

**WP5 Controller Documentation**

## 

Overview: This document provides an overview of the controller system developed in Work Package 5 for Deliverable 5.4, to enable other partners to make use of these controllers. It provides an overview of the working principles of the controllers; the basic hardware configuration; how the software operates; and finally provides some examples of how to set up different tactile displays.

Current Version: v1.0

Document History:

- 9/5/2019 - v1.0 Raymond Holt (UNIVLEEDS) – first version created.

1. Controller Overview

The controllers are intended to provide signals that can be used to drive a set of actuators in a defined pattern. The purpose of the controllers developed here is to provide a toolkit for prototyping and psychophysical testing: for this reason, flexibility and adaptability have been prioritised above the fidelity that could be achieved by a dedicated controller tuned to a specific actuator. This section provides an overview of the principles on which the controllers work, the hardware used, and finally the software setup being used, before the rest of the document goes into greater detail on these.

**1.1 Monodirectional and Bidirectional Control**

Actuators come in many forms, and they are not all controlled in the same way. Most, however, adjust their intensity based upon the voltage supplied to them: as voltage is increased, a vibration motor will vibrate faster and with greater strength, a peltier module will heat more rapidly, a motor or linear actuator will move faster, and so forth. For some actuators, the direction of this voltage also has an effect. For example, in a peltier element, reversing the voltage across it reverses the direction in which heat is pumped and whether a given surface is attempting to heat or cool. Accordingly, the controller distinguishes between bidirectional control (where the direction of voltage can be reversed), and monodirectional control (where the direction of the voltage is fixed). In this design, bidirectional control is provided H-bridge which takes intensity and direction signals from the controller.

**1.3 Working Principles of the Controllers**

Controllers work as an interface between a Central Unit (which could be a desktop or laptop computer, or a small computer such as the Raspberry Pi), and the actuators used to display a haptic signal. The controllers are based around the concept of displaying a sequence of frames to a set of channels, as specified by the Central Unit. Each channel drives one actuator, and the frame specifies the intensity of the haptic stimulus each channel should display for that frame, and how long the frame should last: all channels will display the requested intensity for the duration of the frame. It is worth noting that the intensity and duration of each frame define the signal sent to the actuator: how the actuator then behaves will depend upon the properties of the actuator. The controller has two different methods of operation – Open Loop, in which the intensity represents the duty cycle of the PWM signal used to drive the actuator; and Closed Loop, in which the intensity represents the target reading on a sensor that the controller will try to achieve using Proportional-Integral Derivative Control. If bidirectional display is enabled, then negative intensities can be sent, these provide an equivalent magnitude in terms of intensity, but in the reversed direction. If bidirectional display is not enabled, then negative numbers will just be treated as positives of the same magnitude.

For Open Loop control, intensity values must be in the range -100 to 100: any values less than -100 will be treated as -100; any values greater than +100 will be treated as 100, since the PWM signal cannot have a duty cycle of greater than 100%. For Closed Loop control, it makes no sense to speak of a percentage, so the intensity requested denotes a value to be added to a defined baseline to provide a target ADC reading for the feedback setting. In this case, each increment of 1 denotes an increment on the ADC, and values beyond -100 and +100 can be used. The only limit is that the ADC on the Arduino only accepts values in the range 0 to 1023. As the controller does not calibrate the signal (so that it does not need to be changed when different actuators are used), this must be done by the Central Unit.

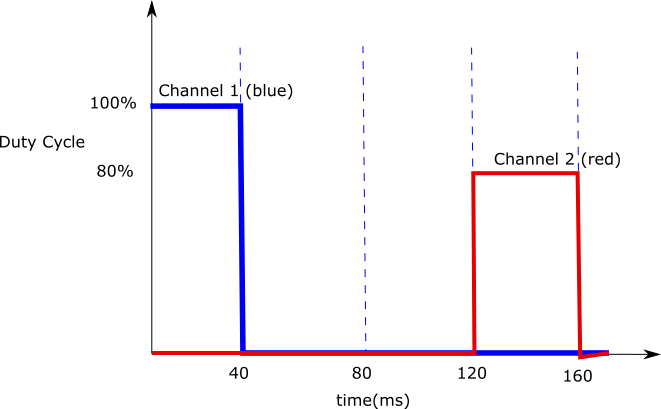
To display a signal, a sequence of numbers are sent to the controller over a serial connection (such as a USB cable from a desktop computer), specifying the intensity for each channel for each frame, and the duration of each frame. For example, with a two channel controller, a row comprising: “100, 0,40, 0, 0, 80, 0,80,40” would lead to the following frames:

Frame 1: Channel 1 at 100% duty cycle, Channel 2 at 0% Duty Cycle, for 40ms

Frame 2: Both Channels at 0% duty cycle for 80ms

Frame 3: Channel 2 at 80% duty cycle for 40ms.

