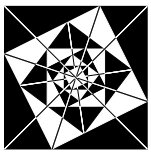


Technische Universität Berlin

Development of a Test Rig for the Automated Measurement of the Static Force Characteristic of Pneumatic Muscle Actuators

Bachelor Thesis

Michael Drummond
351750

16th December, 2016

Examiner: Prof. Dr.-Ing. J. Raisch
Supervisor: M.Sc. M. Martens

Control Systems Group
Institute of Energy and Automation Technology
Faculty IV - Electrical Engineering and Computer Science
Technische Universität Berlin

Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichert an Eides statt.

Berlin, den

Kurzfassung

Pneumatische Muskelantriebe werden aufgrund ihrer Leichtigkeit und Konformität in einer Reihe von Roboteranwendungen eingesetzt. Das statische Kraftkennfeld beschreibt die Beziehung zwischen dem Innendruck eines pneumatischen Muskels, seiner Länge und der Kraft, die er ausübt. Das Kennfeld kann mathematisch oder experimentell ermittelt werden, wobei letztere zeitaufwendiger, aber aktuell genauer ist. Dies kann dann beispielsweise dazu verwendet werden, eine "sensorlose" Kontrollstrategie für ein antagonistisches Muskelpaar in einem Gelenk zu entwickeln.

Diese Arbeit beschreibt die Hard- und Softwareentwicklung eines Prüfstands zur schnellen und automatischen Messung des statischen Kraftkennfeldes eines pneumatischen Muskels. Das Gerät basiert auf einem Linux-Singleboard-Rechner, für den zusätzliche Schaltungen entworfen wurden und eine Leiterplatte entwickelt wurde. Es ist mit einem Display und Jog-Dial ausgestattet, mit dem ein Menü und Optionen navigiert und ausgewählt werden können. Ein digitaler PID-Regler regelt den Innendruck des zu testenden pneumatischen Muskels mit Hilfe eines Drucksensors. Schließlich nimmt ein Kraftsensor Messungen vor, um das statische Kraftkennfeld zu ermitteln.

Um die Wirksamkeit und Eignung der Vorrichtung vorzuführen, wird sie verwendet, um das statische Kraftkennfeld von zwei pneumatischen Muskeln zu bestimmen. Die Messungen zeigen die Leichtigkeit und Genauigkeit, mit der das Kennfeld solcher Muskeln ermittelt werden kann. Anschließend werden einige mögliche Erweiterungen und Verbesserungen der Vorrichtung besprochen.

Abstract

Pneumatic muscle actuators are used in a range of robotics applications due to their lightweight and compliant nature. The static force characteristic describes the relationship of a pneumatic muscle's internal pressure, to its length and the force it exerts. The characteristic can be ascertained mathematically or experimentally, with the latter being time consuming but currently more accurate. This can then be used, for example, to develop a "sensorless" control strategy for an antagonistic muscle pair in a joint.

This thesis describes the hard- and software development of a test rig to rapidly and automatically measure the static force characteristic of a pneumatic muscle. The device is based around a Linux powered single board computer, for which additional circuitry was designed and a printed circuit board developed. It is equipped with a display and jog wheel, with which a menu and options can be navigated and selected. A discrete PID controller regulates the internal pressure of the pneumatic muscle being tested with the help of a pressure sensor. Finally, a force sensor takes readings to establish the static force characteristic.

To demonstrate the device's efficacy and suitability, it is used to determine the static force characteristics of two pneumatic muscles. The measurements show the ease and accuracy with which the characteristic of such muscles can be established. Lastly, some possible extensions and improvements to the device are discussed.

Contents

1 Introduction and Theory

1.1 Pneumatic Muscle Actuators

Pneumatic muscle actuators go by many names, such as braided pneumatic muscles (BPMs), McKibben pneumatic artificial muscles (PAMs), fluidic muscles or biomimetic actuators. First developed in the 1950s for use in an orthopaedic appliance by the physician Joseph McKibben [1], PMAs have since been put to use in a range of applications from industry, to medicine and entertainment [2]. PMAs typically consist of an elastomeric tube or bladder, a braided mesh of non-elastic fibres that are either worked into the bladder or wrap around it like a sleeve, and two end fittings that seal the tube but for an opening which attaches to a source of compressed air. The non-elastic mesh is attached to the fittings so, when the bladder is inflated via the opening at one end, the bladder can only expand radially and a tensile force is generated in the axial direction [3].

In robotic applications, PMAs have certain advantages over other actuators, such as servomotors. In particular, they are lightweight and inherently compliant. As they are principally made of a membrane and some end fittings, a muscle such as that tested in Section 4 can weigh a little over $200\mathrm{g}$ yet exert a maximum force of $6\mathrm{kN}$. Furthermore, if a force is exerted on the muscle, it will "give in", without increasing the force in the actuation. This compliant nature makes it ideal in human robot interactions [4].

1.2 Static Force Characteristic of PMAs

In robotics applications, fundamentally and in a physical sense, the actuators do work. In other words, $W = Fs$, the product of a force (F) that moves a displacement (s) is equal to the work done (W). This is evidently also the case for PMAs whose length changes with internal pressure. It is fair to say, that if no force is applied by the muscle, no work is done. Specifically, in a control environment, it is therefore crucial to understand the parameters that relate to this force.

The static force characteristic of a PMA describes the relationship between its internal pressure, its length and the force exerted. It is *static* because the relationship is defined for situations in which the parameters do not change over time. This relationship is fundamental to all mathematical models that describe PMAs and these models always incorporate the static force characteristic. For example, the model for position control of Festo pneumatic

muscles as developed by Boblan [5] is described as follows

$$F_{Festo}(L,P) = P \cdot \frac{\pi \cdot D_0^2}{4{L_0}^2 \cdot \tan^2(\Theta_0)} \cdot \left(3L^2 - \frac{{L_0}^2}{\cos^2(\Theta_0)}\right) - \frac{\pi \cdot E \cdot H \cdot D_0 \cdot L^2}{L_0 \cdot \tan(\Theta_0)} \left(\frac{1}{\sqrt{\frac{{L_0}^2}{\cos^2(\Theta_0)} - {L_0}^2}} - \frac{1}{\sqrt{\frac{{L_0}^2}{\cos^2(\Theta_0)} - L^2}}\right)$$

This model expresses the force exerted by a Festo muscle in relation to its length (L) and pressure (P). In addition, the diameter of the muscle (D_0), the muscle membrane's thickness (H) and the angle of the fibres running through this (Θ_0) tailor the model. (E) is an "elasticity module" that factors the elasticity of the membrane into the calculation.

Plot a graph of the above equation with all possible lengths and pressures that the muscle can attain and you get the static force characteristic as a curve. This can then be compared with a similar curve established experimentally. The comparison can inform the modification of the model.

The aim of the model is to describe the static force characteristic in a mathematical sense, so that this can then be used to create accurate position control. A further, common use for these models is in angle controllers for a joint based around an antagonistic muscle pair. In these cases, the parameter to be controlled is the angle of the joint, and the difference between the forces of the flexor and extensor muscles dictates the torque applied to the joint. Equations that describe the function of the joints at all points can be developed. For microcontroller based control in anthropomorphic robots these non-linear equations are linearised around an operating point. From there the state-space model and transfer function can be derived and a robust controller developed. [5]

In situations in which direct control of joint torque is wanted, the addition of force/torque sensors is necessary. These sensors invariably lead to more complex mechanical structures and can have a serious impact on cost in systems with many joints to control. However, there have been suggestions to develop a "sensorless" torque control strategy suitable for a joint based around an antagonistic muscle pair, that obviates the need for a force/torque sensor. The idea is to control the torque by directly controlling the force exerted by each muscle in the pair. Instead of using a force sensor, the force is estimated indirectly from the muscle length by means of the static force characteristic which has been measured beforehand. [6]

An accurate, experimentally measured static force characteristic can take some time to mea-

sure manually. The advantage of an automated test rig is, that much larger quantities of data can be gathered. Firstly, this is beneficial in generating a more accurate picture of the static force characteristic; and secondly, that this permits the analysis of the data with statistical tools to identify random error.

1.3 The Test Rig

The aim of the thesis was to create a self-contained device with which the static force characteristic of PMAs of different sizes could be tested automatically. Static force maps have previously been measured manually (M. Martens, personal communication, 2016). The set-up can be summarised as follows: a PMA is held at both ends at a given length with the aid of a rigid metal frame. A force sensor bridges the connection from the PMA to the frame at one end. As the internal pressure of the PMA is increased a tensional force is exerted on, and can be measured by, the force sensor. The force exerted is measured at multiple pressure points for a given length that the PMA is set at. This is repeated for multiple lengths, resulting in a "map" of the force response of a PMA at different pressures and lengths (see Figures 20 and 21 as examples).

The manual set-up involved opening a valve connecting the PMA to a source of compressed air by hand, adjusting to a given internal pressure of the PMA and reading the exerted force. The operation of the valve was not only time consuming but a laborious, repetitive action. The force sensor itself was a strain gauge which translates changes in applied force into a voltage. This in turn needed to be amplified before being read with a multimeter, the result was then converted to ascertain the original force exerted.

Following a run through of the pressure points at a given length the PMA then needed to be manually unbolted from the rigid frame that was holding it. The length of the PMA would then have to be changed by adjusting its internal pressure, before being bolted to the frame once more. See 4 for a photo of this set-up.

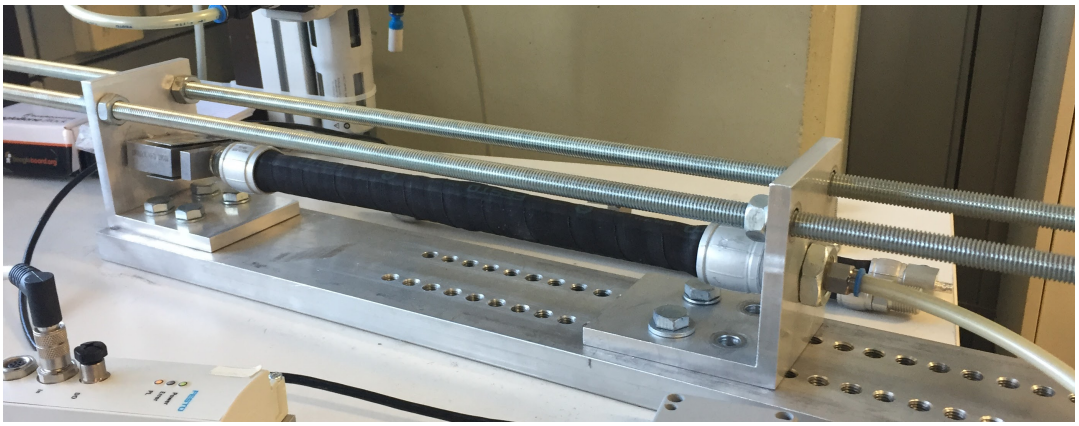


Figure 1: The adjustable metal frame holding the muscle, with bolts and a line of screw holes to change the muscle's length. The strain gauge is just visible at the far end and the compressed air inlet at the near end.

For this thesis a device was to be developed reduced the time with which an operator would have to continually contend with adjusting the valve to achieve predefined pressures. In addition, the automated measurement of the force applied by the PMA was to be incorporated into the design, such that measurements could be repeated any number of times and the statistical relevance of the collected measurements be improved.

2 Hardware Development

2.1 Overview

Complex systems are best divided into their component parts. Figure 2 gives an indication of the relevant connections between the elements of the device, whether it is an input or output, and by which means the connection is made.

The BeagleBone Black, a single board computer (SBC) running Linux, was chosen as the core of the device. For this application a microcontroller board such as the Arduino Due would potentially have been sufficient. However, it was felt the additional power and possible future expansion options (onboard data processing; web server user interface) available on a Linux powered SBC outweighed the negatives of dealing with a more complex system. Other SBCs considered were the Raspberry Pi, but the sheer number of connection options available to the BeagleBone Black (critical to this project) made it the better choice.

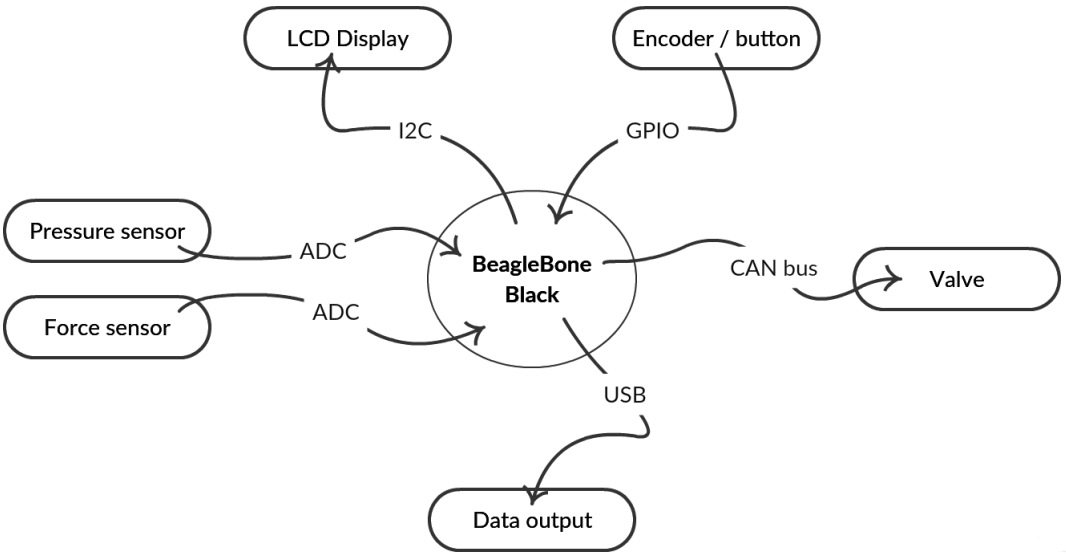


Figure 2: A simplified breakdown of device components, indicating direction of data transfer (I/O) and by which means

The BeagleBone Black is responsible for the control loop feedback mechanism that regulates the PMA internal pressure by means of the pressure sensor input and valve aperture output. Communication between the BeagleBone Black and the valve takes place over a CAN bus. The pressure sensor signal voltage has to undergo a small amount of signal conditioning before being converted by means of an analogue to digital converter on the Beaglebone Black.

