blink LED at 50Hz
        uint16_t array[26];
        while (1) {
                // check for trigger// from periodicCallback()
                if (triggerSensorPolling) {
                        triggerSensorPolling = false;
                        readflexs(&readings);
                        readacc(&readings, ACC_LEFT );
                        readacc(&readings, ACC_RIGHT );
                        datapackettoarray(readings, array);
                        chararrraytoUART((uint8_t*) &array, 52);
                        pc.printf("\n");

                }
        }
}
```

sensors.cpp:

```
#include "sensors.h"
#include "mbed.h"
#include "wire.h"


AnalogIn analogL(P0_4); // Analog senson input polling A3; Left In
AnalogIn analogR(P0_5); // Analog senson input polling A4; Right In

DigitalOut flexA(P0_7); // Both Hand Finger Selectors:  A
DigitalOut flexB(P0_6); // B
DigitalOut flexC(P0_15); //  C

I2C i2c(p29, p28);



void WriteBytes(uint8_t addr, uint8_t *pbuf, uint16_t length, uint8_t DEV_ADDR)
{

        i2c.start();
        i2c.write( DEV_ADDR <<1 );
        i2c.write( addr );
        for (int i =0; i<length; i++) {
                i2c.write( *pbuf );
                pbuf++;
        }
        i2c.stop();
}

void ReadBytes(uint8_t regaddr, uint8_t *pbuf, uint16_t length, uint8_t DEV_ADDR)
{
        i2c.start();
        char data[length];
        // i2c.write( (DEV_ADDR <<1) +1);
        i2c.write( (DEV_ADDR << 1));
        i2c.write( regaddr);
        i2c.start();
        i2c.read( (DEV_ADDR<<1), data, length);

        for(int i=0; i<length; i++) {    // equate the addresses
                pbuf[i]=data[i];
        }
}

void WriteByte(uint8_t addr, uint8_t buf, uint8_t DEV_ADDR)
{

        i2c.start();
        i2c.write( DEV_ADDR <<1 );
        i2c.write( addr );
        i2c.write( buf );
        i2c.stop();
}

void setupI2C(void){
        // Wire.begin(SCL, SDA, TWI_FREQUENCY_100K);
}

void setupacc(uint8_t ADDRESS){
        WriteByte(SMPRT_DIV, 0x13, ADDRESS); // Set rate divider to 19, so sample rate is
        WriteByte(CONFIG, 0x05, ADDRESS); // Set the digital low pass filter to 10Hz Cutof
        WriteByte(GYRO_CONFIG, 0x00, ADDRESS); // Set the gyro range to \pm 250 degrees/se
        WriteByte(ACCEL_CONFIG, 0x00, ADDRESS); // Set the acc range to \pm 2g
        WriteByte(FIFO_EN, 0x00, ADDRESS); // Disable FIFO
        WriteByte(I2C_MST_CTRL, 0x00, ADDRESS); // Disabling external I2C
        WriteByte(INT_PIN_CFG, 0x30, ADDRESS); // Active high, push-pull, high until cleare
        WriteByte(INT_ENABLE, 0x00, ADDRESS); // DataRDY is the last bit if needed
        WriteByte(USER_CTRL, 0x00, ADDRESS); // No FIFO or I2C Master set
        WriteByte(PWR_MGMT_1, 0x00, ADDRESS); // TODO: Sleepmode is here; Disabled now; Cy
        WriteByte(PWR_MGMT_2, 0x00, ADDRESS); // No lowpower mode, all sensors active

}

void readacc(Datapacket *data, uint8_t accadr){
        uint8_t rorl = 1; //
        if(accadr == ACC_RIGHT) {rorl=0; };
        uint8_t buffer[14]={0};
        uint16_t temp;
        ReadBytes(0x3B, buffer, 14, accadr);
        temp = ((buffer[0]<<8)+buffer[1]);
        for (int i=0; i<7; i++) {

                temp=0;
                temp = ((buffer[2*i]<<8)+buffer[2*i + 1]);
                data->acc[rorl][i]= (uint16_t) temp;
        }
        // Acc data structure: ACC X, Y, Z; Temperature measurement; GYRO X, Y, Z;

}




void readflexs(Datapacket *data){

        // Order of the flex sensors:
        // 0 - right thumb, 4 - right pinky, 5 - R elbow;
        // 6 - left thumb, 10 left pinky, 11 - L elbow;
        // Read right hand in:

        // AnalogIn analogL(P0_4); // Analog senson input polling A3; Left In
        // AnalogIn analogR(P0_5); // Analog senson input polling A4; Right In
        // Address is CBA
        // Read thumbs:
        flexC = 0;
        flexB = 0;
        flexA = 0;
        data->flex[6] = (uint16_t) (analogL.read()*65535);
        data->flex[0] = (uint16_t) (analogR.read()*65535);

        flexC = 0;
        flexB = 0;
        flexA = 1;
        data->flex[7] = (uint16_t) (analogL.read()*65535);
        data->flex[1] = (uint16_t) (analogR.read()*65535);

        flexC = 0;
        flexB = 1;
        flexA = 0;
        data->flex[8] = (uint16_t) (analogL.read()*65535);
        data->flex[2] = (uint16_t) (analogR.read()*65535);

        flexC = 0;
        flexB = 1;
        flexA = 1;
        data->flex[9] = (uint16_t) (analogL.read()*65535);
        data->flex[3] = (uint16_t) (analogR.read()*65535);

        flexC = 1;
        flexB = 0;
        flexA = 0;
        data->flex[10] = (uint16_t) (analogL.read()*65535);
        data->flex[4] = (uint16_t) (analogR.read()*65535);


        flexC = 1;
        flexB = 0;
        flexA = 1;
        data->flex[11] = (uint16_t) (analogL.read()*65535);
        data->flex[5] = (uint16_t) (analogR.read()*65535);