This is illustrated in Figure 1, below.

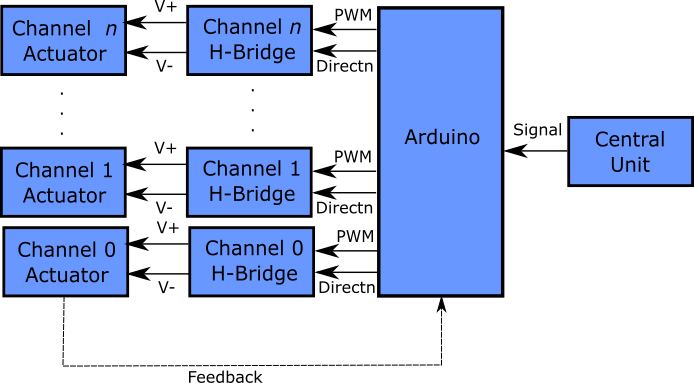
****

**Figure 1:** Example of how the Controller will interpret “100, 0,40, 0, 0, 80, 0,80,40” with two channels for a Simultaneous Channels signal.

The downside to this is that such signals contain a lot of information, which makes them laborious to write and slow to transmit, particularly when large numbers of channels and frames are involved. As the controller waits until it has received the full signal before beginning to display it, the gap between the signal being sent and it being displayed by the actuators increases as signals get more complex. To enable users to manage the trade-off between signal complexity and display time, three modes of communication are provided which allow signals to be sent more efficiently. These are discussed in greater detail in Section 3: Software Setup.

**1.2 Hardware Overview**

The controllers have been designed to be implemented upon the Arduino platform, using a Digilent Pmod Hb3 H-bridge board where bidirectional control is required. The code has been developed and tested using Arduino Nanos, but the code can readily be implemented with other Arduino platforms (such as the Arduino Mega) to offer more channels. The basic setup is illustrated in Figure 2, below. In order to allow bidirectional control, each channel requires two pins on the Arduino: a PWM pin for the control signal, and a second pin for determining direction, as illustrated in Figure 2. The Arduino can also gather data from analog sensors via its analog pins. If Closed Loop control is used, then a sensor must be attached for Channel 0.



**Figure 2:** Haptic Display Unit Physical Architecture

An Arduino Nano is able to provide five such channels, and read from up to six sensors. If bidirectional control is not required, then it is possible to use the direction pins to control a further five channels, though these are non-PWM pins, so it is only possible to set the intensity at 0% or 100%, rather than setting it to some intensity in between. The H-bridges serve two purposes: to allow more current to the actuators than the Arduino could handle directly, and to allow the reversing of current direction for bidirectional display. If bidirectional display is not required, then the h-bridges can be replaced with transistors.

**1.3 Software Overview**

The code on the Arduino runs in four phases, as illustrated in Figure 3, below. Upon starting up, the Arduino attempts to initiate a handshake with the Central Unit over the serial connection. During this handshake, the Central Unit specifies to the Arduino a set of parameters determining how it should behave. At the most basic, this covers how many channels and sensors should be used, and whether Closed Loop control should be used on Channel 0. In addition, parameters can be set to determine the tuning of closed loop control, and whether feedback or diagnostic data should be provided. Once the handshake is complete, the Arduino will wait to receive a signal. Once a signal has been received, it will display it, then offload any feedback or timing data it has gathered.



**Figure 3:** Arduino Process Stages

In principle, as long as data is received in the correct order, any program could be used to communicate with the Arduino, but to help support this process, a SUITCEYES Python module has been produced that can be imported into Python code and used to manage the communication properly.

1. Hardware Overview

This section provides a more detailed discussion of the Hardware Setup used by the controller, and how the controllers can be set up.

**2.1 Connecting Actuators**

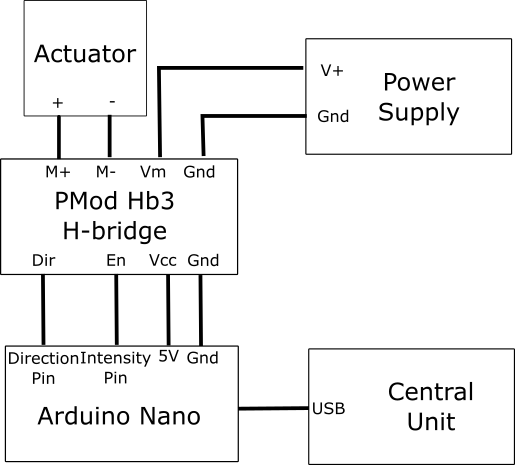
The Arduino is connected on the basis of “channels”. The setup of the Arduino means that the channel numbers are not able to correspond to the pin numbers of the Arduino. This is partly because some of the pins are required for specific functions that might cause them to interfere with the display[[1]](#footnote-1) and because the PWM-enabled pins on the Arduino are not sequential. The configuration and setup of these channels is determined by the Handshake between the controller and the Arduino. If 5 or fewer channels are selected, then bidirectional display is enabled; if 6 to 10 channels are selected then bidirectional display is automatically disabled.

