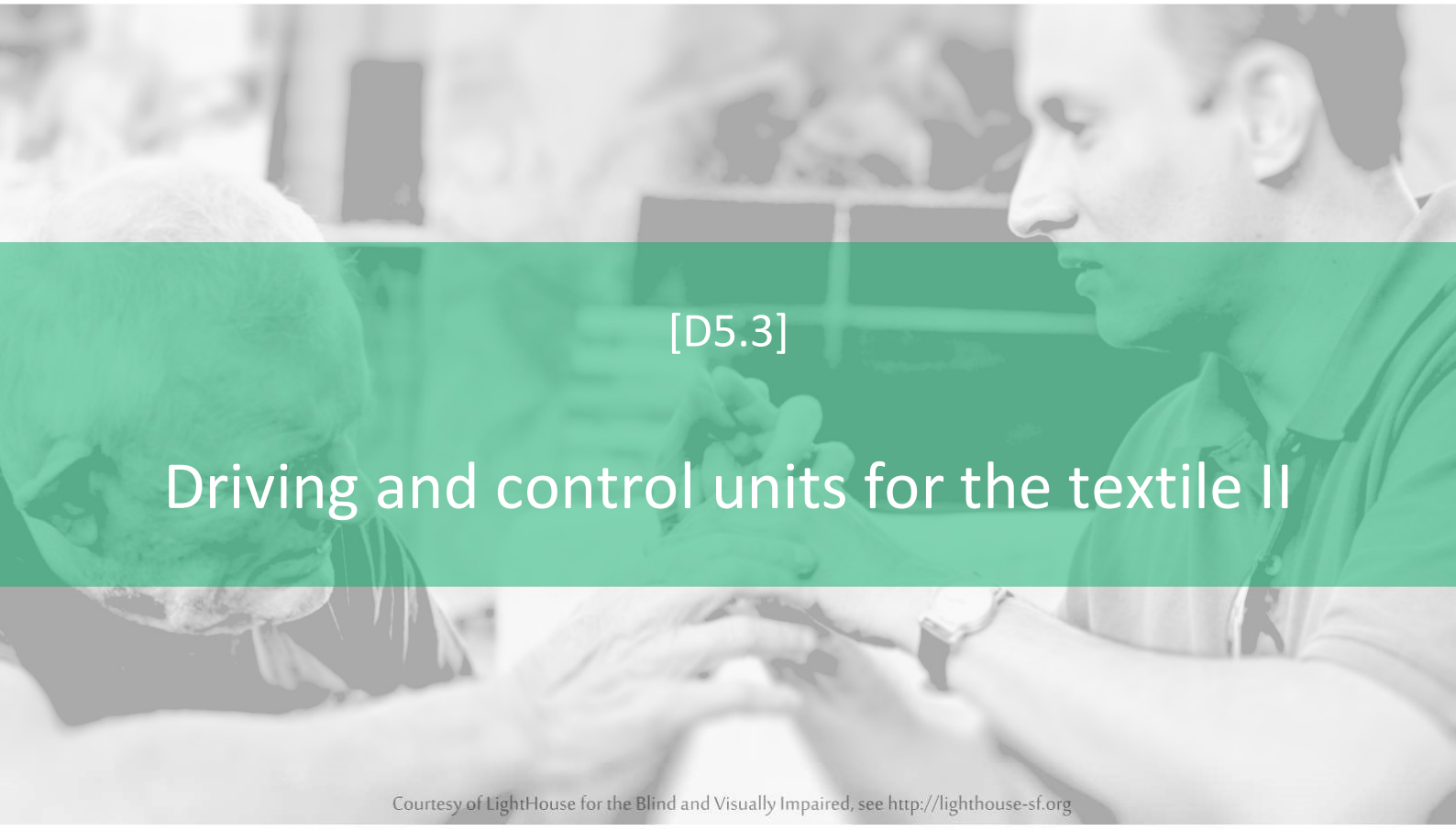
SUITCEYES

1 Jan 2018 - 31 Dec 2020

Smart, User-friendly, Interactive, Tactual, Cognition-Enhancer, Yielding Extended Sensosphere

Appropriating sensor technologies, machine learning, gamification and smart haptic interfaces



Courtesy of LightHouse for the Blind and Visually Impaired, see http://lighthouse-sf.org

[D5.3]

Driving and control units for the textile II

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780814.