}
```

sensors.h:

```
#include "mbed.h"

#ifndef SENSORS_H
#define SENSORS_H


// Register names, can look up in the datasheet
```

```
#define SMPRT_DIV      0x19 //
#define CONFIG         0x1A //
#define GYRO_CONFIG    0x1B //
#define ACCEL_CONFIG   0x1C //
#define FIFO_EN        0x23 //
#define I2C_MST_CTRL   0x24 //
#define INT_PIN_CFG    0x37 //
#define INT_ENABLE     0x38 //

#define ACCX158        0x3B //
#define ACCX70         0x3C //
#define ACCY158        0x3D //
#define ACCY70         0x3E //
#define ACCZ158        0x3F //
#define ACCZ70         0x40 //
#define TEMP158        0x41 //
#define TEMP70         0x42 // Temperature in C = ((signed int16_t) value)/340
+ 36.53
#define GYROX158       0x43 //
#define GYROX70        0x44 //
#define GYROY158       0x45 //
#define GYROY70        0x46 //
#define GYROZ158       0x47 //
#define GYROZ70        0x48 //

#define USER_CTRL      0x6A //
#define PWR_MGMT_1     0x6B //
#define PWR_MGMT_2     0x6C //
#define WHO_AM_I       0x75 // Check device number to see if it works

const uint8_t ACC_LEFT  = 0x69; //     0b01101000
const uint8_t ACC_RIGHT = 0x68; //     0b01101001


typedef struct {

        uint16_t flex[12];
        // Order of the flex sensors:
        // 0 - right thumb, 4 - right pinky, 5 - R elbow;
        // 6 - left thumb, 10 left pinky, 11 - L elbow;
        uint16_t acc[2][7];
        // 7th value is the temperature value
        uint16_t temp;


} Datapacket;

typedef struct {

        uint8_t pixels[8];
} Screen;

void readacc(Datapacket data);

void setupacc(uint8_t ADDRESS);

void WriteBytes(uint8_t addr, uint8_t *pbuf, uint16_t length, uint8_t DEV_ADDR);

void WriteByte(uint8_t addr, uint8_t buf, uint8_t DEV_ADDR);

void ReadBytes(uint8_t addr, uint8_t *pbuf, uint16_t length, uint8_t DEV_ADDR);

void readflexs(Datapacket *data);

void readacc(Datapacket *data, uint8_t accadr);

void setupI2C(void);