**2.1.1 Bidirectional Display**

If bidirectional display is enabled, then each channel is allocated both an Intensity and Direction Pin, as listed in Table 1, so that they can be attached to a Pmod HB3 H-bridge as illustrated in Figure 4. The Pmod HB3 was chosen as it can supply 2A and provide reversible PWM output with only two pins.

**Table 1: Bidirectional Display Settings, enabled for up to 5 channels.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Channel Number** | **Intensity Pin** | **Direction Pin** | **Closed Loop Possible?** |
| 0 | DIO3 | DIO2 | Yes – feedback sensor must be attached to A0 |
| 1 | DIO5 | DIO4 | No |
| 2 | DIO6 | DIO7 | No |
| 3 | DIO9 | DIO8 | No |
| 4 | DIO10 | DIO12 | No |



**Figure 4: Setting up a PMod HB3 H-Bridge**

If bidirectional display is not required, then there is no need to use an H-bridge, even if 5 or fewer channels are being used: a transistor can be used as an alternative using the setup described in the next section – in this case the direction pin does not need to be connected to anything.

If bidirectional display is enabled, then Closed Loop control can be enabled on Channel 0 (only). In this case, the corresponding feedback sensor must be attached to Pin A0 on the Arduino.

Bidirectional display is applicable to actuators that can have their direction reversed: such as peltier elements; it can also be used to provide a braking effect on vibration motors.

**2.1.2 Monodirectional Display**

If more than 5 channels are requested in the Handshake, then bidirectional display is automatically disabled, and the Direction Pins are repurposed as intensity pins. However, as these pins are not PWM-enabled on the Arduino, they can only be set “on” or “off”, hence any Intensity of a magnitude of 51% or above will be treated as 100%, and any intensity with a magnitude of 50% or less will be treated as 0%. Note that as bidirectional display is disabled, sending negative numbers will do nothing to the direction of the display: they will be treated by positive values of the same magnitude on both PWM and non-PWM enabled pins. Hence, a signal of “-60” will be treated as 60% intensity if sent to channel 0 and 100% intensity if sent to Channel 5. It is possible to use the Pmod HB3 H-bridge with a monodirectional display (as a way of providing more current than possible through the Arduino alone), but given the expense, it is probably easier to use a transistor. An example circuit for setting up a transistor with a vibration motor is shown in Figure 5.

**Table 2: Monodirectional Display Settings, enabled when 6-10 channels are requested.**

|  |  |  |
| --- | --- | --- |
| **Channel Number** | **Intensity Pin** | **PWM Possible?** |
| 0 | DIO3 | Yes |
| 1 | DIO5 | Yes |
| 2 | DIO6 | Yes |
| 3 | DIO9 | Yes |
| 4 | DIO10 | Yes |
| 5 | DIO2 | No |
| 6 | DIO4 | No |
| 7 | DIO7 | No |
| 8 | DIO8 | No |
| 9 | DIO12 | No |

**-**

**+**

**M**

transistor

Power

Supply

Vibration

Motor

Capacitor

Schottky

Diode

Resistor

Arduino

Output

Pin

Gnd

**Figure 5: using a Transistor to Control a Vibration Motor**

1. Software Overview

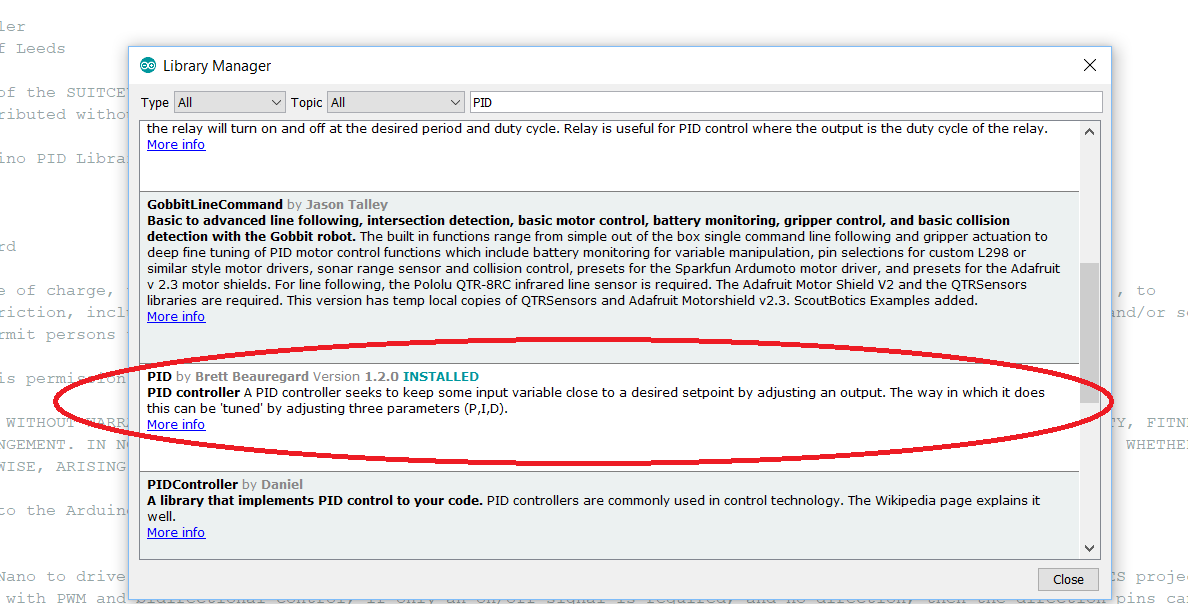
This section provides a detailed overview of the Arduino program used on the controller, as well as the Python library developed to facilitate communication with it.