Dissemination level		
PU	PUBLIC, fully open, e.g. web	X
CO	CONFIDENTIAL, restricted under conditions set out in Model Grant Agreement	
CI	CLASSIFIED, information as referred to in Commission Decision 2001/844/EC.	

Deliverable Type		
R	Document, report (excluding the periodic and final reports)	
DEM	Demonstrator, pilot, prototype, plan designs	X
DEC	Websites, patents filing, press & media actions, videos, etc.	
OTHER	Software, technical diagram, etc.	

Deliverable Details	
Deliverable number	5.3
Part of WP	5
Lead organisation	UNIVLEEDS
Lead member	Raymond Holt

Revision History			
V#	**Date**	**Description / Reason of change**	**Author / Org.**
v01	-	Structure proposal	-
v02	2018-10-11	First draft for internal review	Raymond Holt / UNIVLEEDS
V03	2018-10-19	Second draft for PMB review	Raymond Holt / UNIVLEEDS
V04	2018-10-26	Final draft	Raymond Holt / UNIVLEEDS

Authors	
Partner	**Name(s)**
UNIVLEEDS	Raymond Holt

Contributors		
Partner	**Contribution type**	**Name**
HB	Reviewer / Li Guo	
HB	Reviewer/ Nils-Krister Persson	
CERTH	Reviewer/ Panagiotis Petrantonakis	

Glossary	
Abbr./ Acronym	**Meaning**
SUITCEYES	Smart, Userfriendly, Interactive, Tactual, Cognition-Enhancer that Yields Extended Sensosphere
Modality	Examples hereof are vibration, heat etc.
HIPI	Haptic intelligent personalized interface – the goal of SUITCEYES and built as a textile structure.
Actuator	An actuator is a component of a machine that is responsible for moving and controlling a mechanism or system.
USB	Universal Serial Bus, a port in a computer for the transmitting of data and/or electricity.
Raspberry Pi	A tiny computer made for teaching computer science. Widely used in development projects.
Windows PC	A computer using the Microsoft windows operating systen
H-Bridge	An electronic circuit that can change the direction of an input signal (reversing positive and negative) in response to a control signal so that an actuator (such as a motor) can be driven in different directions.
Boolean direction signal	A digital signal to an H-bridge telling it whether to reverse the direction of the input signal based. Typically, the output is forward if the direction signal is “high” and reversed if the direction signal is “low”.
Breadboard	A solderless construction used for prototyping electronics and circuit design
Monodirectional Actuator	An actuator whose direction cannot be reversed – a solenoid or vibration motor, for example.
Bidirectional Actuator	An actuator whose direction can be reversed – a peltier module, DC motor or linear actuator.
Controller Personality	Different behaviours that can be uploaded to a controller, according to the type of actuator and behavior required. Three are defined at

	present: Monodirectional PWM, Monodirectional Binary, Bidirectional PWM.
Monodirectional PWM	A controller personality that allows an actuator to be driven with varying intensities in a single direction - intended for use with vibration motors, where the intensity of vibration can be varied.
Bidirectional PWM	A controller personality that allows an actuator to be driven with varying intensities in two different directions - intended for use with peltier modules, DC motors and linear actuators.
Monodirectional Binary	A controller personality that allows an actuator to be turned on or off, with no variation in intensity. This enables a larger number of channels to be used on the same Arduino.
Central Processor	The computer that forms the core of the HIPI, co-ordinating the sensors and actuators.
PWM	Pulse Width Modulation – a method for varying the amount of power delivered over time from a digital output. While a digital output can only be “on” or “off”, by rapidly cycling it between its “on” and “off” states, and adjusting the proportion of time for which it is on (the duty cycle), fine adjustments can be made to the power delivered to actuators and so control the speed of motors, rate of heating of a peltier module, brightness of an LED, etc.