#endif
```

9 Appendix B: Pictures

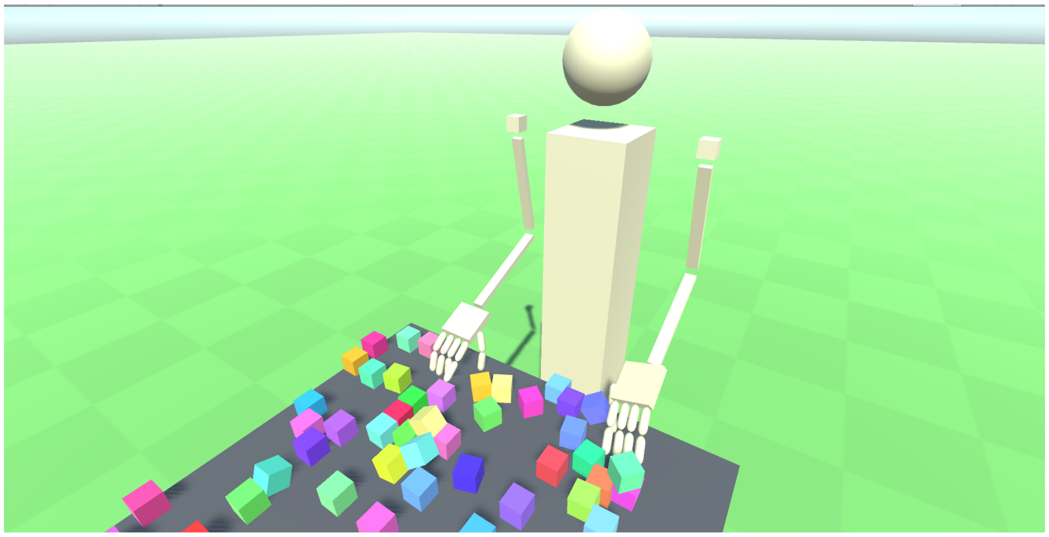


Figure 8: Initialisation of the simulation

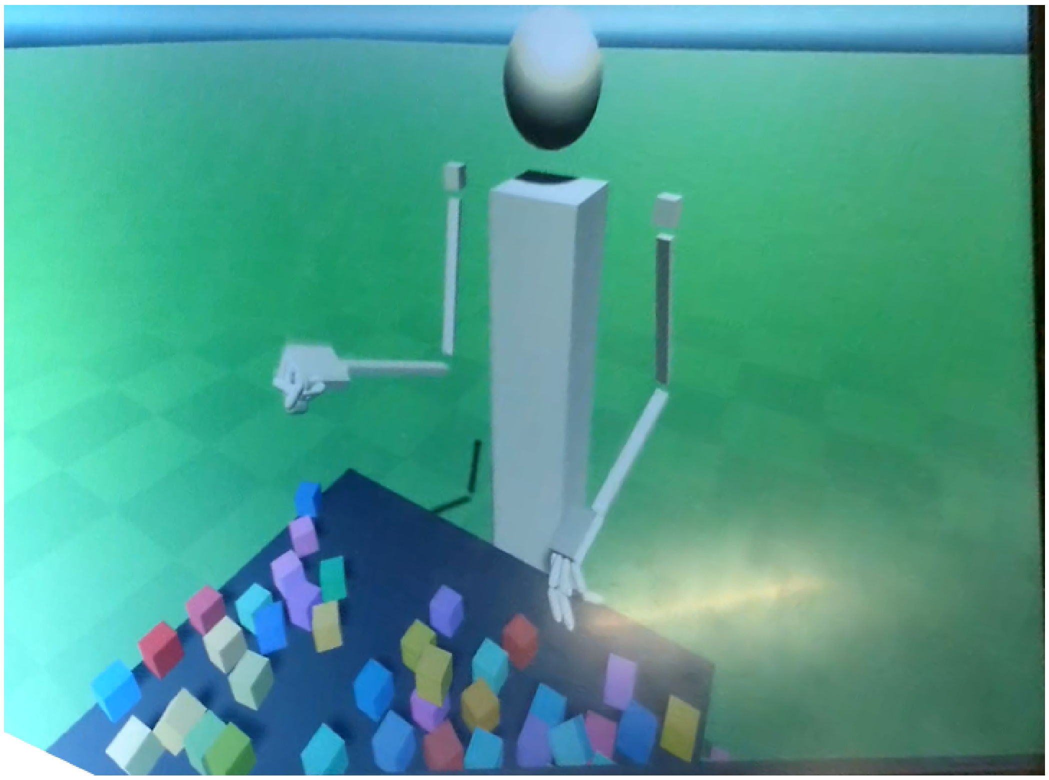


Figure 9: Simulation in action

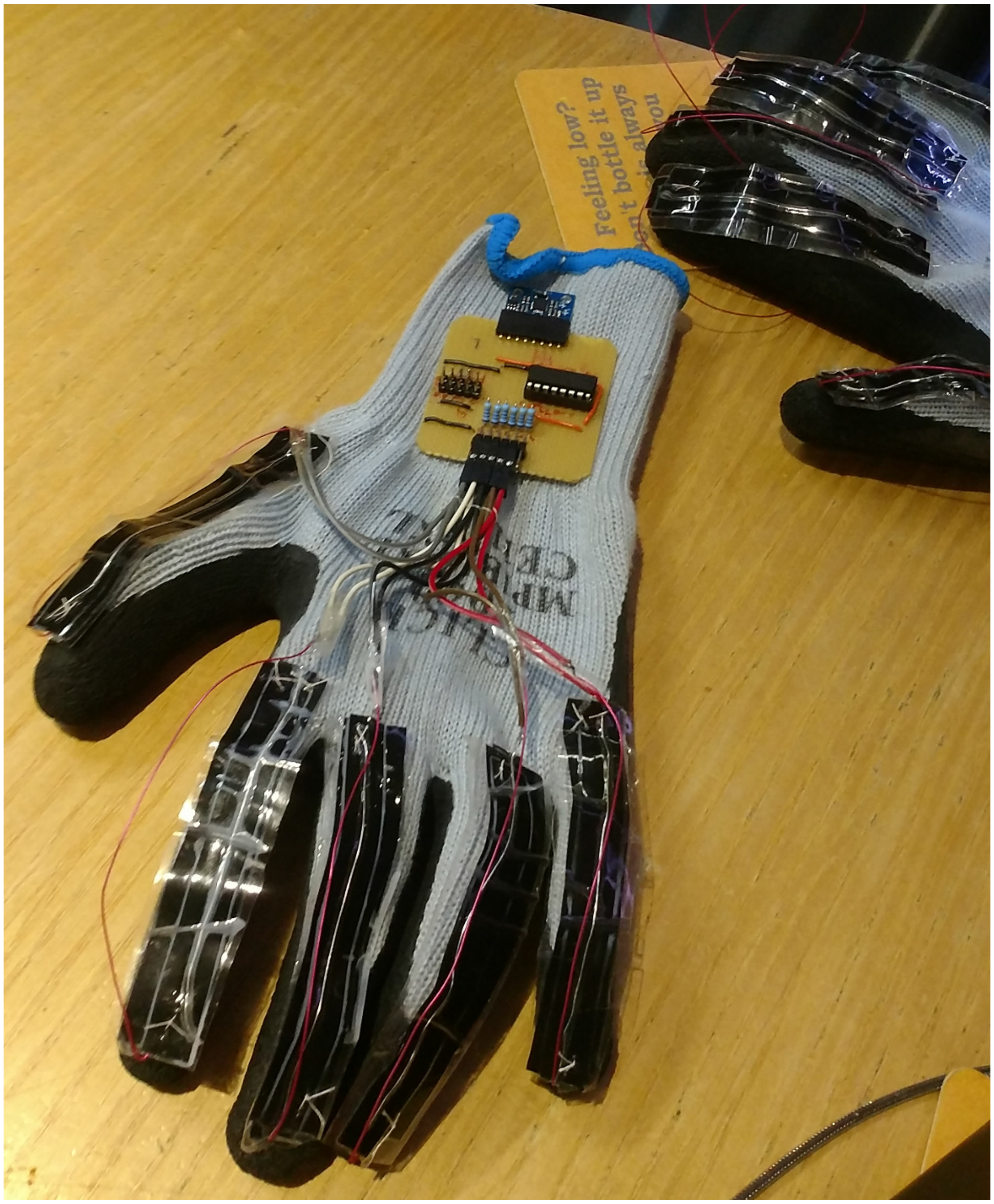


Figure 10: Finalised glove, ready to be connected.

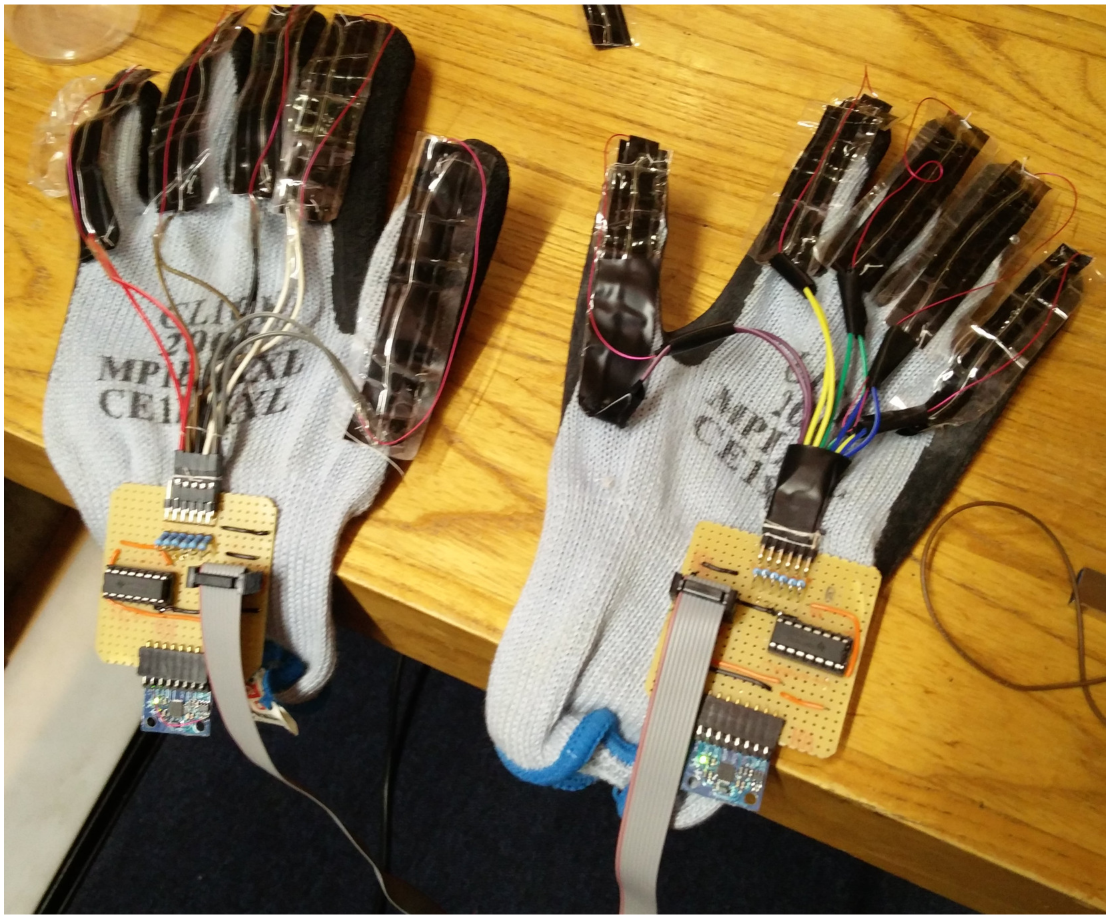


Figure 11: Both gloves connected to the main board.

10 Appendix C: How to Make a Cheap Flex Sensor

You will need:

- Exposed wire.
- Sticky tape or duct tape.
- Conductive and strain-sensitive plastic called Velostat.
- $\approx 30 minutes$ for each of the first few sensors.

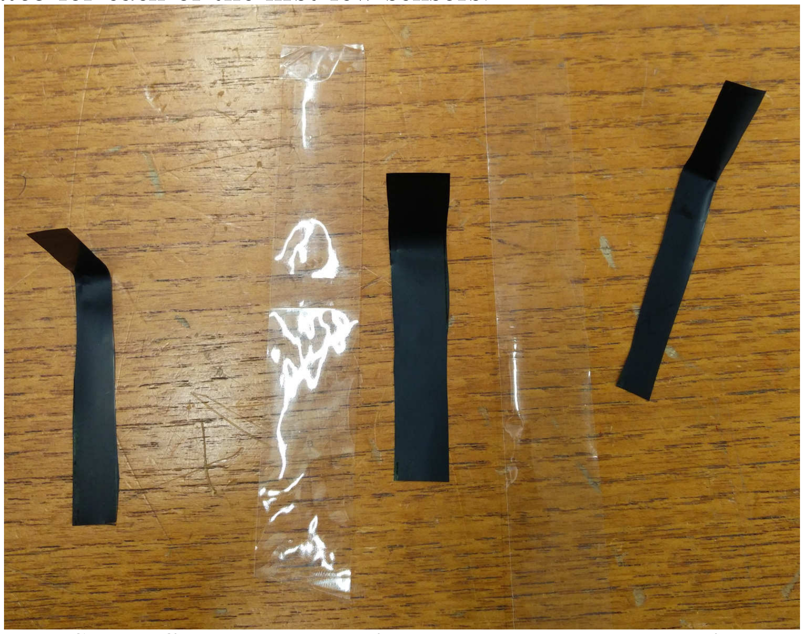


Figure 12: Start off with 3 strips of Velostat and two strips of sticky tape.

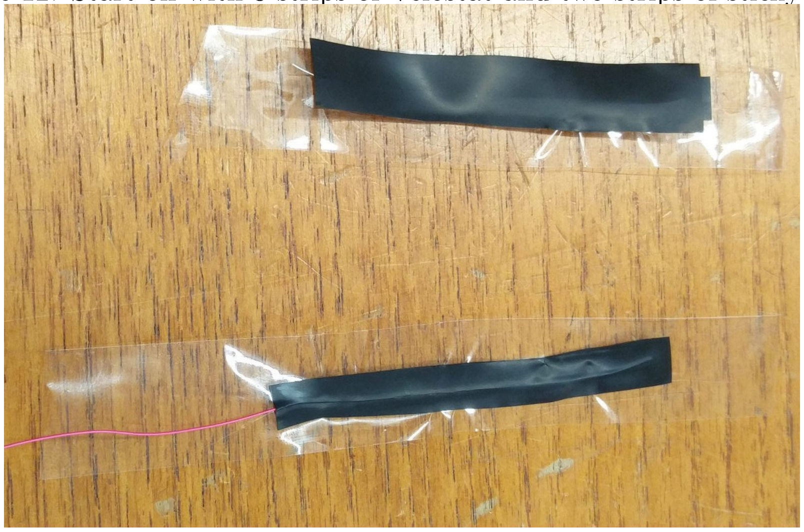


Figure 13: Make a sandwich: $tape- > wire- > velostat- > velostat- > velostat- > wire- > tape$. The wires have to be exposed.

IIA Project SB3 First Report

By Aidas Liaudanskas (al747) & Luke Wren (lw509)

1. **Project Description**

Our team is building a multi-functional device - British Sign Language (BSL) translator and a 3D real time bomb defusion simulator. Such an unusual choice was made because we could not decide on a single thing