The static force characteristic is what ultimately needs to be established, that is the relationship between the internal pressure of the pneumatic muscle, its length and the force that it exerts. The user of the test rig will set the muscle to a given length, while the controller sets the internal pressure of the muscle. This will result in a force, and it is this that is to be measured.

The signal from the force sensor is first passed through an external measurement amplifier before receiving signal conditioning and conversion via the BeagleBone Black's ADC. Once the measurements have been taken, the data (that is: force at a given length and internal pressure) can be transferred to a USB stick in an appropriate format for further processing elsewhere. The device and its function is controlled entirely with the use of a rotary encoder with a push button and an LCD Display (see 3.4 and Figure 19). A connection to a computer or other device is therefore not required.

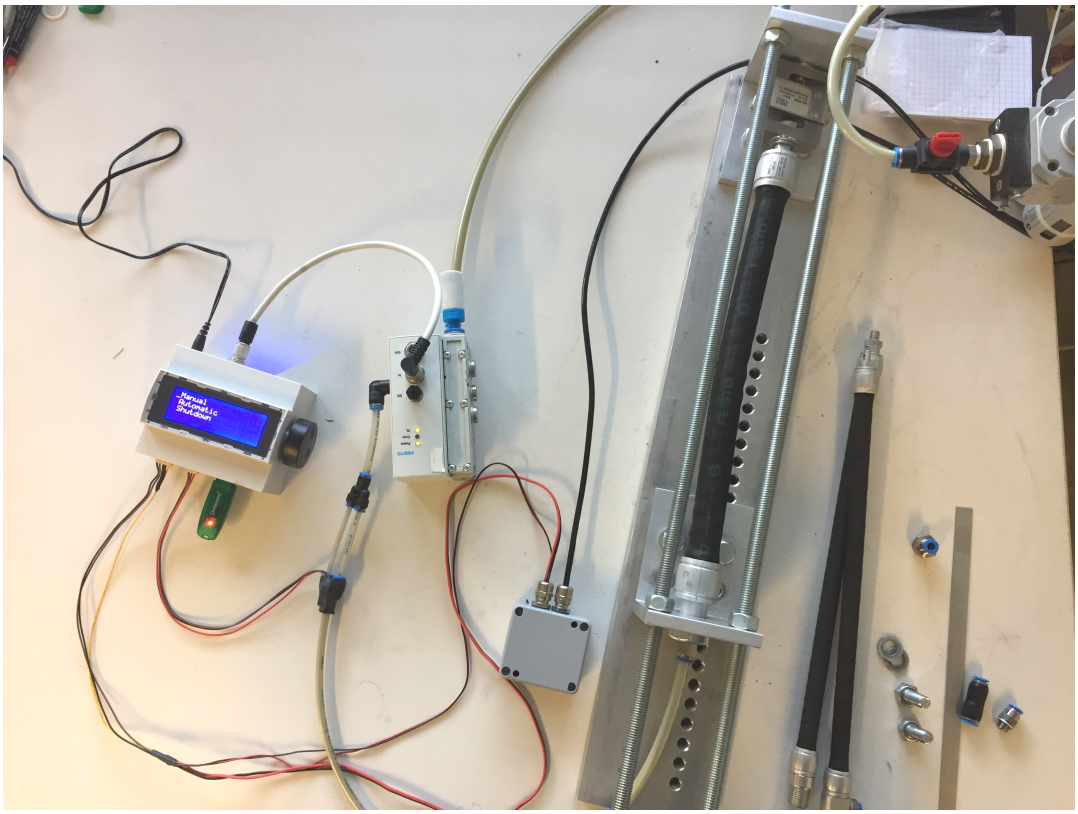


Figure 3: The complete test rig, with (from left to right) the control device, compressed air valve with a pressure sensor attached to its output, measurement amplifier (little grey box), and pneumatic muscle in adjustable frame.

The BeagleBone black is designed to mate with expansion boards (known as capes), with which its functionality can be extended. For this thesis a cape was developed to accommodate the additional circuitry required to make the device self-contained. This additional circuitry can broadly be divided into the following sub-circuits: power regulation, analogue signal conditioning, level-shifter for the LCD, CAN bus, buttons and LEDs. Surface mount components were used almost exclusively, to keep the overall size of the cape small. The small size allowed for the device to be fitted inside an off-the-shelf housing, reducing total design time.

The circuit and board were designed in Eagle, and the complete schematic and board layout can be viewed in appendices A and B respectively. In addition, the complete bill of material is in appendix C.

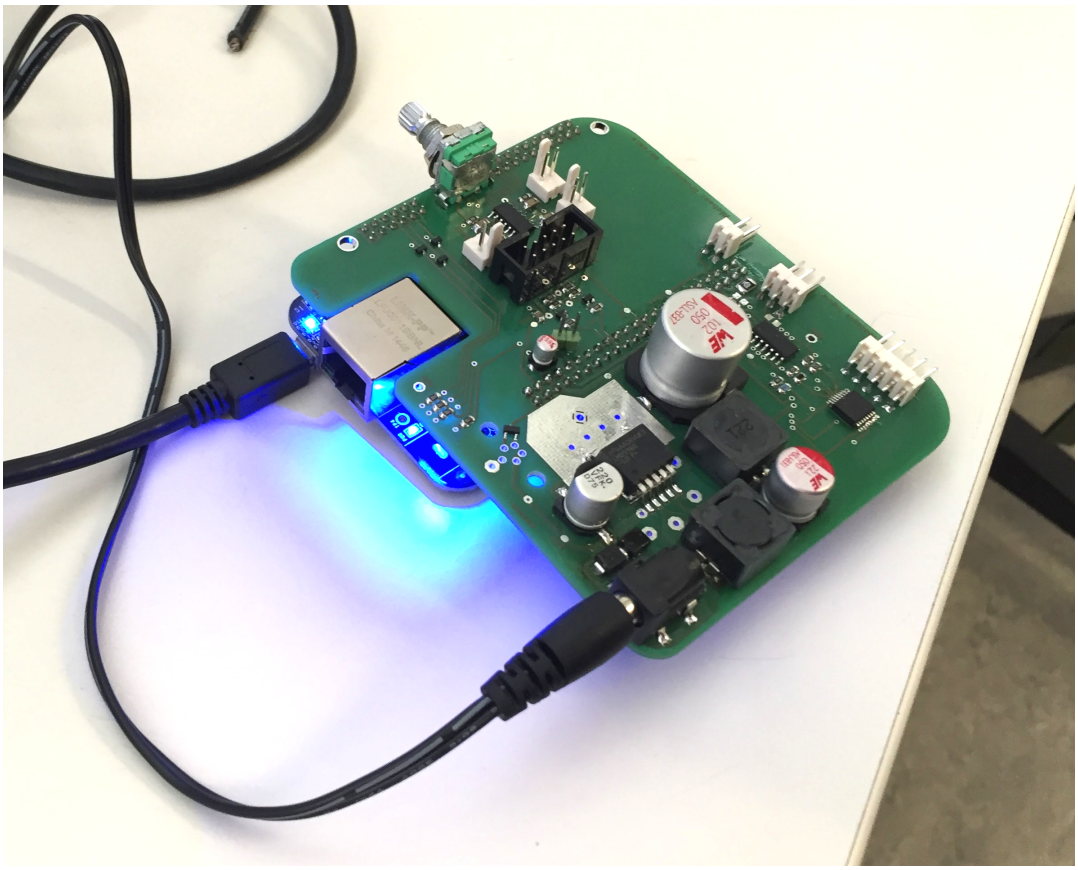


Figure 4: Completed PCB attached to the BeagleBone Black (underneath the left side of the board). Power supply circuitry is on the near side of the board. Behind it and to the right are the analogue inputs and signal conditioning circuitry. Towards the back and the left (from this angle) is the connector to the display and optional buttons, with the rotary encoder poking out to the side. The CAN bus transceiver and other elements are not visible on the bottom side of the board.

2.2 Power Supply

Table 1 outlines the power requirements of the connected devices. Important to note are the supply voltages of the measurement amplifier and the valve. As a result the system was designed from the outset to take a 24V input, with this being passed on to the measurement amp and the valve, while a DC/DC converter takes this down to 5V to run the BeagleBone Black and the other peripheral devices.

	Power requirements	
Device	**Voltage** $[\mathrm{V}]$	**Current** $[\mathrm{mA}]$
BeagleBone Black	5	max. 460
LCD display	5	1.6
Pressure sensor	5	<2
Measurement amp	24	55...75
Valve	17...30	max. 1100

Table 1: Supply voltage and current draw of the connected devices

While the BeagleBone Black has a supply voltage of $5\mathrm{V}$, this is itself regulated further down internally to $3.3\mathrm{V}$. As a result the digital I/O pins of the BeagleBone Black all only accept or put out $3.3\mathrm{V}$. This is particularly relevant to the LCD display, which expects $5\mathrm{V}$ signals over an I^2C bus (see 2.5.1). The BeagleBone Black conveniently provides a pin at which to tap the regulated $3.3\mathrm{V}$ and provide a third power rail to a number of sub-circuits describe later.

Furthermore, the analogue to digital converter of the BeagleBone Black only accepts voltages between $0\mathrm{V}$ and $1.8\mathrm{V}$. Here, again, an analogue voltage reference pin is provided and its use with the ADC is described in 2.7.1.

2.2.1 DC/DC Regulator

There are essentially two types of voltage regulation: linear regulation and switching regulation. In its most basic form a linear regulator uses a series resistor to dissipate excess power as heat. To achieve a stable output voltage independent of the load a variable resistor and optionally a feedback loop is used. Such regulators are practical in their simplicity, but inefficient as excess power is dissipated as heat. If large voltage drops are required, the heat loss can be considerable.

A switching regulator is an active device that achieves an average output by switching the voltage source on and off. A similar feedback mechanism to that found in linear regulators is used to control the output. While more expensive, switching regulators are considerably more efficient than linear regulators, as the only power losses occur during switching and these are small in comparison. Other advantages of switching regulators is that with some clever switching they can generate output voltages higher, or of opposite polarity than the input voltage.

For this device a voltage drop from $24\mathrm{V}$ to $5\mathrm{V}$ was required. Table 1 lists the current draw of the main connected devices. The devices that draw the most current are the valve and the BeagleBone Black itself. The BeagleBone Black can reach its maximum current draw of $460\mathrm{mA}$ during the boot process; while idle, current draw is in the region of $300\mathrm{mA}$ [7]. The valve draws $100\mathrm{mA}$ in mid-position but can reach $1100\mathrm{mA}$ at full stroke. As the valve is not in use during the BeagleBone Black boot process the current draw of the system will, during normal operation, not exceed $1500\mathrm{mA}$, and this only for short periods of time.

Given the above considerations a linear regulator would have to dissipate almost $30\mathrm{W}$ of power at peak load ($1.5\mathrm{A} \cdot (24 - 5)\mathrm{V} = 28.5\mathrm{W}$). This isn't practical for small form-factor devices such as this and would generate too much heat. The better alternative is a switching regulator. Several manufacturers produce switching regulator integrated circuits. For this application the LM2596-5 Simple Switcher® from Texas Instruments was selected. Specifically, this step-down voltage regulator provides a fixed $5\mathrm{V}$ output and is capable of driving a $3\mathrm{A}$ load with excellent line and load regulation. Its comparatively high switching frequency of $150\mathrm{kHz}$ allows for smaller sized filter components, a useful attribute when space is at a premium. [8]

The LM2596 data sheet comes with extensive instructions on selecting the appropriate values for the peripheral components required. A maximum load current of $3\mathrm{A}$ was assumed, and where possible values were selected to minimise output voltage ripple (for example in the selection of the output capacitor). The circuit was subsequently built and tested on perfboard. The final design of the power supply can be seen in Figure 5.

The system receives $24\mathrm{V}$ via a barrel connector from a power supply.