**3.1 Arduino Software**

As shown in Figure 3, the Arduino goes through four stages: 1) Handshake; 2) Wait for signal; 3) Transmit Signal; and 4) Reporting. During each of these stages, the Arduino needs to communicate with the Central Unit over the Serial connection, but with slightly different information in each stage. This section outlines the communications used in each stage.

**3.1.1 Setup**

The Arduino code can be installed to the relevant Arduino using the Arduino IDE. Note that the Arduino PID Controller Library v1.2.0 by Brett Beauregard must be installed for the program to compile properly, even if you do not intend to use Closed Loop control. This can be downloaded via the Arduino IDE, by going to Sketch>Include Library>Manage Libraries and then searching for “PID”. Note that several PID controllers are available, so make sure it is the one called “PID” by Brett Beauregard, as shown in Figure 6.



**Figure 6:** The Arduino PID Library by Brett Beauregard, required for installation.

**3.1.2 Verbose Mode, Feedback and Timing Data**

The Arduino several modes that can be turned on or off during the handshake, that are used for diagnostics, or to gather information from any sensors attached, albeit at the expense of performance.

**Verbose Mode:** When this mode is active, the Arduino reports a lot of information over the serial communication to confirm what is happening, what signals it has received and how it has interpreted them. This is intended for debugging purposes, and can be turned on to identify whether the Arduino is behaving as it should. However, as it sends a large amount of serial data, this greatly slows the Arduino down, so should only be used when needed.

**Feedback:** When this mode is active, the Arduino will take readings from the specified number of analogue pins (one for each sensor requested) and report this during signal transmission. This has the side effect of making frame timing less precise, particularly for shorter frames.

**Timing Data:** When this mode is active, the Arduino will report the actual start time of each frame during reporting. This is really intended as a function to verify that the signal is transmitting in the pattern expected, and to investigate how settings impact the transmission of the signal in practice.

**3.1.3 Data Packets**

Transmission from the Arduino to the Central Unit is on the basis of the “Serial.println” such that each sequence sent out ends with the end of line character ‘\n’. This means that data can be read into a program such as Python using “Serial.readline()” which will automatically read the content of the serial buffer until the ‘\n’ character is encountered. Each line of data sent to the Central Unit from the Arduino begins with an identifying character to tell the Central Unit what is being reported.

The Arduino does not have an equivalent function to readline, so communication from the Central Unit to the Arduino is co-ordinated by sending information in “packets” of a known number of characters. Each packet comprises a set of digits making up a single integer, with packets separated by a comma. The default packet size is 4, though this can be adjusted in the Arduino code (allowing intensities of -100 to 100, and durations of up to 9999ms). Larger packets slow communication down. The Arduino code includes an option to set a multiplier which will be applied to all intensities and durations, allowing larger values to be sent with smaller packet sizes. The multiplier can only be changed in the Arduino code, and defaults to 1, so that it does not affect transmitted values.

As the Arduino uses the packet size to identify when a value has been transmitted, it is important that all characters are used. Hence, you cannot send the value “1” to the Arduino if the Packet Size is 4 as it will wait for a further three characters. Instead, you would need to send 0001 for 1, 0100 for 100, and so forth. Note that a negative sign is counted among these characters, so -100 for -100 and -080 for -80. The Arduino will attempt to read the packet until it finds a comma, and convert the digits before the comma into an integer. This means that only integers can be transmitted to the Arduino. The way the Arduino will interpret and respond to these integers will depend upon which stage of the program it is at: the following sections detail these behaviours.

**3.1.4 Handshake**

The handshake takes the following steps:

1. The Arduino specifies the multiplier it is using, and the Packet Size, so that the Central Unit knows how to communicate with it.