and both applications require identical circuitry. To make it even a bigger challenge we have decided to use Red Bear Labs BLE Nano microcontroller. It is basically the same as the Arduino provided except that it has an inbuilt Bluetooth Low Energy (BLE) capability. The device will consist of two gloves with 5 flex sensors and an accelerometer on each of them. Such sensor choice will allow us to determine the hand orientation and finger positions. We will have an additional flex sensor on the elbow as well.

2. **Analog Circuit Design**

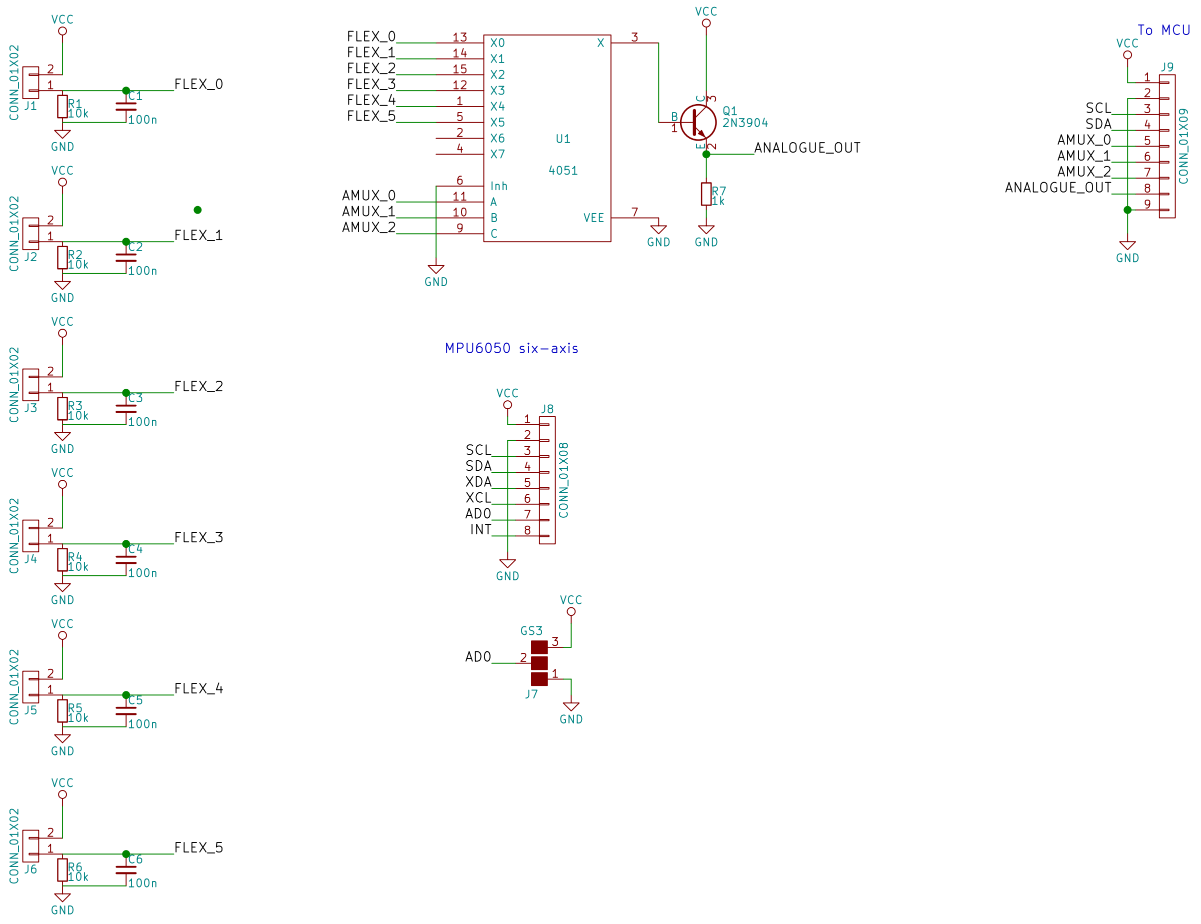


Figure 1: Schematic for the analog circuitry (1 hand/glove)

3. **Work Allocation**

3.1. Bluetooth communication interface - Aidas and Luke
3.2. Sensor and microcontroller communications - Aidas
3.3. Flex sensors - Aidas
3.4. 3D simulation in Unity - Luke (some pictures are shown in the appendix A)

4. **Parts Purchased**

Item Code	Item Description	Amount	Net Cost, GBP
CD4051BE	8 Channel Analog MUX	2	0.61
-	Pressure-Sensitive Conductive Sheet	1	5.39
GY-521 MPU-6050	3-axis Acellerometer module	2	6.50

5. **Communications Block Diagram**

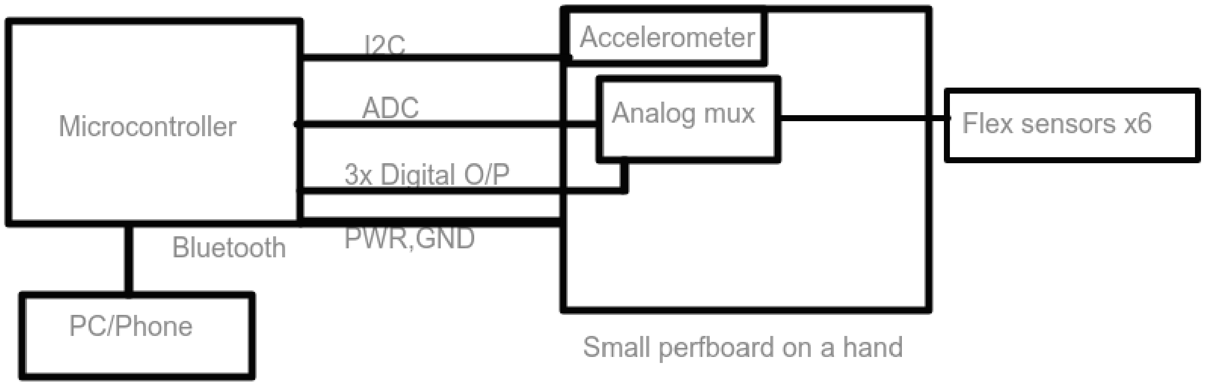


Figure 2: Block diagram for the communications between MCU/PC/Sensors