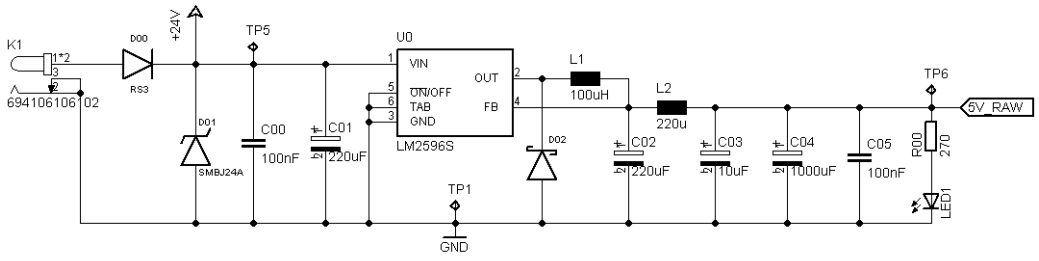


Figure 5: Circuit diagram of the DC/DC regulator and periferal components

The design incorporates several additional elements:

Over voltage protection. This takes the form of a Transient Voltage Suppressor (TVS) diode anti-parallel with the power source (D01). Well laid out TVS diodes are designed to

have negligible impact on the normal operation of a circuit. The breakdown voltage of the TVS diode is usually in the region of 10% greater than the reverse standoff voltage which approximates the circuit operating voltage. When a voltage transient occurs that exceeds the breakdown voltage of the TVS diode, it will limit the voltage spike to the TVS diode's clamping voltage and conduct the transient current to ground [9].

For this device the SMBJ24A $600\mathrm{W}$ TVS diode from Fairchild Semiconductor was chosen due to its combined small package size, peak pulse power capability ($600\mathrm{W}$) and fast response time (typically less than $1\mathrm{ps}$ from $0\mathrm{V}$). The reverse stand-off voltage of the SMBJ24A is $24\mathrm{V}$, the breakdown voltage lies between 26.7 and $29.5\mathrm{V}$, and the peak pulse current is $15.4\mathrm{A}$ [11].

Reverse polarity protection. There are several options for implementing reverse polarity protection. In low voltage systems where a voltage drop can't be afforded, the use of a P-channel MOSFET is common, as its effect on the voltage supply is minimal (Figure 6).

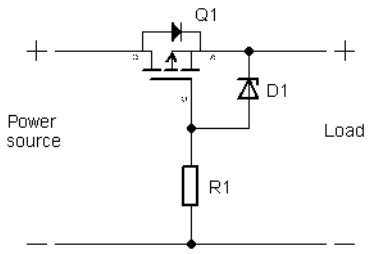


Figure 6: A common protection circuit against reverse polarity

For the system being designed a simpler option is acceptable. A rectifier diode is placed in series with the power source (D00). During normal operation the diode is forward biased and current flows to the circuit. Should the polarity of the voltage be reversed, in other words the diode is now reverse biased, the diode will block and the circuit will see no current. The disadvantage of this simpler solution is that the diode exhibits a forward voltage drop; as a result the circuit will no longer receive $24\mathrm{V}$. The rectifier diode chosen for the circuit (see below) has a forward voltage drop of $1.3\mathrm{V}$, so the circuit can expect to see $22.7\mathrm{V}$ instead of $24\mathrm{V}$. While the nominal operating voltage of the valve is at $24\mathrm{V}$, the operating voltage range lies between $18\mathrm{V}$ and $30\ \mathrm{V}$. Similarly the Measurement amplifier has a nominal supply voltage range of $12\mathrm{V}$ to $24\mathrm{V}$. In both cases the voltage drop caused by the rectifier is of no consequence.

The RS3G Fast Switching Rectifier from Vishay was chosen for its small package size, high forward current ($3\mathrm{A}$) and fast switching time.

Additional voltage smoothing. In addition to the $220\mu\mathrm{F}$ input capacitor (C01) stipulated by the DC/DC regulator data sheet, a $100\mathrm{nF}$capacitor (C00) was added in parallel to remove high frequency input noise. Similarly output ripple and noise were further stabilised with an

additional $220\mu\mathrm{H}$ inductor (L2), $10\mu\mathrm{F}$, $1000\mu\mathrm{F}$, and $100\mathrm{nF}$capacitors (C03, C04 and C05 respectively). The values were chosen so as to suppress a broad sweep of frequencies.

An indicator LED. A simple visual indicator placed at the point at which the $5\mathrm{V}$ produced by the DC/DC regulator enters the BeagleBone Black. The BeagleBone Black should enter its boot procedure once $5\mathrm{V}$ are connected, so if the LED is on and the boot process doesn't occur, a problem can be inferred.

2.2.2 Analogue and Digital Power Rails and Ground Planes

Digital circuitry is noisy, with rapid switching resulting in current transients and electromagnetic interference in the power supply. The logic levels of digital circuits are themselves virtually immune to these effects, however analogue circuitry powered by the same supply can be vulnerable to induced noise. This is particularly important in signal conditioning circuitry where the fidelity of the signal to be measured is important. Similarly, measurements made with respect to ground need a solid ground. Ground planes underneath digital circuits will suffer from switching noise generated by the digital circuit.

To minimise the effect of the digital circuitry on the analogue circuitry a separate analogue ground plane and analogue $5\mathrm{V}$ power rail were introduced. Figure 8 shows how the regulated $5\mathrm{V}$ output from the BeagleBone Black was uncoupled from the $5\mathrm{V}$ analogue power rail. This was achieved with an EMI suppression ferrite bead (L3) and two capacitors ($10\mu\mathrm{F}$ and $100\mathrm{nF}$).

Ferrite beads are passive electronic devices whose impedance phase angle changes with changes in frequency. Initially, at relatively low frequencies a ferrite bead behaves much like an inductor and its reactive impedance dominates. As the frequency rises, so does the ferrite's impedance, which is initially entirely to do with a rise in reactance. However, at some stage there is a frequency dependent rise in resistance, which eventually overtakes the ferrite's reactance. With even greater frequencies the reactance starts to drop off and by the time the ferrite's impedance peaks, it is entirely resistive. See Figure 7 for the typical impedance characteristic of the ferrite bead used in this device.

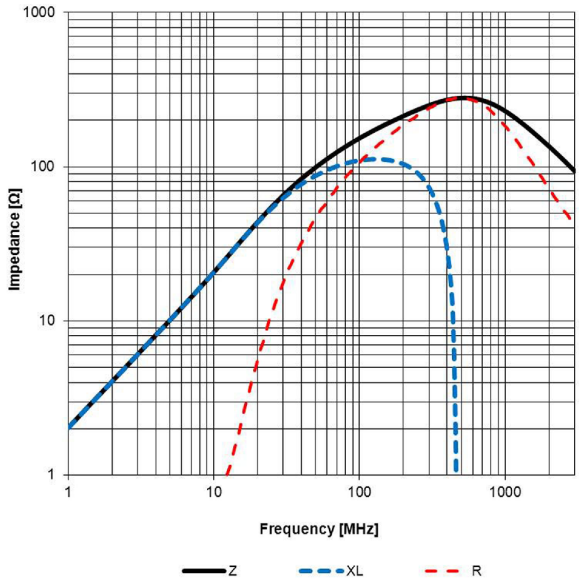


Figure 7: Ferrite bead impedance characteristic (taken from the data sheet of the WE-CBF ferrite bead used in the device)

Ferrites are often used for electromagnetic interference (EMI) suppression applications, where they are placed in series with a power rail. The frequencies to suppress should be well into the resistive region of the ferrite, which causes the high frequency noise energy to be dissipated as heat. [10]

The AM335x of the BeagleBone Black has a clock frequency of $1\mathrm{GHz}$, so a ferrite bead was chosen whose resistance peaks in this region. The ferrite bead is complimented by two capacitors to ground, forming a low pass filter which further suppresses higher frequencies on the supply rail. Space was made for an additional decoupling capacitor (C09) if further filtering is deemed necessary once the board is taken into operation.

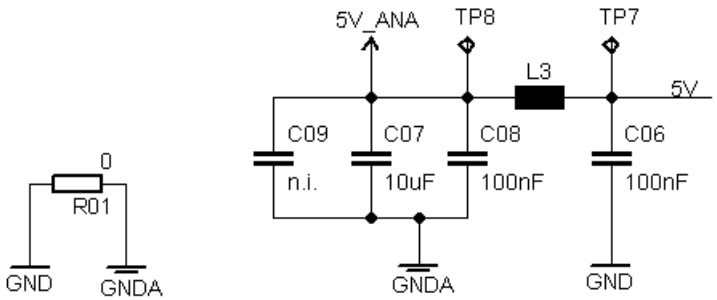


Figure 8: Circuit diagram of the interfaces between the analogue and ground planes (left), and the analogue and digital $5\mathrm{V}$power rails (right)

The circuit of the connection between the digital and analogue ground (Figure 8) is simply a 0Ω resistor. In itself, this means nothing, as the resistor has no effect other than to connect the two grounds. However, in practical terms during PCB layout, this enforces the connection of the two ground planes at a single point. Known as star point grounding, the principle of connecting grounds at a single point ensures that all voltages are measured with respect to a particular point, and not just at an undefined "ground" [13]. In many cases a convenient point at which to connect the ground planes is at the power supply. Figure 9 shows the bottom of the developed PCB, with digital ground occupying most of the board. Analogue ground is in the bottom left quarter of the board. The two grounds are completely separate, except for the 0Ω resistor which lies at the top of the board, analogue ground trace that runs up the left hand side of the board. The resistor sits next to the power source and defines the common, defined ground between the two sides of the circuit.

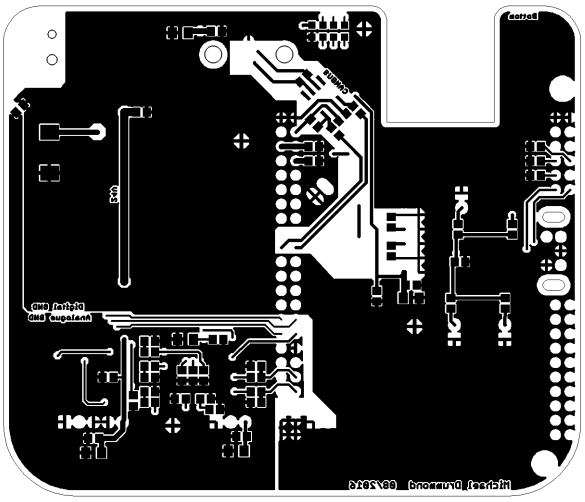


Figure 9: PCB layout showing the ground planes on the bottom of the board

2.3 Rotary Encoder and Buttons

2.3.1 The Rotary Encoder

Control and navigation through the device options (see 3.4) is enabled with the use of a rotary encoder. An encoder or jog wheel was considered a more intuitive interface for setting the valve than a series of buttons, and had the added benefit of giving the device a cleaner appearance.

The rotary encoder used here is a quadrature encoder, in other words it uses two output

channels (A and B) which are 90 degrees out of phase with each other. The square wave output of the two channels can be used to track both position and direction of rotation.

The Texas Instruments Sitara processor on the BeagleBone Black includes an Enhanced Quadrature Encoder Pulse (eQEP) Module as part of its Pulse-Width Modulation Subsystem (PWMSS). The eQEP simplifies the implementation of an encoder by decoding its input and making it available to the programmer as a single integer value that is incremented or decremented accordingly. In terms of hardware, the two encoder signal lines simply need to be connected to the correct two pins on the BeagleBone Black, and ground to ground.

The LCD display on the top face of the device and the device's overall compact size meant placing the encoder next to the display was no longer possible. To make its use ergonomic the rotary encoder was moved to the right hand side of the device, and is thus angled at 90° to the PCB.

2.3.2 The Buttons

The rotary encoder has a built-in push button switch. This momentary switch allows for the selection of options during the navigation of the GUI. As the device was to include the option of expanding its initial functionality, space for a further three switches was made by including three two-pin Molex connectors. The switch and the three headers are connected at one end to ground, at the other to the $3.3\mathrm{V}$ power rail (via a pull-up resistor) and a GPIO pin of the BeagleBone Black via a debounce circuit.

The debounce circuit takes the form of a passive, single pole, low pass filter ($\tau = 1\mathrm{ms}$) followed by a Schmitt trigger (Figure 10). The filter slows and smooths the switch signal, and when combined with the hysteresis of the Schmitt trigger leads to sharper transitions in the signal arriving at the GPIO pins of the BeagleBone Black.

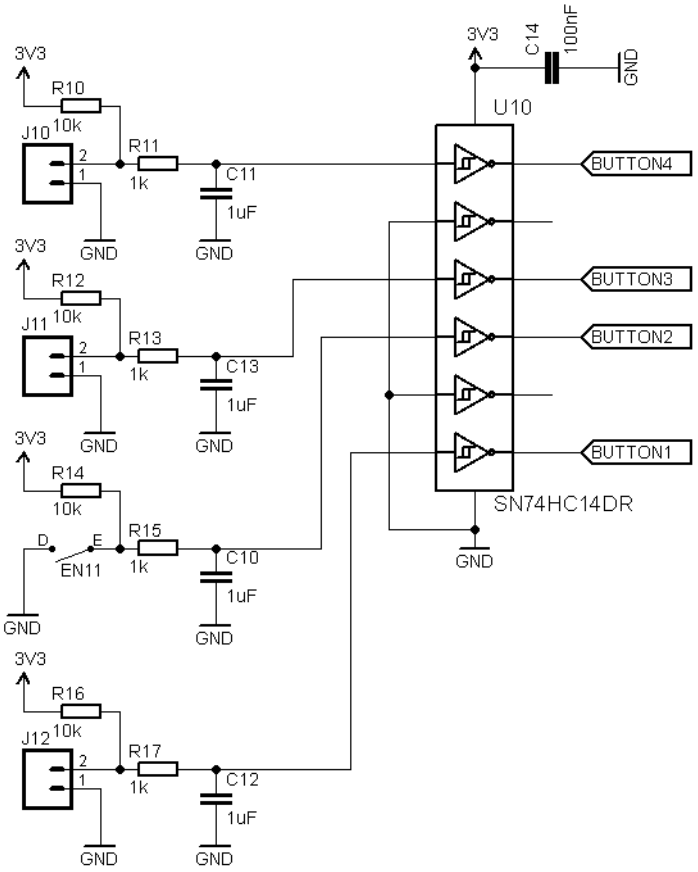


Figure 10: Debounce circuitry of the buttons. Note the encoder connected to button 2.