Table of Contents

Executive Summary

The second iteration of the Controller and Driver for the early version of the textile communicative interface is developed and briefly described. Controller "personalities" for different modes of interaction have been established and designed to be compatible with an event driven-architecture for ease of interfacing with the sensor systems being developed and WP4. At this stage, thermal actuators, solenoids and vibration motors have been used. Verification was done in Leeds in July 2018 with vibrotactile stimuli and initial psychophysical measurements. This provides a flexible controller scheme that can be implemented for testing with a variety of tactile stimuli, ready for interfacing with the wider HIPI system.

DEMO description

The purpose of this document is to describe the second iteration of Controller for Deliverable 5.3, which has been tested in Leeds in vibrotactile form in July 2018, and subsequently extended to the use of solenoids.

Controller Goals

The second iteration of controller has been designed to expand upon the first generation of controller developed in Deliverable 5.2, with the following goals:

1) Reading a signal from a host computer comprising a sequence of frames, of fixed duration, each consisting of an **intensity** as a percentage of the maximum intensity for a given actuator (from 0 to 100%), such that no feedback from the controller is required;
2) Driving an actuator with the requested intensity for the duration of each frame; and
3) Allowing both bidirectional (motors, peltier modules) and monodirectional (vibration motors, solenoids) actuators to be driven.
4) Increasing the number of channels that can be used per controller.

Controller Configuration

The second iteration of controller has been developed to accommodate the developing architecture of the wider HIPI, where haptic signals must be generated in response to sensor data, as illustrated in Figure 1. The proposal is that the HIPI will use an event-driven architecture, where Sensors raise Events that are passed to the Central Processor. The Central Processor will then select an appropriate haptic signal to inform the user of the event. Rather than displaying a single signal comprising a duration and intensity at a given time, the second iteration controller has been developed so that it receives "frames" at a predefined frequency from the Central Processor. Each frame comprises a set of intensities – one for each haptic channel being used by the controller. The number of channels available depends upon the type of controller and type of controller personality being used: these are described in more detail below. Each frame is sent to the controller over a Serial connection. The controller listens for the next frame, reads the frame when it arrives and displays the requested intensities on the relevant channels until the next frame is received and read. In this way, the Central Processor will be able to control the rate at which signals are sent to multiple controllers, and the controllers are agnostic to where they are receiving signals from. In addition, to reduce demand on the Central Processor, the flow of information is one-way: the Central Processor tells the controller what to display, but the controller does not give any information back to the Central Processor.

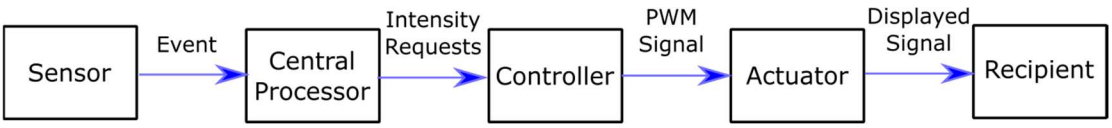


Figure 1: Block Diagram of Controller Configuration

For the purpose of this deliverable, the role of the Central Processor is again taken by a Host Computer that feeds the haptic signal requests to the controller in the defined format.

Controller Design

The hardware for the second iteration of controller is shown in Figure 2 – like the first iteration, this is based around an Arduino Nano and a Digilent PMod3 H-Bridge and assembled on a solderless breadboard. The H-bridge takes in a Boolean direction signal (High for one direction, Low for the other) and a PWM signal, and gives out the same PWM signal in the requested direction, offering the opportunity to drive either bidirectional or monodirectional displays, and to power them from sources other than the Arduino so that more power can be provided. However, there are the following differences:

1) The controller is now multi-channel, and the user can specify the number of channels required when uploading to the controller;
2) The controller no longer takes in a duration signal, only a sequence of intensities;
3) To simplify the code and wiring rather than attempting to delineate whether a given actuator is bidirectional or monodirectional in hardware, three different versions of the controller code (referred to as "controller personalities") have been developed.