2. The Arduino listens for information from the Central Unit. It expects a sequence of 16 integers, separated by commas, which tell it whether Verbose Mode should be used; the number of channels to use; the number of sensors to use; whether Feedback should be gathered; whether Timing Data should be reported; whether Closed Loop control is active; and the various settings for Closed Loop Control.

3. The Arduino then sends a series of codes to confirm what it has understood – which modes are active, the values of the specified parameters, the number of channels and sensors to be used, which pins are being used for intensity display, directional display and sensors, and finally notifications if the number of channels requested is too great for it to display bidirectionally, or at all.

In the SUITCEYES Python module, the Handshake() function will take care of this, and interpret any messages raised, as well as providing default values for the closed loop parameters, so that the user does not need to specify them all manually.

**3.1.5 Listen for Signal**

Once the handshake is complete, the Arduino will wait for a signal to be transmitted to it. . As noted in Section 1.1, the Arduino has three ways of interpreting a signal, depending on how many channels need to be used simultaneously. The Arduino therefore waits for a single integer which tells it which mode it should use, a second integer which tells it how many frames are in the signal; it will then wait until all the data for the signal has been transmitted, before interpreting it using the specified mode.

The Arduino’s three modes are as follows:

**Single Channel:** If the first integer transmitted is 0 or above, the Arduino will interpret the signal as a single channel signal, to be displayed on the specified channel. It will then read the subsequent integers in pairs, representing an intensity and a duration for each frame on the given channel. In the SUITCEYES Python module, the DirectSignal() function will transmit this kind of signal.

**Simultaneous Channels:** If the first integer transmitted is -1, then the Arduino will interpret the signal as a Simultaneous Channels signal, and interpret the integers as outlined in Section 1.1. In the SUITCEYES Python module, the TransmitSignal() function will transmit this kind of signal.

**Sequential Channels:** If the first integer transmitted is -2, then the Arduino will interpret the signal as a sequential channel signal, with each frame to be displayed on a single channel only, but with the capacity to use different channels for each frame. It will then read the subsequent integers in triplets, representing the channel to be used for the given frame, the intensity to be displayed on that channel for that frame; and the duration of the frame. In the SUITCEYES Python module, the MultiDirectSignal() function will transmit this kind of signal.

There is a fourth mode (“Silent Mode”) available, specified by sending the first integer as “-3”. In this case, the Arduino displays nothing for the duration of the signal, though it will still return feedback. This is really intended for co-ordination between Master and Slave Arduinos in multi-Arduino communication. In the SUITCEYES Python module, the BlankSignal() function will transmit this kind of signal.

It is worth noting that in the current setup, timestamps begin from the moment the signal starts to be received. This is so that the delay between transmitting a signal and it starting to display can be identified. If it is desired that timing begins at the start of the first frame, then this can be done in the Arduino code by uncommenting line 633 (which reads “SignalStart = millis();”). The principle benefit of doing this is that it ensures that in Multi-Arduino communication, it ensures that both Arduinos have comparable timestamps (rather than having to calculate the offset between them).

**3.1.6 Transmit the Signal and Reporting**

Once the full signal has been received, the Arduino will begin to display it. If the Arduino is set to Master, it will set Pin 11 to High. This is intended for use in co-ordinating communication with other Arduinos, either for data gathering purposes, or as part of co-ordinating communication among multiple Arduinos. If the Arduino is set to Slave, then it will wait for a High signal on Pin 11 before starting the display.

While the signal is displaying, if Feedback is to be reported, the Arduino will read from the requested number of sensor pins and send a line across the serial connection to the Central Unit, beginning “F,” then followed by the ADC values from relevant the sensor pins, separated by a comma. Note that the use of feedback does make frame timing more variable. The Arduino will signal the end of transmission by sending “C,,,”.

Once the Signal has been transmitted, the Arduino will then try to send its timing data: a record of when each frame started. This will be sent across serial to the Central Unit by sending a line for each frame which begins “T,” followed by the relevant timestamp. The Arduino will signal the end of transmission of Timing Data by sending “C,,,”.

All of the methods for transmitting the signal described in the previous section will return both the data gathered from the Feedback and the Timing Data, and the SUITCEYES module includes WriteFeedback() and WriteTimingData() functions so that these can be written to a CSV file.

Note that if Feedback and Timing Data are turned off only the “C,,,” messages the end of each phase will be reported. These still need to be removed from the serial buffer by the central unit.

**3.2 Python Module**

To help make communicating with the Arduino as straightforward as possible, a module for Python 2.7 (“SUITCEYES.py”) has been written containing a set of functions that automate the communication with the controller, rather than having to deal with the ordering of messages outlined above. They also provide default values for some of the parameters, so that the you do not need to worry about specifying them. For example, if you are not interested in using Closed Loop control, having to specify the 11 parameters associated with it seems wasteful: the SUITCEYES module provides default values that will be transmitted to the controller unless overriden by the user. Note that as the communication all takes place over serial, you will need to use import pySerial and use this to create a serial object.