2.4 LEDs

In addition to the power supply LED (see 2.2.1) a further three LEDs were added to the device to provide someone programming the device with a simple debugging tool or a user with a visual cue that the system is working. These were connected indirectly to individual GPIO pins via N-channel MOSFETs. This adds a level of protection to the pins, should there be excessive current drawn (Figure 11).

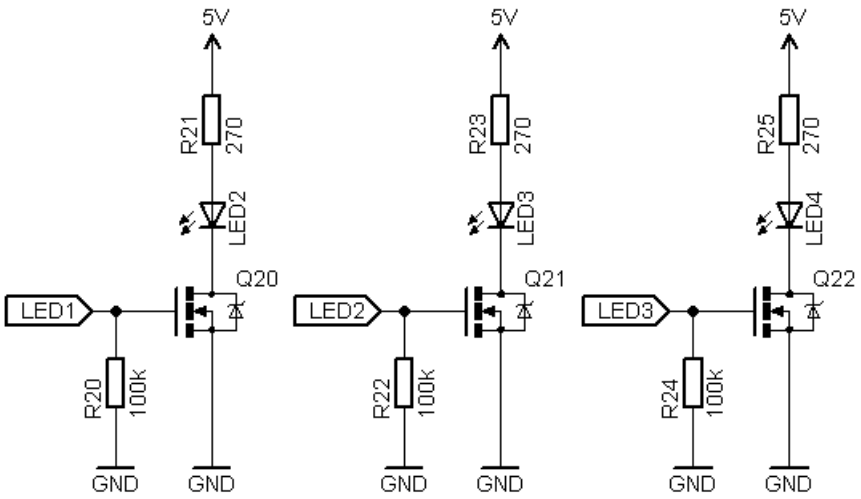


Figure 11: The connected LEDs

2.5 Liquid Crystal Display

The liquid crystal display (LCD) of the device is a 4 line, 20 character, blue backlit display. The LCD's Hitachi HD44780 controller is connected on-board with an I^2C interface, cutting down the number of pins required to control the display significantly (from 12 pins in four-bit HD44780 operation, to four pins for I^2C). Supply and logic level voltage of the display are $5\mathrm{V}$.

2.5.1 I^2C and Level Shifter

Logic level voltage of the BeagleBone Black pins is $3.3\mathrm{V}$, so a bi-directional level shifter circuit was placed between the I^2C data (SDA) and clock (SCL) pins and those of the display (Figure 12). The level shifter ensures that the $3.3\mathrm{V}$ logic signals from the BeagleBone Black are read as $5\mathrm{V}$ signals at the display.

The circuit involves an n-channel MOSFET (Q30 and Q31) placed on each of the signal lines (SDA and SCL), with source connected to the $3.3\mathrm{V}$ bus line and drain to the $5\mathrm{V}$ bus line. These are flanked by pull-up resistors to the two power rails (R30 to R33). The gates of both MOSFETS are connected to the lower voltage supply rail ($3.3\mathrm{V}$).

In this configuration, if neither side is pulled down, the pull-up resistors result in $3.3\mathrm{V}$ at one end of the bus line and $5\mathrm{V}$ at the other. Should the $3.3\mathrm{V}$ end of the bus line be pulled down, $\mathrm{V_{GS}}$ will rise, the MOSFET conduct and the $5\mathrm{V}$ end of the the bus will also be pulled down.

Equally, should the $5\mathrm{V}$ end of the bus be pulled down, the body diode of the MOSFET will conduct, again drawing $\mathrm{V_{GS}}$ down until the MOSFET conducts and the bus line is pulled down to the same level at both ends. [12]

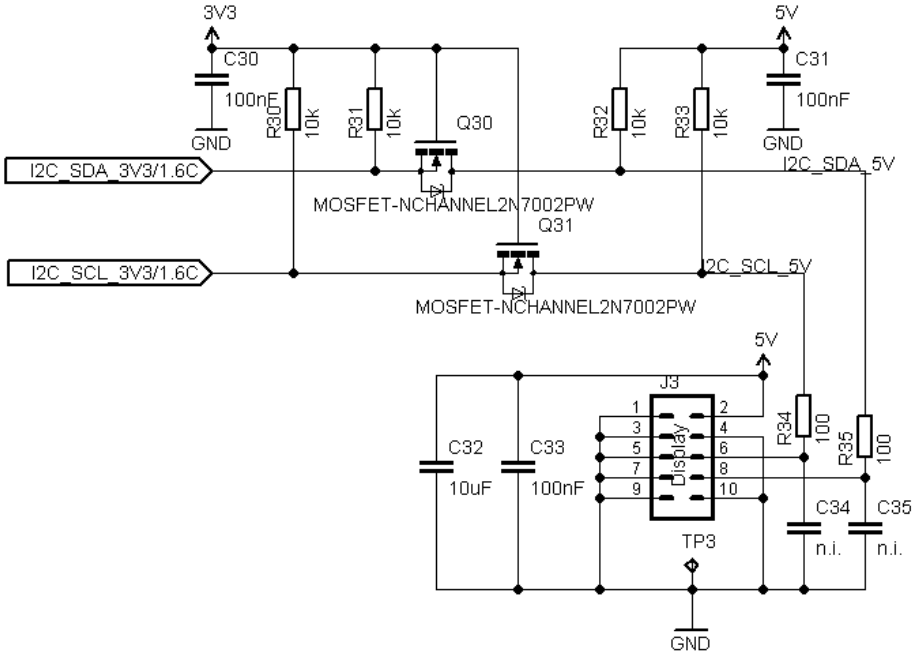


Figure 12: The logic level shifting circuit between the $\mathrm{I^2C}$ pins of the BeagleBone Black and those of the display

2.6 CAN bus

The AM335x processor on the BeagleBone Black incorporates a module for accessing a Controller Area Network (CAN) bus. The DCAN module provides support for CAN protocol version 2.0 at bit rates up to 1MBit/s [14]. The AM335x includes two instantiations of the DCAN controller (DCAN0 and DCAN1), both are connected to pins on the P9 header of the Beaglebone Black [7]. Due to the high frequency switching on the RXD and TXD lines of the CAN controller these should not be placed over a ground plane. The RXD and TXD of DCAN1 are connected to pins 24 and 26, but their use would interfere with the ground plane that extends from the left to the right side of the board. The ground plane is itself constrained by the analogue circuitry at the bottom left of the board. For this reason, DCAN0 with its RXD and TXD lines connected to pins 19 and 20 was chosen to provide the CAN controller.

2.6.1 CAN Transceiver

Additional hardware, in the form of a transceiver, is required for the connection to the physical CAN bus [15]. As the RXD/TXD pins of the BeagleBone Black operate at $3.3\mathrm{V}$an appropriate $3.3\mathrm{V}$CAN bus transceiver was needed. For this the Texas Instruments SN65HVD232 was chosen (U40, Figure 13). Following the data sheet layout guidelines, serial resistors (R40 and R41) were placed to limit current on the digital lines [16]. An optional pull up resistor (R42) was used to drive the recessive input state of the transceiver. High speed CAN (see ISO 11898-2 [17]) requires termination resistors at both ends of the transmission lines. The resistors match the nominal impedance of the cable, which the CAN standard specifies as 120Ω. R43 is such a termination resistor, placed as close to the CAN High and CAN Low pins of the transceiver as possible. In addition, a bypass capacitor was placed as close as possible to the supply pins of the transceiver (C40), and bulk and bypass capacitors were placed on the $24\mathrm{V}$ supply line of the valve (C41 and C42).

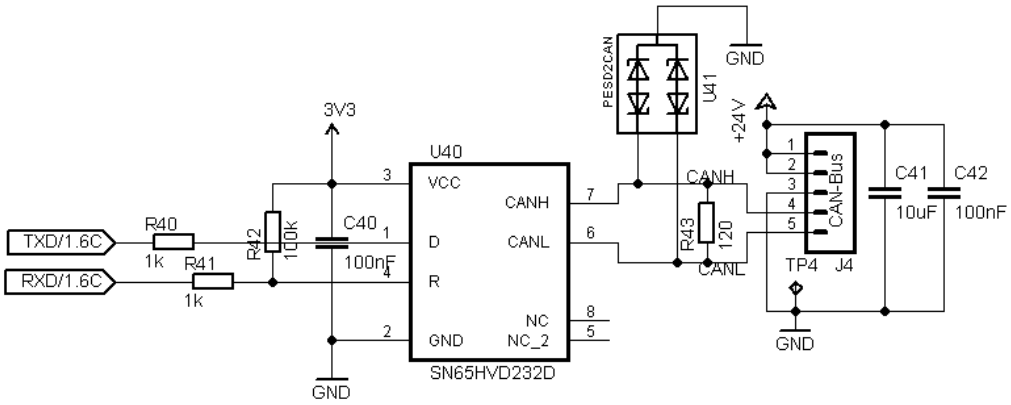


Figure 13: CAN transceiver, ESD protection diodes, and circuitry connecting the on-board CAN controller with the CAN bus

2.6.2 CAN Bus Circuit Protection

The on-chip electrostatic discharge (ESD) protection of the CAN transceiver is good enough for many applications, but insufficient for electrical fast transients (EFT) and surge transients occurring in industrial environments [16]. While this device is unlikely to leave the laboratory, good practice in designing a robust and reliable bus node was followed and additional circuit protection was added. This took the form of a CAN bus ESD protection diode from NXP Semiconductors (PESD2CAN, U41 in Figure 13). This was placed close to where the bus

connects with the board to prevent the transients from propagating on the PCB.

2.7 Analogue Inputs

The analogue inputs are core elements of the device. Input from the pressure sensor is used in the control function to find the pressure set point selected by the user. The accurate setting of the muscle is conditional on the accuracy of the feedback response measured by the pressure sensor. Once the set point has been reached, the device takes a force measurement from the force sensor. This data underpins the measurement of the static force characteristic of the PMAs, and so its accuracy is likewise essential.

The AM335x processor of the BeagleBone Black includes a subsystem with an 8-channel general-purpose analogue-to-digital converter (ADC), with optional support for interleaving touchscreen conversions for a resistive panel (abbreviated to TSC_ADC_SS) [14].

The subsystem comprises of a single 12 bit Successive Approximation Register (SAR) ADC that is multiplexed to the 8 channels. The ADC will measure voltages between 0 and $1.8\mathrm{V}$. The $1.8\mathrm{V}$ supply that drives the ADC is provided as a voltage reference on a pin of the BeagleBone Black, as is an analogue ground pin. Both the pressure and force input signals require conditioning to bring them within the voltage span of the ADC. In both cases this was achieved with a simple voltage divider (Figures 15 and 16).

The ADC has a maximum sampling rate of $200\mathrm{kHz}$. This is significantly faster than the sub kilohertz sampling rate anticipated for the device, even if measurements are taken from two channels. The sampling frequency determines the input impedance of this ADC, so with both pressure and force measurements each taken at $100\mathrm{Hz}$ the input impedance equates to [18]

$$\frac{1}{65.97\times10^{-12}\ \cdot\ 200\mathrm{Hz}} \approx 75.8\mathrm{M}\Omega$$

While this input impedance may be sufficient given the device's current application, a change (for example) in measurement frequency could easily bring the introduced error caused by input overloading to an unacceptable level. For this reason a unity-gain buffer was placed between the voltage dividers and the ADC input (Figures 15 and 16). This impedance matching was implemented with the MCP6004 quad op-amp IC from Microchip Technology inc.

A further consideration was the small capacitor that the sample and hold element of the

ADC uses. This can cause problems for the buffer's op-amp at high sampling frequencies. If the dynamics of the op-amp are too slow, the capacitive load on the op-amp can result in settling times that can skew the result. To deal with this, a simple RC filter can be placed between the buffer output and ADC input. Space for such filters were made in the design, but given the low sampling frequencies planned for this device, they were not instantiated.

2.7.1 Voltage Reference

The BeagleBone Black $1.8\mathrm{V}$ ADC voltage reference pin was used to drive four op-amps required for impedance matching in the signal conditioning circuitry of the analogue inputs (described above). To prevent the circuit drawing a current that could damage the BeagleBone Black, the output of the voltage reference pin was itself buffered, as in Figure 14. The component used here was a MCP6001 single op-amp, supplied by the $5\mathrm{V}$ analogue rail. In addition to a $100\mathrm{nF}$ bypass capacitor, a $10\mu\mathrm{F}$ bulk capacitor and 5Ω resistor were placed at the end of the $5\mathrm{V}$ supply to prevent any drops in the supply and maintain an accurate reference.