The number of channels available depends upon the type of controller and the controller personality being used. The two PWM personalities (Monodirectional PWM and Bidirectional PWM) correspond to the two functions of the original controller, and only PWM-enabled pins on the Arduino can be controlled – this allows six channels on an Arduino Nano, but could be as many as 13 on an Arduino Mega. The third personality, Monodirectional Binary, only allows two intensities: 0 and 100% (any requested intensity above zero will be treated as 100%). However, this personality can use all the available digital pins on the controller, except those used for serial communication. This would allow 12 channels to be used on the Arduino Nano, or 52 on the Arduino Mega. The downside to this is that it does make it possible to misconfigure the controller – a signal intended to be for one of the PWM personalities will behave differently if sent to the Binary personality, and a six-channel signal will behave differently if sent to a three-channel signal. Accordingly, a handshake process has been introduced, in which the controller reports its type and number of channels when first connected. In this way, problems can be identified when the controller is first connected. The Arduino Code for these personalities can be found in Appendix 1.

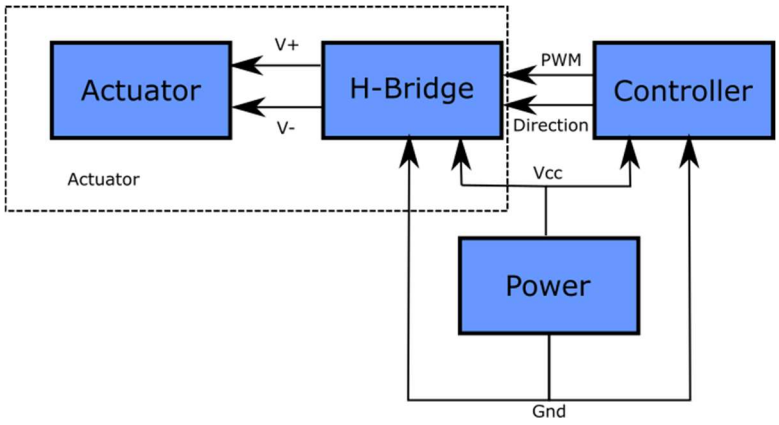


Figure 2: Block Diagram of Controller Hardware

Photographs of the physical circuit configured to display vibration signals through a vibration motor and a solenoid are shown in Figure 3.