**3.2.1 Handshake():**

Carries out the Handshake with the Arduino, provides values in the correct ordering, and reports the packet size being used by the Arduino. It will also interpret the messages sent by the Arduino so that feedback can be provided when in Verbose mode. As a minimum, the user must specify the Serial Object over which communication is to take place, the number of channels to be set up, and the number of sensors to be used. All other values have defaults, and do not need to be specified unless required. By default, Verbose, Feedback, Timing Data and Closed Loop Control are all turned off.

**3.2.2 TransmitSignal()**

Sends a simultaneous channel signal to the Arduino, and gathers any feedback and timing data reported by the Arduino as arrays. The user must specify the Serial Object over which communication will take place, the number of channels to be used, the Packet Size being used, and the Signal itself. The number of channels used in the signal can be less than or the same as the number of channels defined in the handshake, provided it is consistent with the way the signal has been formatted. Optionally, the user can specify whether the program should be Verbose, printing everything received from the Arduino: unlike the Handshake function, this has no effect upon what is reported from the Arduino, only whether it is printed to the screen.

**3.2.3 DirectSignal()**

Sends a single channel signal to the Arduino, and gathers any feedback and timing data reported by the Arduino as arrays. The user must specify the Serial Object over which communication will take place, the channel to be used, the Packet Size being used, and the Signal itself. Optionally, the user can specify whether the program should be Verbose, printing everything received from the Arduino: unlike the Handshake function, this has no effect upon what is reported from the Arduino, only whether it is printed to the screen.

**3.2.4 MultiDirectSignal()**

Sends a sequential channel signal to the Arduino, and gathers any feedback and timing data reported by the Arduino as arrays. The user must specify the Serial Object over which communication will take place, the Packet Size being used, and the Signal itself. Optionally, the user can specify whether the program should be Verbose, printing everything received from the Arduino: unlike the Handshake function, this has no effect upon what is reported from the Arduino, only whether it is printed to the screen.

**3.2.5 WriteFeedbackData()**

Writes the feedback data array gathered from TransmitSignal(), DirectSignal() or MultiDirectSignal() to the specified CSV file.

**3.2.6 WriteTimingData()**

Writes the timing data array gathered from TransmitSignal(), DirectSignal() or MultiDirectSignal() to the specified CSV file.

1. Example Python Script

To help demonstrate the controller, I have provided an example Python script that makes use of the SUITCEYES python module. In the “User Defined Variables” section, you can set the number of channels, senors, a working directory, and the Serial port address for the Arduino, as well as the various options for Feedback, Timing Data, Verbose mode, and Closed Loop control. You can also specify a “haptic library” file, where signals are stored in a CSV file.

When the program begins you will be asked to give a participant identifier, which will be used to label all files produced. After this, you will be asked which signal to display. You can then specify a number to request one of the rows in the Haptic Library file (0 being the first row, 1 the second row and so forth, as Python is zero-indexed). Alternatively, you can enter one of several command words to request the ability to type out a signal directly:

1. **“override”**: allows you to manually enter a Simultaneous Channel Signal. You will be prompted to enter the signal.
2. “**direct”:** allows you to enter a Single Channel Signal: you will be prompted for the channel, followed by the signal. Finally, you can enter the word “multidirect”.
3. **“multichannel”:** allows you to enter a Sequential Channel Signal: you will be prompted to enter the signal directly.

**Appendix 1:** Detailed Description of the Arduino Handshake Process

This section provides a detailed descriptions of the Handshake process used by the controller, in case one wishes to use an alternative to the SUITCEYES Python module’s Handshake() function. The first thing the Arduino does upon turning on is initiate serial communication at 9600 baud, and initiate the Handshake, with the following steps:

1. The Arduino specifies the multiplier it is using, and the Packet Size, so that the Central Unit knows how to communicate with it.

2. The Arduino then listens for information from the Central Unit. It expects a sequence of 16 integers, separated by commas, which tell it the following in this order:

|  |  |
| --- | --- |
| **Sequence** | **Interpretation** |
| 1 | Whether Verbose Mode should be used:  “1” turns Verbose mode on; any other value turns Verbose Mode off. |
| 2 | Number of channels to use. |
| 3 | Number of sensors to use. |
| 4 | Whether Feedback should be reported:  “1” turns Feedback mode on; any other value turns Feedback Mode off. |
| 5 | Whether Timing Data should be reported:  “1” turns Timing Data on, any other value turns Timing Data off. |
| 6 | Whether Closed Loop control should be active on Channel 0:  “1” turns Closed Loop control and Feedback Mode on; any other value turns Closed Loop off. |
| 7 | Proportional Gain for the PID controller. |
| 8 | Integral Gain for the PID controller. |
| 9 | Derivative Gain for the PID controller. |
| 10 | The ADC Offset to be used in Closed Loop Control (the ADC reading on Pin A0 that should be taken as a value of “0” when signals are presented). |
| 11 | Whether a Variable Offset should be used – if enabled, the ADC Offset will be overridden by taking an average of ten readings from Pin A0, and using this as the offset.  “1” turns Variable Offset on; any other value turns it off. |
| 12 | Whether a Cut Off should be active on Channel 0: if the Cutoff is active, any ADC reading on Pin A0 that falls below the specified Lower Cutoff, or above the Specified Upper Cutoff will cause the signal to cease. |
| 13 | The Lower Cutoff value to be used (0 to 1023) |
| 14 | The Upper Cutoff value to be used (0 to 1023) |
| 15 | The threshold for determining when a Closed Loop reset has reached the target point (that is, how close the reading on Pin A0 needs to be for the reset to be considered complete). |
| 16 | Whether a Closed Loop Reset should be used – if active, this will attempt to return the reading on Pin A0 to the specified ADC Offset +/- the specified threshold. |

3. The Arduino will then send a set of feedback to confirm the settings it has received, and any errors it has raised. These are:

E1: identifies that more channels have been specified than there are pins available (including direction pins). If this happens, the Arduino will automatically set the number of channels to the maximum available.

Ch: Number of channels identified

E2: identifies that the number of channels requested is more than the number of bidirectional channels available. In this case, bidirectional mode is automatically switched off.

OP: states the digital pins to be used as intensity pins.

DP: States the digital pins to be used as direction pins.

Se: States the number of sensors

SP: Secifies which pins are being used for the Sensors.

PI: Specifies the identified Packet Interval

MaxSL: Max signal length in frames.

V: Verbose Mode active.

NB: Verbose Mode off.

M: Specifies that the Arduino is in Master Mode.

S: Specifies that the Arduino is in Slave Mode.

FB: Specifies that Feedback is active.

NFB: Specifies that Feedback is not active.

Ti: Specifies that Timing Data is active.

NT: Specifies that Timing Data is not active.

CL: Specifies that Closed Loop is active.

GADC: Specifies that Variable Offset if active.

ADCO: specifies the ADC Offset requested.

Finally, the Arduino prints “C,,,,” to indicate that the handshake is complete.

**Appendix 2:** Detailed Descriptions of the SUITCEYES Python 2.7 Module

This appendix provides more detail on the functions provided in the SUITCEYES module.

**A2.1 Handshake():**

Carries out the Handshake with the Arduino, provides values in the correct ordering, and reports the packet size being used by the Arduino. It will also interpret the messages sent by the Arduino so that feedback can be provided when in Verbose mode.

**Syntax**

Handshake(Port, channels, sensors)

Handshake(Port, channels, sensors, Verbose, Feedback, Timing, ClosedLoop, Kp, Ki, Kd, Offset, VariableOffset, CutOff, LowerCutOff, UpperCutOff, ADCThreshold, ADCReset)

**Parameters**

**Port:** The serial object to over which the handshake should be conducted (Serial object).

**channels:** The number of channels required (integer).

Sensors (integer): The number of analog pins for sensors to be used (integer)

**Verbose:** Whether the Arduino should run in Verbose Mode (boolean). False by default.

**Feedback:** Whether the Arduino should gather feedback from the sensors during signal transmission (boolean). False by default.

**Timing:** Whether theArduino should report timing data during the reporting stage (boolean). False by default.

**ClosedLoop:** Whether theArduino should use Closed Loop control on Channel 0 (boolean). False by default.

**Kp:** Theproportional gain for the PID controller (integer). 1 by default.

**Ki:** The integral gain for the PID controller (integer). 0 by default.

**Kd:** The derivative gain for the PID controller (integer). 0 by default.

**Offset:** TheADCreadingon Pin A0that shouldbe treatedaszero intensity when Closed Loop control is active. 510 by default.

**VariableOffset:** Whether the Offset should be overridden by reading the initial value on Pin A0 (boolean). False by default.

**CutOff:** Whether the Arduino should turn off the signal if the reading on Pin A0 falls outside the specified limits (boolean). False by default.

**LowerCutOff:** Thelowest acceptable reading on Pin A0 before CutOff is activated (integer). 0 by default.

**UpperCutOff:** Thehighest acceptable reading on Pin A0 before CutOff is activated (integer). 1023 by default.

**ADCThreshold:** How close the reading on Pin A0 needs to be to the Offset value for reset to be considered complete (integer). 10 by default.

**ADCReset:** Whether the PID controller should try to return the reading on Pin A0 to the specified Offset before allowing a new signal to begin (boolean). False by default.

**A2.2 TransmitSignal()**

Sends a simultaneous channel signal to the Arduino, and gathers any feedback and timing data reported by the Arduino as arrays.