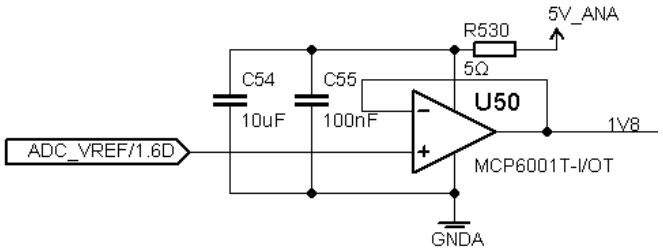


Figure 14: VREF

2.7.2 Pressure Sensor

The $1.8\mathrm{V}$ voltage reference supplies the quad op-amp IC, of which one op-amp is used in the signal conditioning of the pressure sensor input (Figure 15). The pressure sensor is supplied by the analogue $5\mathrm{V}$ rail, filtered and supported by bypass and bulk capacitors. The sensor output is a value between 0 and $5\mathrm{V}$ which is paired down in a voltage divider to bring it to between 0 and $1.8\mathrm{V}$. Resistors with 0.1% tolerance were selected to improve accuracy. The voltage divider is followed by a unity gain buffer to match the divider's impedance to that of the analogue input pin of the BeagleBone Black (described above). The total resistance of the divider was aimed in the region of $150\mathrm{k}\Omega$. The relatively high resistance prevents the sensor measurement from being negatively impacted by a high current draw. Moreover, the

unity gain buffer negates the problems the high resistance of the divider might cause with the ADC input.

The pressure sensor was connected with AIN0 of the BeagleBone Black.

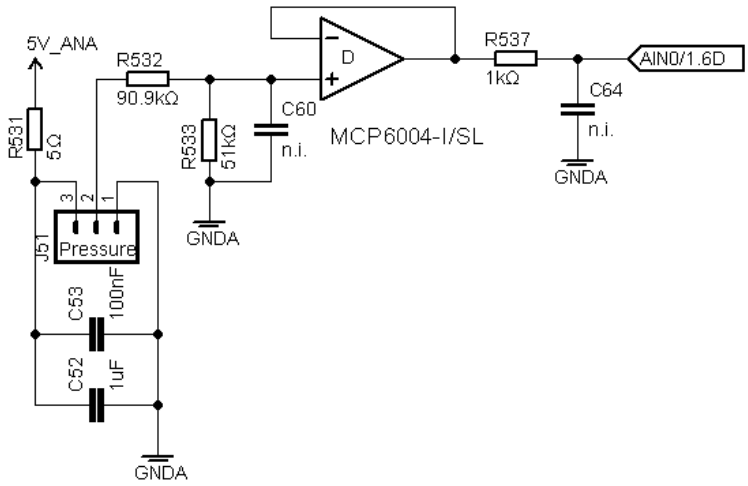


Figure 15: Pressure

2.7.3 Force Sensor

The force sensor use the rig was a KD9363S from ME-MeSSsysteme GmbH with a full scale nominal load of 1 $\mathrm{t}$and an output of $3\mathrm{mV/V}$at full scale. This was supplied with $5\mathrm{V}$ by a GSV-1A measurement amplifier, also from ME-MeSSsysteme GmbH. The amplifier's sensitivity is $2\mathrm{mV/V}$.

The signal conditioning circuitry connecting the force sensor to the BeagleBone Black is built on a similar set-up to that of the pressure sensor. That is to say, a voltage divider pairs the signal down to between 0 and $1.8\mathrm{V}$ and a unity gain buffer matches the impedance to that of the input of the ADC. However, several further additions enhance the accuracy of the measurements (Figure 16).

The output voltage of the force sensor's measurement amplifier is -10 to $10\mathrm{V}$. A positive voltage output indicates a tensile force on the strain gauge, while a negative voltage a compressive force. For this application, only tensile forces need to be measured and are expected. So, while the signal first passes through a voltage divider in which both positive and negative voltages are paired down, the unity gain buffer that follows will limit the signal to positive voltages. Thus, only voltages between 0 and $1.8\mathrm{V}$ (representing the positive

output range of the measurement amplifier) will be passed on to the BeagleBone Black's ADC. This maximises the measurement resolution by ignoring the negative voltages.

The PMAs to be tested by the rig are available in different diameters, from $5\mathrm{mm}$ to $40\mathrm{mm}$. The larger the diameter of the PMA, the greater the force that it is able to exert. So, while the force sensor and external measurement amplifier are set up to read up to the $6\mathrm{kN}$ that the largest muscle can produce and convert it to a voltage between -10 and $10\mathrm{V}$, the maximum voltage output will only ever be reached by the very largest muscles. It is, therefore, only the force measurements of these PMAs that will use the full span of the 12 bit ADC on the BeagleBone Black, and that of the smaller PMAs will have a corresponding loss in resolution. To compensate for this, three voltage dividers with differing ratios were implemented. A CMOS analogue switch (U52, a Maxim MAX4662) placed before the dividers allows the user of the device to choose between them. During use only one switch will be "on" at any given time, and the signal will pass through this divider to be read by one of the ADC channels.

The voltage dividers themselves are composed of 0.1% tolerance metal film resistors. To improve accuracy and their flexibility in use, each divider is made up of two pairs of resistors in series. In so doing, sourcing appropriate resistor values for the divider ratio was made easier. The voltage divider ratios, and the switches they are connected to, are listed in table 2. The user will select the input path based on the maximum voltages expected. For any given signal input path the dividers will convert the signal from between $0\mathrm{V}$ and the respective maximum input voltage to between 0 and $1.8\mathrm{V}$.

	Voltage divider	
Analogue switch	**Max. input** $[\mathrm{V}]$	**Ratio**
Switch 1	3.3	$1.8\overline{33}$
Switch 2	10	5.55
Switch 3	6	3.34

Table 2: Analogue force sensor voltage dividers. Output voltage of the dividers is in all cases between 0 and $1.8\mathrm{V}$given an input between $0\mathrm{V}$ and the maximum input of the signal path chosen by the switch.

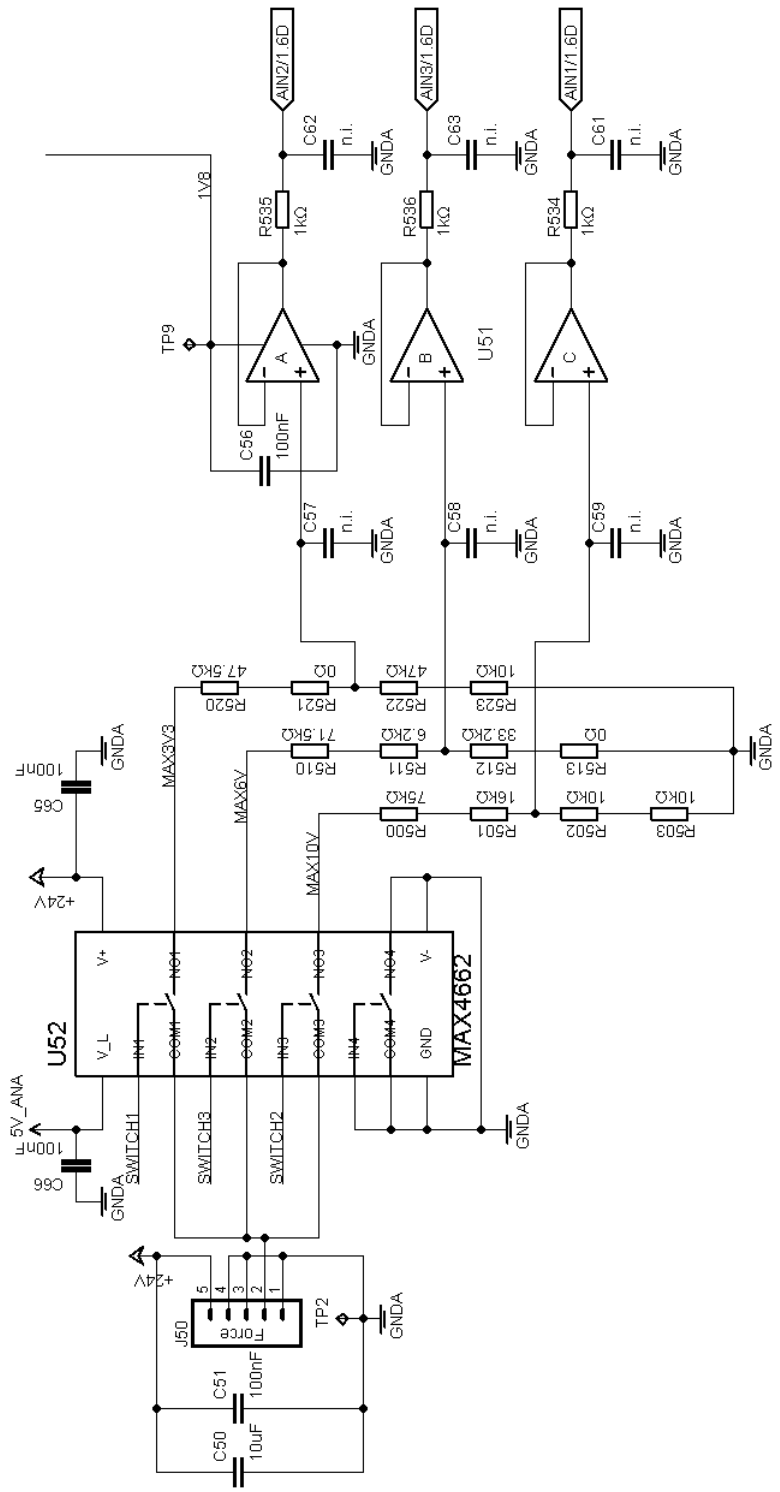


Figure 16: Force

3 Software Development

3.1 Overview

The BeagleBone Black can run a fully fledged Linux operating system, courtesy of the Texas Instruments AM3358 ARM Cortex-A8 processor. Debian 8.6 (Jessie) was installed on the Beaglebone Black, and a cross-compilation environment to program in C++ was set up according to the instructions in chapter 7 of [19]. This involved the Eclipse IDE for C/C++ Developers (version: Mars.1 Release 4.5.1) running on a Debian remote machine. The software was compiled with the Cross GCC Compiler with the -std=c++11 compiler flag to enable support for the C++11 standard. As POSIX threads were used in this project and the pthread library is not included by default by the GCC compiler, it was enabled with -lpthread in the linker options.

The development of the software can conceptually be divided into the following parts. The control system and measurement routine are central to what the device does, while the user interface and data handling facilitate its use. The control system has a PI-control function at its core, which uses ADC and user input values to calculate output values for the valve. Once pressure levels are reached a force measurement is taken and stored. Which pressure values are set, the lengths the muscles are set to, and how often measurements are taken, are part of the measurement routine. The user interface revolves around a menu that pulls together the LCD display with the encoder and button input.

The user has access to, and can modify, the measurement routine via the user interface. The control system and data handling, however, remain largely hidden.

3.1.1 Software Architecture

The UML (Unified Modeling language) class diagram in Figure 17 shows the class structure of the test rig software. Broadly speaking, the upper half of the diagram represents the classes and functions responsible for the user interface (menu, encoder, button and LCD). While the lower half depicts those elements responsible for the control system and data handling. The top-level functions in *MenuStructure* form that part of the system that a user directly interacts with, and includes the measurement routine. These functions, and the underlying framework supporting the menu and its use, are described further in Section 3.4. This is accompanied by a state diagram of the menu.

The *DeviceControl* class and its role in bringing the relevant aspects of the control system together is described in Section 3.3. Section 3.3.3 describes how this class manages the data used and generated by the test rig. An API of part of the software can be found in Appendix E.