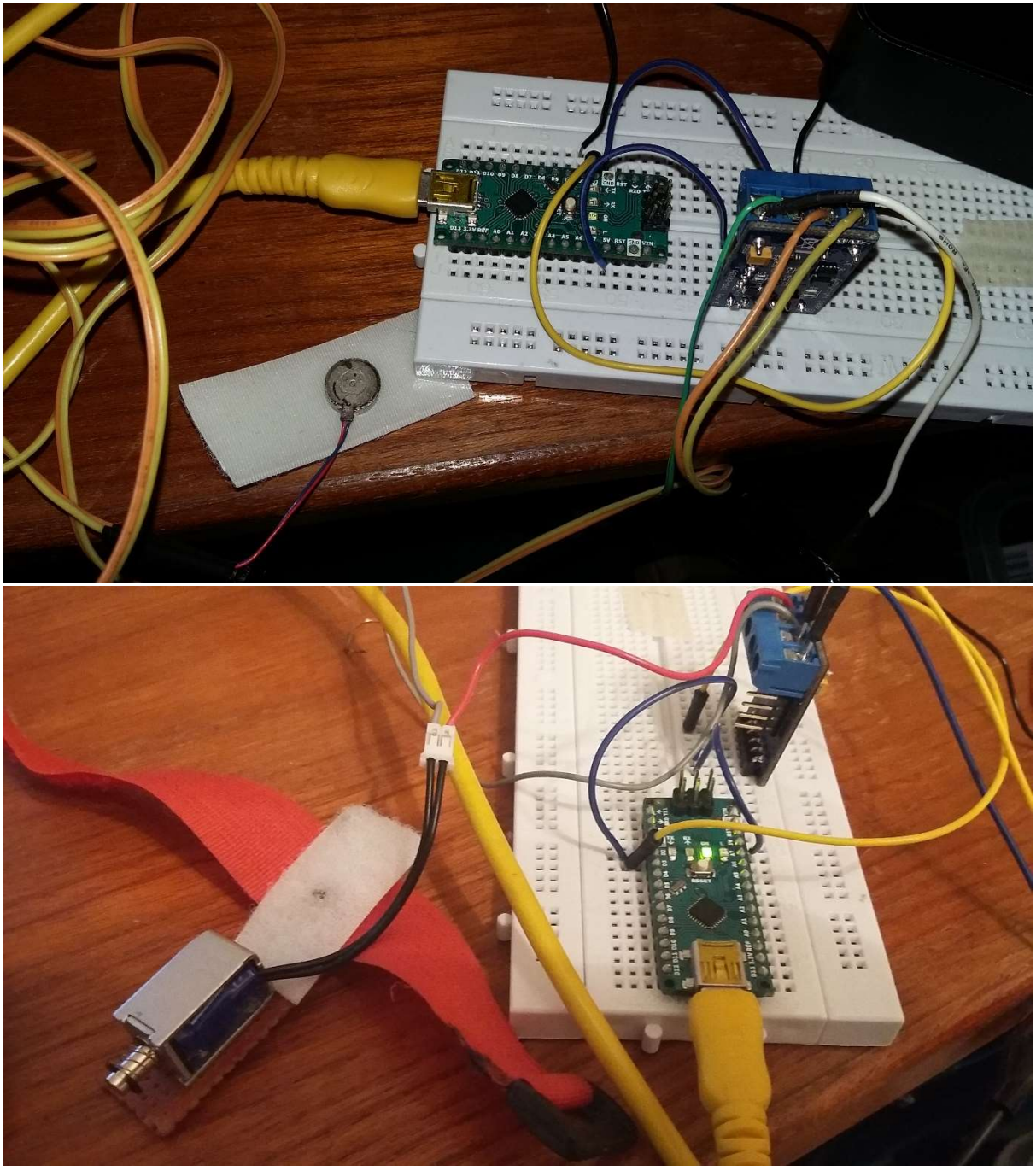


Figure 3: Physical Circuit Set up to display signals via Vibration Motor (top) and Solenoid (bottom).

For testing purposes, a Haptic Signal generator was written in Python 2.7, sending intensity requests to the controller. The speed at which requests are sent (and therefore the duration of each “frame”) was set in the Haptic Signal Generator code (included in Appendix 1). Each frame requests an intensity for each channel, as a percentage of maximum. For the monodirectional PWM controller, this is scaled such that 100 is 100% Duty Cycle, and 0 is 0%; while the Bidirectional was scaled in the same way as the first iteration of the controller (where a requested intensity of 0 is 100% duty cycle in the reverse direction, an intensity of 50 is a duty cycle of 0

(no output), and a requested intensity of 100 is a duty cycle of 100% in the forward direction). For the binary controller, a requested intensity of 0 is taken as no output, and any other value is taken to be a HIGH output from the associated channel.

Conclusions

A second iteration of the Controller and Driver for the early version of the textile communicative interface has been developed. This has been demonstrated successfully with peltier modules, vibration motors and solenoids. This has also been developed to be compatible with the wider software and hardware architecture being developed for the HIPI.

Appendix 1: Controller Code

This Appendix contains the Arduino sketches and Python scripts used for the controller personalities and Haptic Signal Generator.