6. **Additional Comments**

6.1. 3D Simulation will be developed in Unity environment using C#
6.2. Microcontroller code will be developed in ARM mbed environment using C++
6.3. Sign language recognition and signal processing will be done in Python
6.4. Such software language choice might require quite complicated interprocess communication on the PC/phone side, so might make some adjustments here
6.5. Flex sensors cost 6GBP each, so we decided to make our own using Velostat (purchased item No. 2) . They perform almost as well at a fraction of the cost

Appendix A: Figures

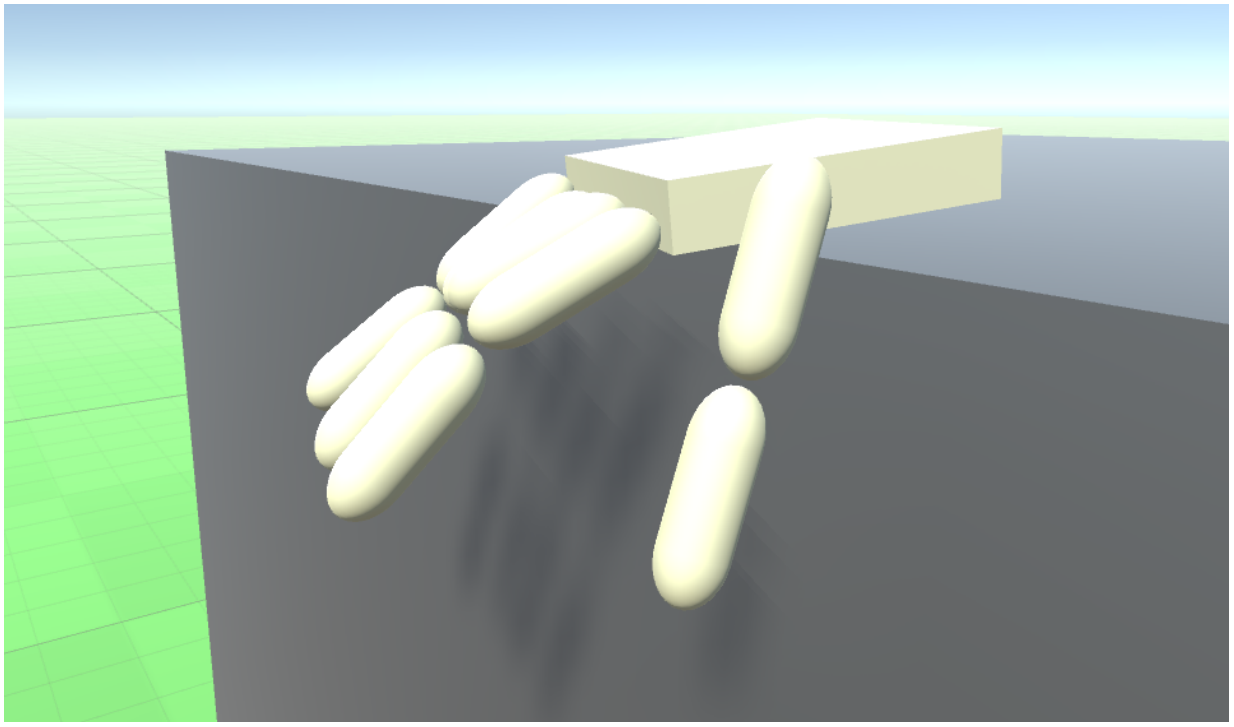


Figure 3: 3D hand in an open position

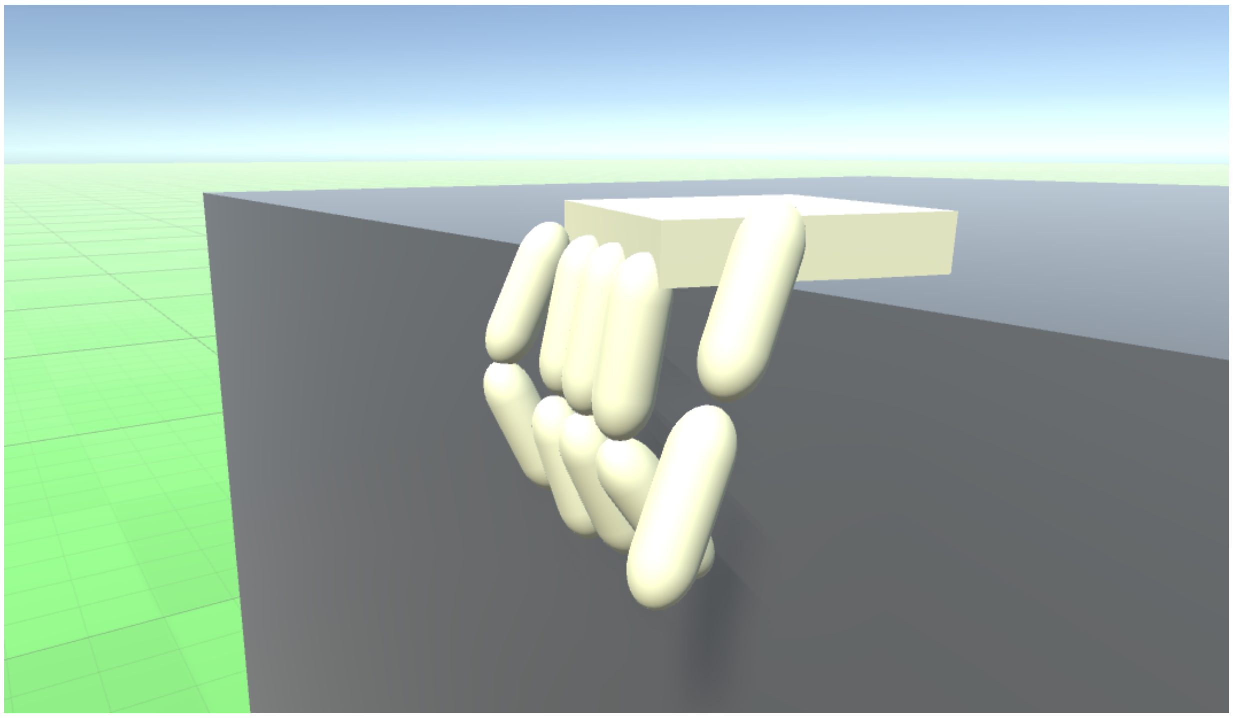


Figure 4: 3D hand gripping an object in a closed position

Smart Glove

Motion Capture and Gesture Recognition System

Applications
Telepresence, Conference Calling, Gesture-Based User Interfaces

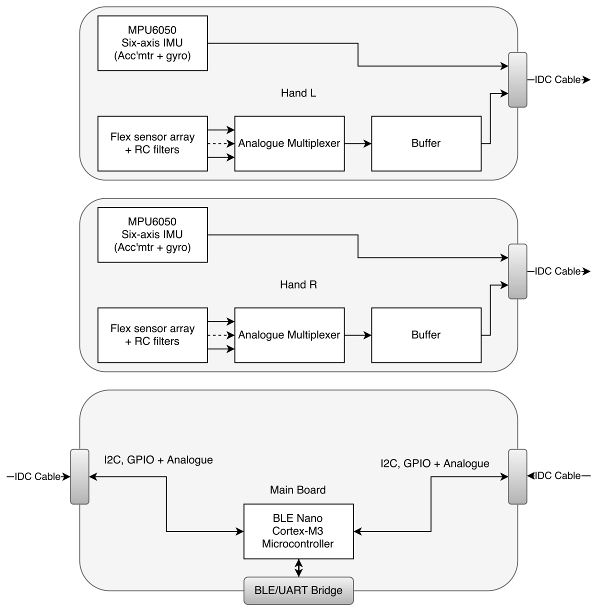
Telepresence: A single skilled worker can perform manual tasks at multiple job sites from afar
Conference calling: Add natural body language and ”hand gestures” to virtual conference environments
Gesture based input: Whether for cutting-edge CAD or the latest VR games