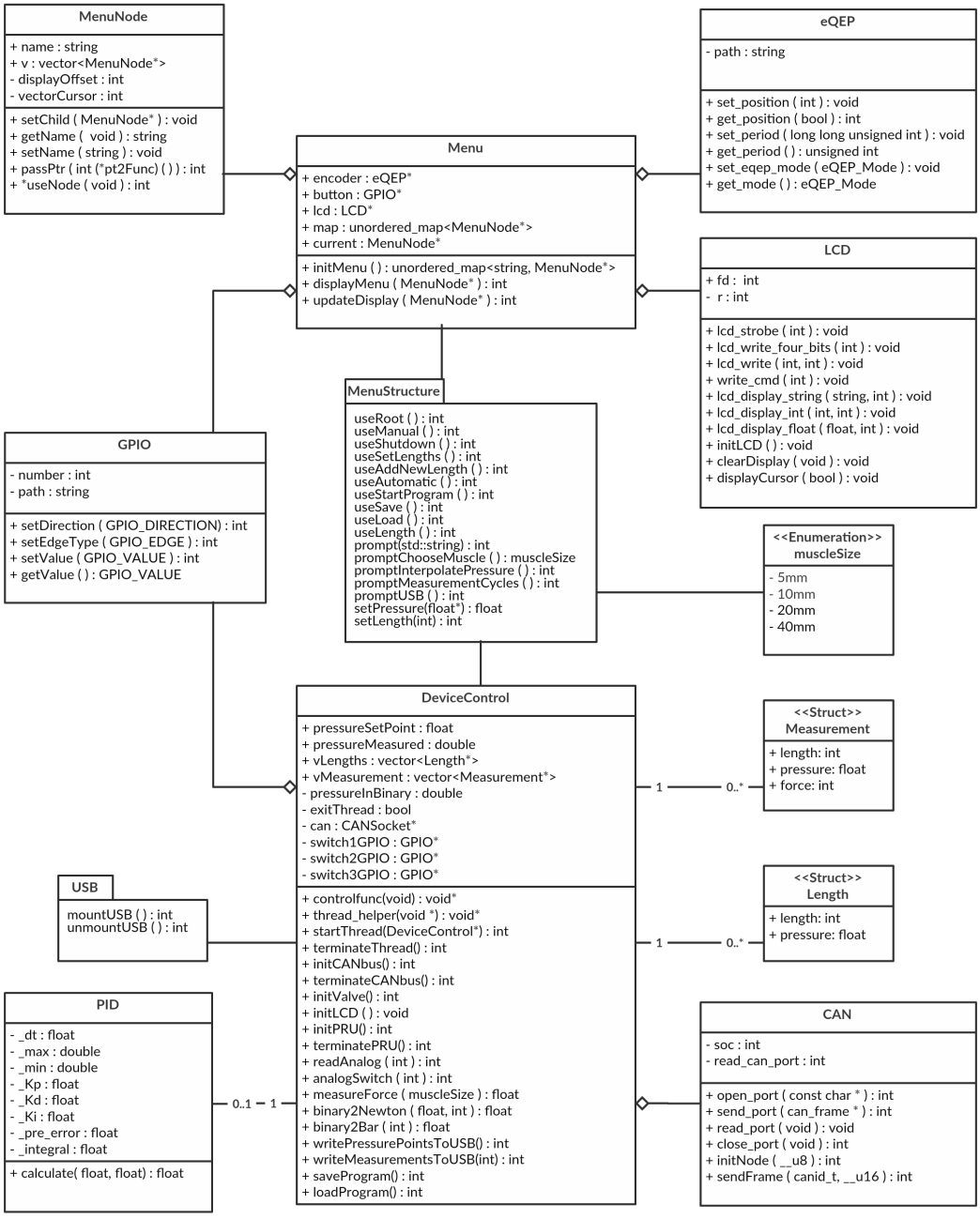


Figure 17: A UML class diagram indicating the classes, their attributes and functions, used in the test rig. The UML package symbol is used here (MenuStructure and USB) to represent collections of top-level functions that do not belong to a class.

3.1.2 Control System and Programmable Real-Time Units

A standard Linux kernel, such as the one that Debian is built on, is a preemptive kernel. Preemptive scheduling has many advantages, but it cannot execute tasks in real time. This is problematic in implementations of discrete-time PID controllers that rely on precise time intervals.

The BeagleBone Black's AM3358 includes a Programmable Real-Time Unit Subsystem and Industrial Communication Subsystem (PRU-ICSS), which provides a convenient way to implement real-time functionality where it is needed. The PRU-ICSS can be thought of as two 32 bit on-chip microcontrollers (known as PRUs) running at $200\mathrm{MHz}$. They have full access to the internal memory of the processor and it's peripherals, and single-cycle IO access to a number of pins (but not all). So, for example, an initial mock-up of the test rig had a PRU reading pressure values from the ADC and outputting a Pulse Width Modulation (PWM) signal to a valve. The PRU has single-cycle access to the ADC values, and the software defined PWM output used a GPIO pin.

The PID controller would have been implemented on one of the PRUs with the help of an on board single-cycle 32 bit multiplier, however, external factors led to the PWM valve being replaced with a CAN-bus valve. The PRUs don't have direct access to the DCAN module that performs CAN protocol communication on the AM3358, so a hard real-time set-up of the control system fully implemented using the Programmable Real-Time units was no longer possible.

To keep the control system simple, yet keep some of the benefits of the PRU-ICSS, the control system was programmed in Linux space, while an accurate timer was implemented using one of the PRUs. The PRU generates an interrupt at precise time intervals, that alert the controller running in a high priority thread in Linux space to calculate a new output value.

3.2 Device Tree Overlays

The device tree is a data structure for describing hardware, that obviates the need to hard code details of a device into an operating system. Aspects of the hardware can be described in the device tree, which is passed to the operating system at boot time. Device Trees have become mandatory for all new ARM SoCs, and Debian Jessie (used on this device) is built on the Linux kernel version 3.16, which implements the Flattened Device Tree (FTD) model [20]. To enable runtime configuration of inputs and outputs on the BeagleBone Black,

a system of Device Tree Overlays (DTOs) and a cape manager exist [21].

The following device tree *.dts* files were programmed, compiled as device tree binary overlays (*.dtbo*), and placed in the /lib/firmware folder of the BeagleBone Black. At boot, the system loads the *.dtbo* files to extend the device tree. Backup copies of these files were placed in /home/⟨username⟩/dts. An example .dts file (GPIOs.dts) is in Appendix D.

- **ADC.dts** Enables the use of the ADC channels 0 to 6. Pins 32 to 40 of the P9 header are exclusively used by the ADC. These pins include the voltage reference and analogue ground (described in 2.7) along with the 7 aforementioned input channels.
- **DCAN0.dts** Enables one of the DCAN controller modules that performs CAN protocol communication on the AM3358. DCAN0 was enabled for PCB design layout considerations (see 2.6). RXD and TXD of DCAN0 are multiplexed to pins 19 and 20 of the P9 header of the BeagleBone Black. As the I^2C2 bus is set to default on these pins, it was first necessary to modify the device tree binary that is used at boot (am335x-boneblack.dtb). This involved decompiling the binary, commenting out the section related to I^2C2 (starting line 78), recompiling and rebooting.
- **eQEP2.dts** The Enhanced Quadrature Encoder Pulse (eQEP) Module (described in 2.3.1 of the AM3358 chip provides an easy way to implement a quadrature encoder. Two of these modules are provided for on the BeagleBone Black; both on the P8 header. The rotary encoder was placed on the right-hand side of the device necessitating the removal of pins 13 to 26 of the P8 header. While both eQEP modules were still accessible, the placement of the buttons and their connection to pins 27 to 30, meant connecting the encoder to eQEP2 was the parsimonious choice.
- **GPIOs.dts** Enables the pins 7 to 9 on the P8 header as outputs for the LEDs. In addition, pins 27 to 30, also on the P8 header, are configured as inputs for the buttons. Four inputs for buttons were made available, three connected to Molex headers that were not used and the fourth to the encoder button (see Section 2.3). All seven pins are pulled down with no signal.
- **I2C1.dts** With I^2C2 disabled to allow for DCAN0, and I^2C0 not exposed on the expansion headers, only I^2C1 was available to communicate with the display. The data (SDA) and clock (SCL) lines of the I^2C1 module are available either at pins 17 and 18 or pins 24 and 26 of the P9 header. To maximise space for the ground plane (see 2.2.2) the module was connected to pins 24 and 26.
- **PRU.dts** The Programmable Real-Time Units are also enabled over a device tree

overlay. In this case both PRUs were enabled, although only one is used. Furthermore, a debug pin (pin 27 on header P9) was enabled to help verify the timer instantiated on the PRU.

3.3 Control System and Data Handling

The *DeviceControl* class (Figure 17) controls access to the device specific elements of the test rig. The control system sub-systems include reading the ADC, placing packages on the CAN bus, and the PID controller itself. The *DeviceControl* class includes functions to initialise and control these elements, as well as thread management functions, to run the controller in a separate thread to the main program. In addition, the *DeviceControl* class includes data structures and functions to handle the user defined and generated data.

At program start an object of class *DeviceControl* is instantiated and a POSIX thread is started with the *startThread()* member function. This creates a thread running with a Round Robin schedule policy and a high thread priority. Portable Operating System Interface (POSIX) threads, or pthreads, are part of a family of standards specified by the IEEE [22]. It is a parallel execution model that exists independently of a language and allows a program to control multiple different flows of work that overlap in time.

On creation, the thread is passed a pointer to the function *controlfunc*, which instantiates an object of class *PID* and runs the control system in a loop. The thread will exit the loop and terminate when the global *DeviceControl* member variable *exitThread* is set to true. This only happens at shutdown.

3.3.1 PID Controller with PRU Timer

A proportional-integral-derivative (PID) controller is a control loop feedback mechanism that repeatedly calculates the error between a measured process variable and a desired set point. The controller modifies the output based on the proportional error, the integral of the error over time, and the derivative of the error. The original analogue PID controller can be represented as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt},$$

where e is the error at time t and K_p, K_i, and K_d are the coefficients of the proportional,

integral and derivative terms.

For a digital system various methods exist for creating a discrete time version. For the purposes of this device the digital PID controller used can be expressed as follows:

$$u(k) = K_p e_k + K_i \sum_{n=1}^{k} e_n \Delta t + K_d [e_k - e_{k-1}] \Delta t,$$

where, k is the most recent sample and Δt is the sampling period.

The controller runs in a loop in the aforementioned thread, and this is paused in each iteration of the loop. The thread waits for an interrupt from the PRU timer with *prussdrv_pru_wait_event()* and only calculates a new output once the interrupt is triggered. The PRU timer was programmed in Assembler and sets the interrupt exactly every $10\mathrm{ms}$, which results in the control algorithm running with this period. The reading of the ADC, running the control algorithm and placing the new output value on the CAN bus occurs significantly faster than $10\mathrm{ms}$. That these actions are faster than the time interval is important, as it means the actions that are undertaken on every iteration of the control loop are happening at the defined time interval and not simply forever playing catch-up. That they are significantly faster, means that for practical purposes we can assume that these actions are all taking place at the instant that the interrupt is being triggered.

The function *calculate()* in the class *PID* calculates and normalises the error between set point and measured pressure, assuming a maximum pressure of 8 bar. The P, I and D terms, and finally a new output are calculated. This is then denormalised and fit to the 12 bit input value of the pressure valve, before being returned and placed on the CAN bus.

The control loop was tuned manually and the establishment of P, I and D terms was undertaken largely by trial and error. The proportional gain was set to 0.7, the integral gain to 1 and the derivative gain to 0. Figure 18 shows the effect of the integral term on the step response (proportional gain held at 0.7 and no derivative term). The derivative term is useful in control loops where an overshoot is highly undesirable. In these cases it helps to dampen the effect of the proportional and integral terms, but this isn't particularly the case here. Furthermore, it can have undesirable effects in fast acting loops (such as flow and pressure loops) as its effect is strongest during the greatest change. In these instances it can be difficult to tune the controller well, it is susceptible to noise, can be unstable and brings few benefits.

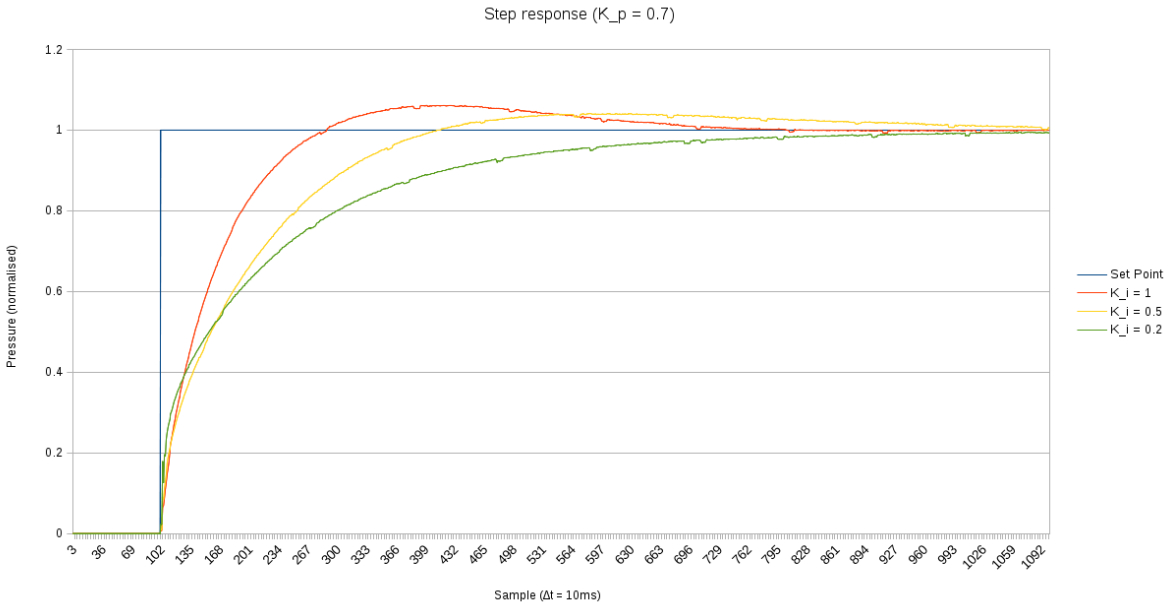


Figure 18

3.3.2 CAN bus

To access a CAN bus the BeagleBone Black uses SocketCAN, a set of open source Linux CAN drivers and networking stack. In addition, the *can-utils* package was downloaded to the device, as it provides some userspace utilities to facilitate the use of the SocketCAN subsystem [23].