Monodirectional PWM Personality Arduino Sketch

```
//define pin setup - currently set for 3 channels in this example.
int PinSetUp[] = {3, 5, 6}; //An array to store output pins for channels -
const int PinCount = (sizeof(PinSetUp)/sizeof(int)); //Identify number of pins from PinSetUp Array

//declare other variables
int Bytes[PinCount] = {}; //An array to store bytes sent to each channel - automatically set to be the same size as the number of pins declared in PinSetUp using the PinCount variable.

//define functions
int GetBytes() { //function to get Bytes - included as a separate function so that it can easily be adjusted for I2C communication.
  int ByteRead = Serial.parseInt();
  return ByteRead;
}

int CalculateOutputBytes(int BytetoProcess) {
  int BytetoReturn = BytetoProcess*2.55;
  return BytetoReturn;
}

void setup () {//initial setup
//set PinModes
for (int Pin = 0; Pin < PinCount; Pin++) {
    pinMode(PinSetUp[Pin], OUTPUT); //Iterate through each declared pin, and set it as an Output.
    }

//Begin Serial Communication
Serial.begin(9600);
Serial.println("1D PWM"); //tell the Central Unit which personality is being used.
Serial.println(PinCount); //tell Central Unit how many channels are available.
}

void loop () {
```

```
  //main loop
    while (Serial.available()>=PinCount){ //if there are as many or more bytes waiting as there are channels, pick up the next set.
      for (int Channel = 0; Channel < PinCount; Channel++) {//iterate through each channel in turn and collect communicated data
        Bytes[Channel] = GetBytes(); //assign next integer to next channel
      }
      for (int Channel = 0; Channel < PinCount; Channel++) {//iterate through each channel in turn, displaying the requested value
        analogWrite(PinSetUp[Channel],CalculateOutputBytes(Bytes[Channel])); //write requested intensity to output
      }
    }
}
```

Monodirectional Binary Personality Arduino Sketch

```
//define pin setup - currently set for 3 channels.
int PinSetUp[] = {3,5,6}; //An array to store output pins for channels
const int PinCount = (sizeof(PinSetUp)/sizeof(int)); //Identify number of pins from PinSetUp Array

//declare other variables
int Bytes[PinCount] = {}; //An array to store bytes sent to each channel - automatically set to be the same size as the number of pins declared in PinSetUp using the PinCount variable.

//define functions
int GetBytes() { //function to get Bytes - included as a separate function so that it can easily be adjusted for I2C communication.
  int ByteRead = Serial.parseInt();
  return ByteRead;
}

void setup () {//initial setup
//set PinModes
for (int Pin = 0; Pin < PinCount; Pin++) {
    pinMode(PinSetUp[Pin], OUTPUT); //Iterate through each declared pin, and set it as an Output.
    }

//Begin Serial Communication
Serial.begin(9600);
Serial.println("1D Binary");
Serial.println(PinCount); //tell Central Unit how many channels there are
}

void loop () {
  //main loop
    while (Serial.available()>=PinCount){ //if there are as many or more bytes waiting as there are channels, pick up the next set.
      for (int Channel = 0; Channel < PinCount; Channel++) {//iterate through each channel in turn and collect communicated data
        Bytes[Channel] = GetBytes(); //assign next integer to next channel
```

```
    }
    for (int Channel = 0; Channel < PinCount; Channel++) {//iterate through each channel in turn,
displaying the requested value
      if (Bytes[Channel] == 0) { //if the intensity requested is zero, send a LOW signal.
      digitalWrite(PinSetUp[Channel],LOW);
      }
      else {//if the intensity requested above zero, send a HIGH signal.
      digitalWrite(PinSetUp[Channel],HIGH);}
      }
    }
  }
```