Smart Glove

The Smart Glove system is a fully integrated motion capture and gesture recognition solution.

- Independent finger position measurement using our own in-house flex sensing technology
- 6-axis IMUs on each hand to provide low-latency motion input
- Advanced two-way communications protocol to guarantee fresh, responsive, low-latency interactivity
- Uses powerful digital signals processing techniques to maximise your data quality
- 50Hz update rate
- 38ms end-to-end latency
- Total power consumption under 100mA at 5V

System Architecture

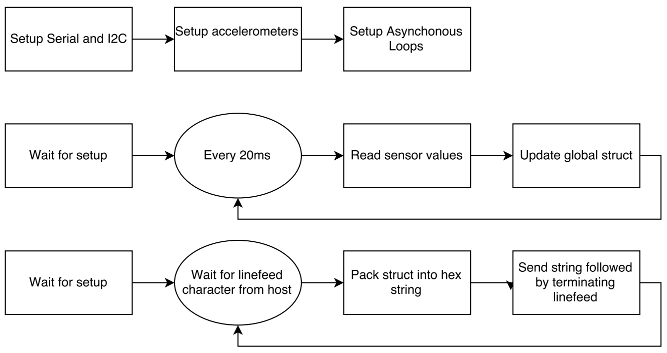


The electronic portion of the Smart Glove system is described above.

Two independent glove-sensor-units (GSUs) are controlled by a single, low-power, proprietary microcontroller host board.

In spite of its expansive functionality and cutting-edge technology, this system is relatively inexpensive, with a BOM cost just of just below £14.

Communicating with the Smart Glove

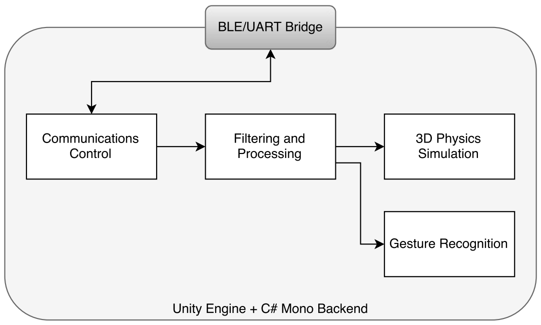


As described in the software flowchart, the Smart Glove host board updates its sensor values at a polling rate of 50Hz. The IMUs refresh at a much higher rate of 1 kHz to minimise the overall latency of the system. Host applications on a PC can then request the most recent sensor values from the host board in an asynchronous manner.