Details of the proprietary Festo CAN protocol used by the pressure valve was obtained from an internal data sheet [24]. The example at [25] was modified and extended to cover the Festo protocol, and wrapped in a class. The *CAN* class is instantiated at start up and passed to the instantiated *DeviceControl* object.

3.3.3 ADC and Data Handling

The user of the device selects which muscle diameter is to be tested as described in Section 3.4 and Figure 19. This choice switches the input path of the force sensor signal between the three voltage dividers. For this, *DeviceControl* has three *GPIO* pointers which are instantiated in its constructor and control the analogue switch (hardware described in Section 2.7.3). In addition, the three input paths connect to separate ADC channels, so when a call to the ADC is made the correct input channels must be selected.

Analogue to digital conversion of the pressure sensor signal is undertaken with the *DeviceControl* member function *readAnalog()*, which reads the ADC channel input via a file stream. The value is first converted to a value in millibar with the help of the member function *binary2Bar()* before being passed, together with a pointer to the pressure set point, to the *PID* member function *calculate()*.

The set point and measured pressure are both stored as public member variables of the *DeviceControl* class (*pressureSetPoint* and *pressureMeasured* respectively), and accessed by the *PID calculate()* member function.

DeviceControl contains two vectors *vLengths* and *vMeasurements* that store instantiations of structs *Length* and *Measurement* respectively. A *Length* struct stores the pressure and length values selected by a user (see 3.4.2. The *vLengths* vector forms the basis of the measurement and is reflected in the *Set lengths* menu. The fixed, predefined lengths allowed by the frame of the test rig means, that for any given muscle there will be no great variation in the *vLengths* vector from one measurement to another.

The *vMeasurments* vector is populated immediately prior to a measurement being undertaken. The *Measurement* struct contains *pressure* and *force* fields. The user is asked to define how many additional pressure points should be measured between each length. This enables an increased resolution of the resulting static force characteristic of the PMA. This information and the *Length*s from *vLengths* are used to generate *Measurement* structs in *vMeasurements*. During the measurement routine the *force* fields are filled by the measured forces.

Following a measurement, the stored pressures, lengths and forces are moved to the *data.txt* file on the USB stick. The first line lists the pressures at which a force measurements were made, with tabs separating the values. Subsequent lines start with a length value followed by the force measurements at this length. If multiple measurements for a given length were made, these are grouped together. Otherwise, the lines of measurements are in the order of the lengths in *vLengths* and by extension the *Set lengths* menu vector.

3.4 Menu and User Interface

A menu was created to allow a user of the device to select and change parameters of the test rig. A *Menu* class acts as a container for a class of menu nodes (class *MenuNode*, described in *menu.h* and *menu.cpp*), in which instantiated *MenuNode* objects contain a vector of pointers to further *MenuNodes* (simply named *v*). These "child" nodes each have their own

MenuNode vectors, in which, by convention, the first element is the "parent" *MenuNode* in whose vector the child resides. Each *MenuNode* has a *useNode()* function that is defined with a function pointer at object instantiation. In this way, a function defining what should happen when the node is "used" can be passed to the *MenuNode*.

Conceptually, one can differentiate between internal and terminal nodes, although this distinction isn't made in the software. Internal nodes will, when their *useNode()* function is called, display the contents of their *MenuNode* vectors, allowing the user to navigate further into the menu. In contrast, terminal nodes will display settings that can be changed by the user.

Besides a *name* variable and the aforementioned vector, *MenuNode* objects contain the member variables *vectorCursor* and *displayOffset*. The *vectorCursor* variable is essentially an iterator for the vector. When linked with the encoder input, this variable allows the user to navigate the vector, and by extension, the menu. The *displayOffset* variable is needed for displays that have fewer lines than the length of any given *MenuNode* vector. In these cases, the lines that are displayed need to change depending on the *vectorCursor* position in the vector. Both variables are set to zero at the instantiation of a *MenuNode* object.

The *Menu* class contains an unordered map of the *MenuNodes* used and a member function *initMenu()*, which initialises the menu. Menu initialisation involves instantiation of the *MenuNodes* and their vectors, passing function pointers to their *useNode()* functions, and loading them into the unordered map. If the menu were to be extended at a later date, additional menu nodes should be added here.

The display and navigation of *MenuNode* vectors is facilitated with the two *Menu* member functions *displayMenu()* and *updateDisplay()*. These functions use the *MemberNode* member variables *vectorCursor* and *diplayOffset* to display the appropriate nodes of the *MenuNode* vector and cursor position on the LCD.

Figure 19 is a state diagram of the menu and indicates how a user can move between options in the menu. The test rig can be operated in manual or automatic modes. Manual mode simply allows the user to set a certain pressure in the PMA. Automatic mode is specifically for taking measurements to determine the static force characteristic of a PMA. The user can select how many and at which lengths measurements should be taken in the menu node *Set Lengths*. Subsequently, when *Start program* is selected, the user can decide at how many intervening pressure points the force should also be measured, and how many times the measurement cycles are to be repeated.

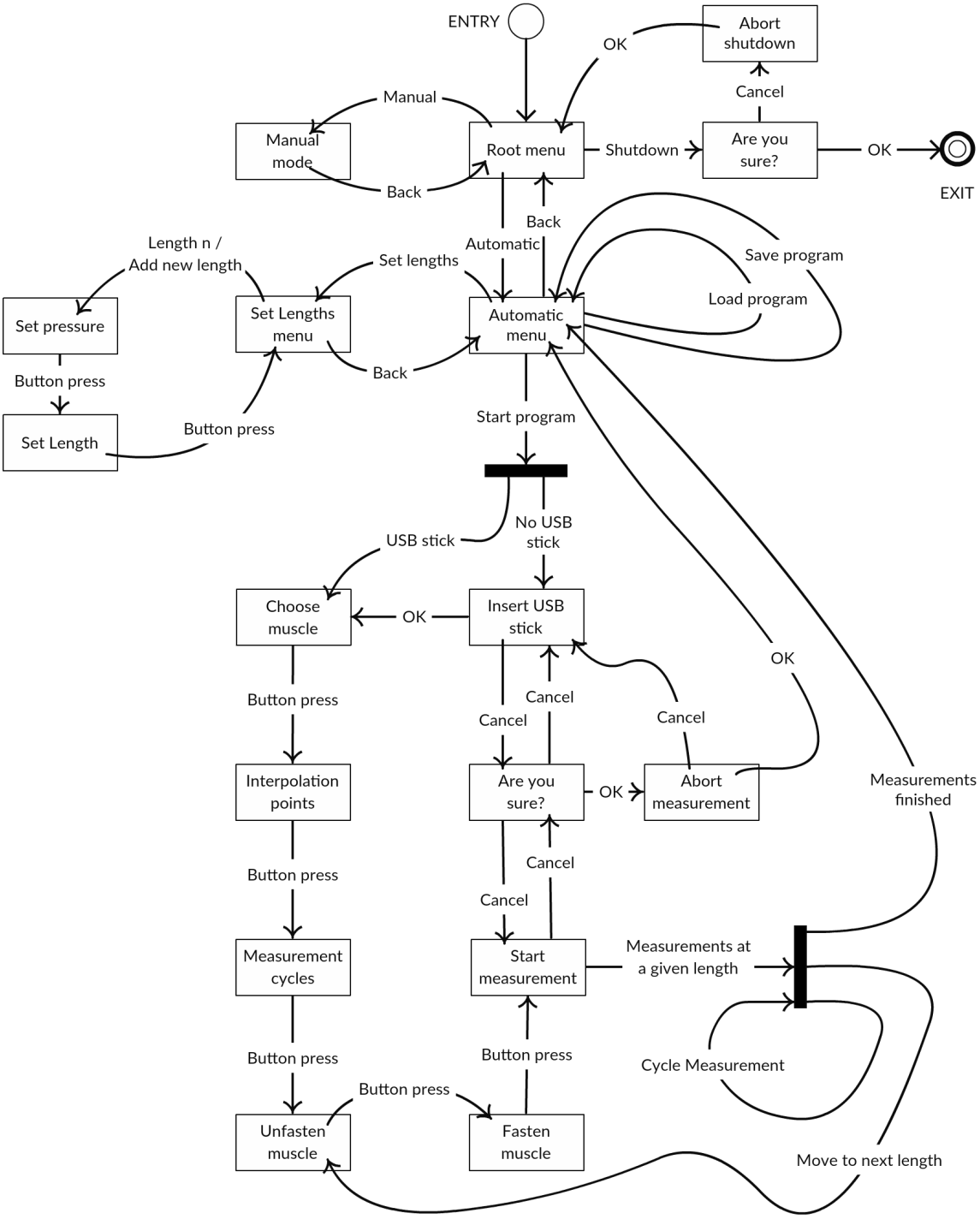


Figure 19: State diagram of the menu. On boot, the user finds themself at the root menu

3.4.1 Menu Structure

The top-level functions that are passed as pointers to the *MenuNode useNode()* member functions are declared and defined in MenuStructure.h and MenuStructure.cpp (Appendix E). The functions have been given the prefix *use* followed by the name of the node (e.g. *useRoot*). The MenuStructure files also include a number of other functions that support the *useNode()* functions.

The menu nodes *root*, *Automatic* and *Set lengths* are internal nodes that primarily display the contents of their vectors, and allow the user to continue on to further options. The first *MenuNode* in a given vector is displayed to the user on the LCD as "Back", as by convention this is the parent (see above), unless it's in the *root* vector.

If the button is pressed while navigating the menu, the *useNode()* function of the *MenuNode* of the current cursor position is called. This points to one of the following functions, whose actions are as described below.

useRoot() is called towards the end of *main()*, once set-up and initialisation of the device has taken place. This is the starting point of the user. The *MenuNode* vector of *root* is displayed until the button is pressed and the user is passed on to the next menu node. Unlike the other *useNode()* functions, *useRoot()* contains an infinite while loop, so the user cannot drop out of the program from here.

useManual() calls a secondary function *setPressure()*, which links the encoder position to the pressure set point of the PI controller running in a background thread. The function *setPressure()* enters a while loop in which the set point is continually updated. To give the user a visual indication of when the set point is reached, the display is updated with the pressure set point and measured value. When the button is pressed, the while loop is exited and the function returns from *setPressure()* and *useManual()*. The program continues in the *useRoot()* infinite loop and displays the contents of the *root* vector.

useAutomatic() is an internal node, that displays the contents of the *MenuNode* vector of *automatic*. It is from here that the user can set-up a measurement program.

3.4.2 Options and Measurement

From the *Automatic* menu (Figure 19) the user can set the lengths of the muscle that measurements should be taken, start the program of measurements or save/load a program stored on the USB stick.

Save and load program either populate the *vLengths* vector (see 3.3.3) with data in the *program.txt* file on the USB stick, or vice versa. In either case the data on the device being written to will be erased. The functions *useSaveProgram* and *useLoadProgram* in *MenuStructure.cpp* handle the front end operation, while back end functionality is passed on to the relevant functions in the *DeviceControl* class.

If no pre-existing program is to be used, the user would first select *Set lengths* which calls the *useSetLengths()* function. This is an internal node, that displays the contents of the *MenuNode* vector of *setLengths*, which itself is a reflection of the *vLengths* vector. If there are no lengths store in *vLengths*, then the user is confronted with the options *Back* and *Add new length*. From here the user can add or modify additional lengths at which to measure the exerted force. As lengths are added the menu is extended.

The result of selecting *Add new length* or to modify an existing length is similar. In both cases the user is moved through two screens: to first dial in a pressure value with the encoder that corresponds to the desired length, and then similarly input the length in the next. The difference is that *Add new length* creates a new *Length* object and adds it to *vLengths* and creates a corresponding *MenuNode* in the *setLengths* vector, while modifying a length simply adjusts the values of the existing length.

This system of setting a pressure value before the length, is necessary with the current test rig frame, as this only allows for the muscle to be fixed at specific predefined lengths. During the selection of the pressure value, the encoder is connected to *pressureSetPoint* of the *DeviceControl* object, and by extension the PID controller that sets the pressure in the muscle. This means that the user can adjust the length of the muscle to fit to the fixing points of the frame. The measurement of length remains a manual affair and the value is selected via the encoder.

At this stage the user will have selected which lengths (with their pressure values) are to form the backbone of the measurements. When the user selects the *Start program* menu node in *Automatic* mode the program enters *startProgram* and some final options can be selected, before the measurement routine commences.

Initially, a couple of housekeeping functions check some of the resources required. First, the CAN bus and valve are initialised again with *initCANbus()*. This line is redundant if the valve remains permanently plugged into the device. However, during testing it became apparent that if the valve was only plugged in after start-up, the control system would fail.

Secondly, the USB stick used to store the data is mounted with the helper function *promp-*

tUSB(). A number of helper functions with the prefix *prompt* are used throughout the software. They in turn use the function *prompt()* to get simple OK/Cancel input from the user. In this case, if the USB stick is not plugged in, the program will fail to mount it and call on the user to plug a USB stick in. Here, the user will be presented with the options OK and Cancel. If *OK* is selected the program will try to mount the USB stick again. On *Cancel*, the user will be asked if they wish to abort.

The function *useStartProgram* continues with three more functions, that each prompt for further information from the user and set the parameters of the measurements to be taken. These are: *promptChooseMuscle()*, *promptInterpolatePressure()* and *promtMeasurementCycles()*.