**Syntax**

TransmitSignal(Port, channels, Packet\_Size, SignalToSend)

TransmitSignal(Port, channels, Packet\_Size, SignalToSend, Verbose)

**Parameters**

**Port:** The serial object to which the signal should be sent (Serial object).

**channels:** The number of channels used in this signal – note that the Arduino will always use channels sequentially. If you want to use Channels 1 and 3 in a sequential signal, you would need to specify 4 channels, so that channels 0 through 3 are used: requesting 2-channels will only use chanenls 0 and 1.

**Packet\_Size:** The number of characters in each transmission. This needs to be the value brought in from the Handshake.

**SignalToSend:** The signal to be sent, as a series of integers separated by commas, representing for each frame in turn the intensity for each channel to be displayed that frame and the duration of the frame in milliseconds. The function will automatically ensure that shorter values are padded to the correct Packet Size. Larger values will be transmitted incorrectly.

**Verbose:** Whether process messages from the Arduino and the function should be reported – unlike the Verbose setting in the handshake, this has no effect on the Arduino, only on what is printed to the screen by Python (boolean). False by default.

**Returns** SensorData, TimingData

**A2.3 DirectSignal()**

Sends a single channel signal to the Arduino, and gathers any feedback and timing data reported by the Arduino as arrays.

**Syntax**

DirectSignal(Port, channel, Packet\_Size, SignalToSend)

DirectSignal(Port, channel, Packet\_Size, SignalToSend, Verbose)

**Parameters**

**Port:** The serial object to which the signal should be sent (Serial object).

**channel:** The number of the channel to be used in this signal.

**Packet\_Size:** The number of characters in each transmission. This needs to be the value brought in from the Handshake.

**SignalToSend:** The signal to be sent, as a series of integers separated by commas, representing for each frame in turn the intensity for to be displayed and the duration of the frame in milliseconds. The function will automatically ensure that shorter values are padded to the correct Packet Size. Larger values will be transmitted incorrectly.

**Verbose:** Whether process messages from the Arduino and the function should be reported – unlike the Verbose setting in the handshake, this has no effect on the Arduino, only on what is printed to the screen by Python (boolean). False by default.

**Returns** SensorData, TimingData

**A2.4 MultiDirectSignal()**

Sends a sequential channel signal to the Arduino, and gathers any feedback and timing data reported by the Arduino as arrays.

**Syntax**

MultiDirectSignal(Port, Packet\_Size, SignalToSend)

MultiDirectSignal(Port, Packet\_Size, SignalToSend, Verbose)

**Parameters**

**Port:** The serial object to which the signal should be sent (Serial object).

**Packet\_Size:** The number of characters in each transmission. This needs to be the value brought in from the Handshake.

**SignalToSend:** The signal to be sent, as a series of integers separated by commas, representing for each frame in turn the channel to be used for that frame, the intensity to be displayed on that channel, and the duration of the frame in milliseconds. The function will automatically ensure that shorter values are padded to the correct Packet Size. Larger values will be transmitted incorrectly.

**Verbose:** Whether process messages from the Arduino and the function should be reported – unlike the Verbose setting in the handshake, this has no effect on the Arduino, only on what is printed to the screen by Python (boolean). False by default.

**Returns** SensorData, TimingData

**A2.5 WriteFeedbackData()**

Writes the feedback data array gathered from TransmitSignal(), DirectSignal() or MultiDirectSignal() to the specified CSV file.

**Syntax**

WriteFeedbackData(FeedbackFile, FeedbackArray):

WriteFeedbackData(FeedbackFile, FeedbackArrayVerbose):

**Parameters**

**FeedbackFile:** The file path (including filename and .csv extension) to be written to.

**FeedbackArray:** The TimingData array to be written to the

**Verbose:** Whether the timing data should be written to the screen while the CSV file is being written– unlike the Verbose setting in the handshake, this has no effect on the Arduino, only on what is printed to the screen by Python (boolean). False by default.

**Returns**

Nothing

**A2.6 WriteTimingData()**

Writes the timing data array gathered from TransmitSignal(), DirectSignal() or MultiDirectSignal() to the specified CSV file.

**Syntax**

WriteTimingData(FeedbackFile, FeedbackArray):

WriteTimingData(FeedbackFile, FeedbackArrayVerbose):

**Parameters**

**FeedbackFile:** The file path (including filename and .csv extension) to be written to.

**FeedbackArray:** The TimingData array to be written to the

**Verbose:** Whether the timing data should be written to the screen while the CSV file is being written– unlike the Verbose setting in the handshake, this has no effect on the Arduino, only on what is printed to the screen by Python (boolean). False by default.

**Returns**

Nothing

1. Specifically: DIO0 and DIO1 are used for Serial Communication; DIO13 operates the LED; and a fourth DIO pin is reserved as a trigger to enable co-ordination with other Arduinos or data gathering systems. [↑](#footnote-ref-1)