Bidirectional PWM Personality Arduino Sketch

```
//define functions
int GetBytes() { //function to get Bytes - included as a separate function so that it can easily be
adjusted for I2C communication.
 int ByteRead = Serial.parseInt();
 return ByteRead;
}

int CalculateOutputBytes(int BytetoProcess) {
 int BytetoReturn = (5.1*BytetoProcess)-255;
 return BytetoReturn;
}


int CalculateOutputDirection(int ValuetoProcess) {
 int ValuetoReturn;
 if (ValuetoProcess<=0) {
  ValuetoReturn = 0;
 }
 else {
  ValuetoReturn = 255;//arbitrary above zero valye so that when used in DigtalWrite command, pin
displays HIGH
 }
 return ValuetoReturn;
}

//define pin setup - currently set for 1 channel.
int PinSetUp[] = {3}; //An array to store output pins for channels.
int DirPinSetUp[] = {2}; //An array to store output pins for Direction Control of H-Bridge.
const int PinCount = (sizeof(PinSetUp))/(sizeof(int)); //Identify number of pins from PinSetUp Array
const int DirPinCount = (sizeof(DirPinSetUp)/(sizeof(int)));//Identify number of pins from
DirPinSetUpArray

//declare other variables
int Bytes[PinCount] = {}; //An array to store bytes sent to each channel - automatically set to be the
same size as the number of pins declared in PinSetUp using the PinCount variable.
int OutputBytes[PinCount] = {}; //An array to store PWM output calculated for each channel -
automatically set to be the same size as the number of pins declared in PinSetUp using the PinCount
variable.
```

```
int OutputDirection[PinCount] = {}; //An array to store PWM output calculated for each channel - automatically set to be the same size as the number of pins declared in PinSetUp using the PinCount variable.

void setup () {//initial setup
//set PinModes

for (int Pin = 0; Pin < PinCount; Pin++) {
   pinMode(PinSetUp[Pin], OUTPUT); //Iterate through each declared pin, and set it as an Output.
   pinMode(DirPinSetUp[Pin], OUTPUT);
   }

//Begin Serial Communication
Serial.begin(9600);
Serial.println("2D PWM");
Serial.println(PinCount);
}

void loop () {
  //main loop
   while (Serial.available()>=PinCount){ //if there are as many or more bytes waiting as there are channels, pick up the next set.
      for (int Channel = 0; Channel < PinCount; Channel++) {//iterate through each channel in turn and collect communicated data
        Bytes[Channel] = GetBytes(); //assign next integer to next channel

for (int Channel = 0; Channel < PinCount; Channel++) {//iterate through each channel in turn and calculate the value to explain communicated data
        OutputBytes[Channel] = CalculateOutputBytes(Bytes[Channel]); //assign next calculated PWM value to next channel
        OutputDirection[Channel] = CalculateOutputDirection(OutputBytes[Channel]); //assign direction to next channel.
      }

     for (int Channel = 0; Channel < PinCount; Channel++) {//iterate through each channel in turn, displaying the requested value
         analogWrite(PinSetUp[Channel],0); //write magnitude of requested intensity to output
         digitalWrite(DirPinSetUp[Channel], OutputDirection[Channel]);
         analogWrite(PinSetUp[Channel],abs(OutputBytes[Channel]));
      }
   }
   }
}
```

Haptic Signal Generator Python 2.7 Script

```
#import required libraries
import serial
import time

#define initial variables

arduinoport = 'COM5' #Set the COM port for the first arduino
duration = 0.1 #duration of each frame in seconds (decimal places permitted)

#set directory to download files to.
dir_path = "/Users/rayjh/Documents/Work/SuitCEYES/" #directory where you want the file to be downloaded to - includes file name
file_title = "DataRun" #title for CSV files to be stored to. Note that they will be appended with a number so that they do not overwrite each other.

iteration = 0 #set iteration counter to 0. This is used to give each signal a unique ID for recording purposes.

with serial.Serial(arduinoport, 9600, timeout=0.5)as ser: # Open the Serial connection to the Arduino

#handshake with Arduino:
    handshake_stage = 0
    print("beginning handshake")
    while handshake_stage < 2:
        if ser.inWaiting() > 0: #if data is waiting.
            character = ser.readline() #read the next line.
            if handshake_stage == 0:
                print(character)
            if handshake_stage == 1:
                channels = int(character)
                print('Channels detected: '+str(channels))
            handshake_stage += 1

#main loop
    while True:
        status = ser.readline()
        print(status)
        signal = raw_input("Enter Display Intensities in " + str(duration) + "s intervals, separated by a comma")
        signal_split = signal.split(",")
        signal_length = len(signal_split)
        current_frame = 0 #integer to keep track of length of signal
        current_channel = 0 #integer to keep track of frames
        while current_frame < signal_length:
            ser.write(signal_split[current_frame]+"/n")
            current_channel+=1
            current_frame+=1
            if current_channel == channels:
                time.sleep(duration)
                current_channel = 0
```