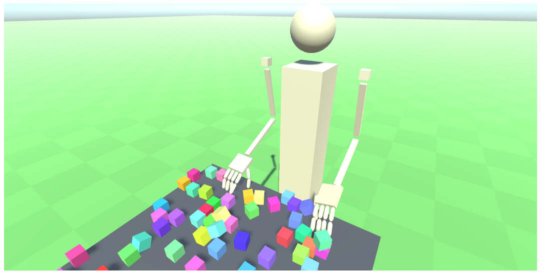
The sample application communicates using a standard TTL-level UART at 115200 baud. However, the host board is Bluetooth Low-Energy enabled, and can make the sensor data available to a BLE Central Device host using a custom GATT characteristic.

Software Examples

We expect users will build their own software appropriate to their needs; however, we provide a sample application which demonstrates the functionality available to Smart Glove customers, as well as containing our proprietary signals-processing code, which is made freely available to our users.



The above figure describes the structure of this program. Full source code listing is available on our website.



The sample application is cross-platform in nature, and will run on any of over 25 platforms, including, but not limited to:

- iOS
- Android
- Windows
- Linux
- MacOS
- Google Daydream VR

Future Additions to Smart Glove Family

The MPU6050 6-axis IMUs used on the GSUs have the facility to be expanded with a magnetometer, attached to an auxiliary I2C bus. The result is a *full 50% increase* in the number of axes. Wren-Liaudanskas Electronics Corp. is fully committed to providing the most cutting-edge technology to our customers.

As well as adding yaw measurement to the Smart Glove's sensorium, this feature will enable the addition our proprietary ElbowSense technology, allowing full inverse-kinematic reconstruction of your upper and lower arm motion, for truly three-dimensional interaction.

The additional analogue channel required for this additional flex sensor is already present on the GSU board, meaning there is plenty of room for future expansion. Our customers can buy today, safe in the knowledge they are prepared for the technology of tomorrow.

General Marketing Buzzwords

- Integrated efficiency maximisation technology
- Sensor utilisation optimiser
  - Complementary deoptimiser in case of overutilisation
- Synergetic management system
- Bluetooth Low-Energy
- Internet of Things
- Wearable Technology

Sales Contacts

If you have further queries about the functionality offered by Smart Glove

Luke Wren
Head of Marketing
lw509@cam.ac.uk

If you are interested in purchasing Smart Glove for your business

Aidas Liaudanskas
Head of Sales
al747@cam.ac.uk