Firstly, with *promptChooseMuscle()* the user is called upon to input the diameter size of the muscle in the test rig ($5\mathrm{mm}$, $10\mathrm{mm}$, $20\mathrm{mm}$, $40\mathrm{mm}$). The selection adjusts which ADC input channel and voltage divider is used for the force measurement by sending the appropriate signals to the analogue switch (see Section 2.7).

Secondly, the function *promptInterpolatePressure()* adds to the number of pressure points at which a force measurement is made. The user selects how many (up to a maximum of three) additional, equidistant pressure points should be added between each length point pair. The original and additional pressure points are all added to the *vMeasurements* vector of the *DeviceControl* object.

Finally, *promptMeasurementCycles()* asks the user to select how many times they would like each force measurement to be taken, and stores this value in the variable *measurementCyclesPerLength*. In this way, large datasets can be generated and averages calculated.

At this point all the options have been set and the measurements can begin. Broadly, this part of the software is structured as two loops, one nested in the other. The outer for-loop is responsible for cycling through the length measurements stored in the *vLengths* vector. The loop warns the user to unfasten the bolts holding the muscle, before it pressurises it appropriately for the next length. Once at the right pressure, the user fastens the muscle again and the measurements at this length can take place.

The inner while-loop cycles through the pressure points in the *vMeasurements* vector, taking a force measurement at each. The measured force is also written to the *vMeasurements* vector. Once all the pressure points at this length have been measured, the force data stored in the *vMeasurements* vector is transferred to the next line of the data file on the USB stick. The vector is then cleared and filled with zeros, ensuring that pressure points that can't be

measured at shorter lengths always record a force of zero. The program then returns to the beginning of the outer for-loop and the user manually unfastens the bolts of the rig in preparation for the move to the next length.

An additional internal while-loop handles repeating measurements that were selected earlier with the *measurementCyclesPerLength* variable. Further details on the PID-controller can be found in Section 3.3.1.

The actual force measurement should only be made once the internal pressure of the muscle has reached the pressure set point. It is therefore important to ensure this happens irrespective of the controller gain settings and external factors, so even if the measured pressure swings around the set point multiple times or there is a drop in air pressure supply. A timer to wait for this to happen is not appropriate, as it's not known in advance how long this could take. In particular, it was noted during testing, that the supply of compressed air occasionally sank below the set point. This could happen for an indeterminate length of time, upto several minutes. An alternative could be to take a measurement when the measured pressure entered certain bounds of the set point. This, equally, had problems, in particular when a large overshoot caused the muscle pressure to enter the bounds and exit it again shortly afterwards.

The solution was a combination of the two approaches, so that a timer would begin to count down once the measured pressure was within $0.1\mathrm{bar}$of the set point. If the measured pressure left these bounds the timer would stop until the bounds were re-entered. Once the timer reached zero, a force measurement was taken. The advantage of this was particularly noticeable when the available pressure sank below the set point. In these instances the device would simply wait for the appropriate pressure to return.

In this way the *data.txt* file on the USB stick is filled with the force measurements that can subsequently be used to determine the static force characteristic of the muscle.

3.4.3 I²C and LCD

Header and source files (LCD.h and LCD.cpp) were developed to take care of the I²C interface to the display and the commands for the LCD itself. The contents of the files is heavily based on a python library for Raspberry Pis with similar functionality [26]. Additional functions that were added, improve the programmer's options in terms of which types can be placed on the display (eg int and float), and the position at which they can be displayed (left or right-hand side).

Following the boot process and the launch of the underlying device software, two things happen related to the display. First, access to the I^2C bus is opened and communication with the slave LCD device is initiated with its I^2C address. This occurs with the initI2C() function. Second, the initLCD() function initialises the LCD, a display that uses the Hitachi HD44780 protocol, in 4-bit interface mode [27]. At this stage the LCD is initialised and ready to display the menu or messages to the user.

3.4.4 Rotary Encoder and Button

The C++ eQEP API written by Nathaniel Lewis [28] was used as the interface to the eQEP driver. The main function creates an instance of the eQEP class based on the root path of the eQEP2 sysfs entry, and the operating mode of the hardware. In this case the eQEP hardware is run in absolute mode. This means its position starts at zero and is incremented or decremented by the encoder's movement, as opposed to being reset after a time interval.

Following instantiation, the period at which the eQEP hardware polls the input is set with the *set_period()* function. Smooth operation of the encoder with the menu was achieved with a period of $10\mathrm{ms}$(or 10,000,000ns).

4 Test Measurements

Two Festo PMAs were tested with the rig. While individual steps have been described previously, the process of testing a muscle is described in its entirety here:

- The device is connected to the force sensor, the pressure sensor, the valve and power. The connection to power should boot the device automatically.
- The PMA to be tested is installed in the rig: one end attached to the force sensor of the rig, the other to the sliding plate. At this stage, the plate can be left to slide and shouldn't be fixed in place.
- In the *Set lengths* menu the lengths to be measured can be added. The rig currently only has fixed, predefined lengths to which the muscle can be bolted. The user should, therefore, increase the pressure until the first set of threaded holes are aligned with the holes on the sliding plate. Pressing the encoder button will store this pressure and allow the user to input the length of the muscle at these holes. The length measurement is currently done manually. This is repeated for the number of lengths desired.

- On selecting *Start program*, the user is asked to insert a USB stick, if this is not already in place. The user can then set the muscle diameter, the number of pressure points to add between each length, and the number of measurements per length to take.
- The device will then confirm that the muscle is free at one end (via the sliding plate), before adjusting the pressure to the first length. Here the user must manually fasten the sliding plate to the rest of the rig. Once the muscle is fastened and the user has confirmed they wish the measurement to proceed, the device will run through all the stored pressure points and record the exerted force at each. If multiple measurements were selected the device will repeat the measurements as necessary.
- At the end of the measurements, the device will ask the user to unfasten the muscle. Again, this is done manually via the bolts holding the sliding plate. The device will then adjust the pressure to bring the muscle to the next length, before getting the user to confirm that they wish to proceed.
- After the final measurement the device will prompt the user to unfasten the muscle one last time before returning to it to the ambient pressure.
- The collected data is available in the *data.txt* file on the USB stick. The first row contains all the pressure points that were used in the measurements, while the first column identifies the length. Thus, the force measurements are assigned to a specific length and pressure.

The two PMAs tested were Festo DMSP-20-300N muscles, i.e. with a diameter of $20\mathrm{mm}$. Six lengths were stored and a further three interpolated pressure points were added between each two lengths. Each measurement was repeated ten times.

The data collected (see Appendix F) was evaluated in MATLAB using a script written by Mirco Martens (see Appendix G). This script averages force measurements of the same length and pressure, and plots these as a three dimensional map of the forces exerted at given muscle lengths and pressures.

4.1 Results

The static force characteristic of the two PMAs tested, as measured with the developed test rig, are given in Figures 20 and 21.

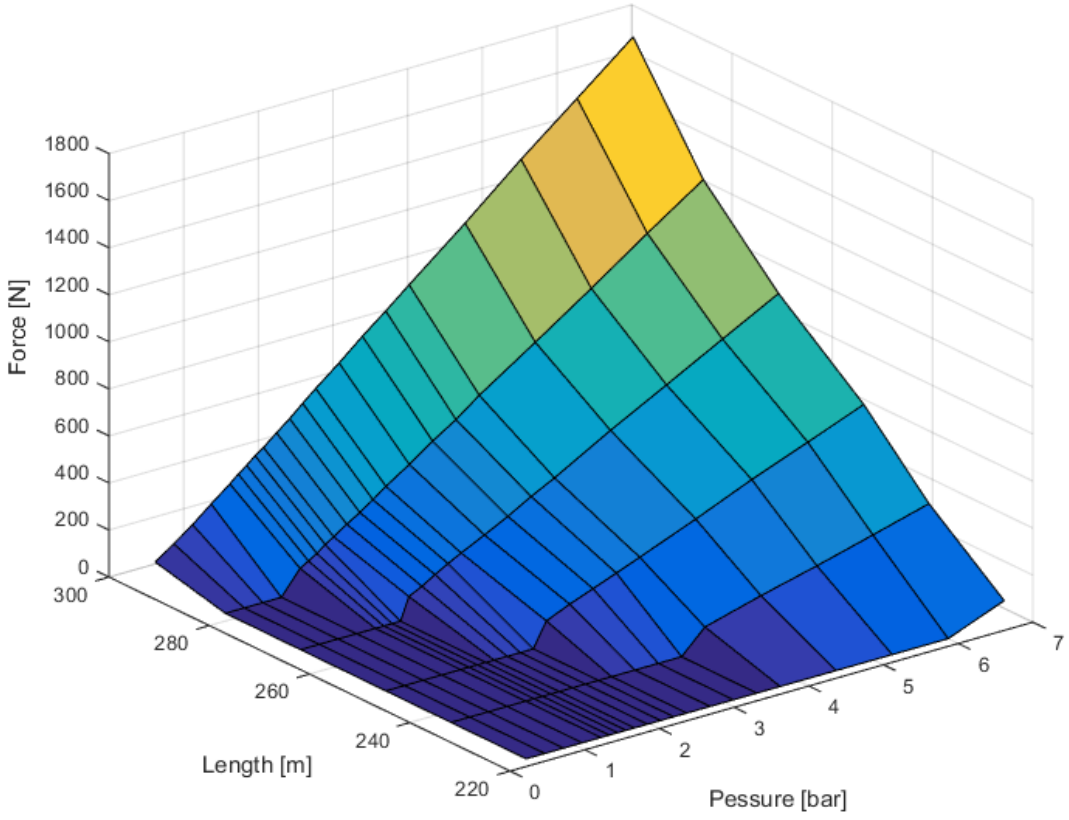


Figure 20: Static force characteristic of PMA 1

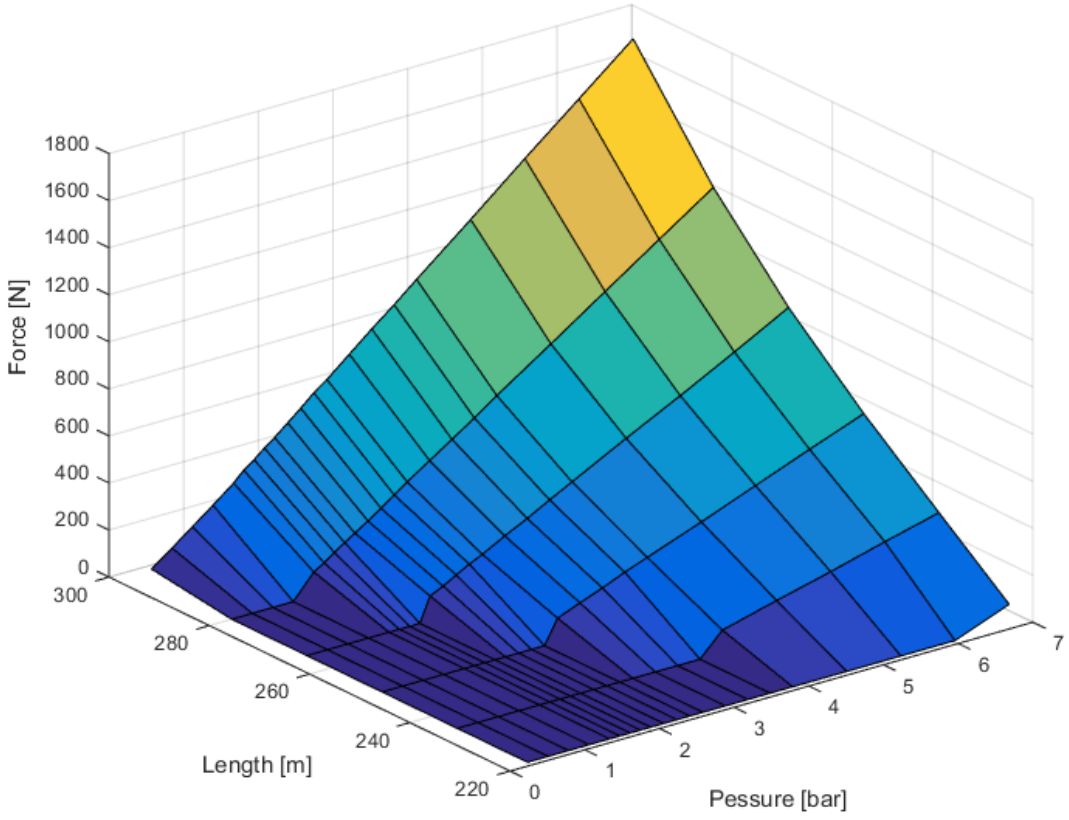


Figure 21: Static force characteristic of PMA 2

5 Discussion

From Figures 20 and 21 are good examples of the static force characteristics of Festo pneumatic muscles. At any given length the force exerted increases linearly with increasing internal pressure. As the muscle is made shorter, the increase in force with pressure remains linear but the gradient decreases.

The visible kink from no measured force to a measured force is an artefact of the measurement method. This is primarily due to the need for tightening the muscle against the direction that the force will be applied. Further issues are related to the nature of the test rig; it is currently only possible to fix the muscle to predefined lengths. The first length, as all lengths, is achieved by applying a pressure to the muscle. The hysteresis and non-linearities at the muscles pressure extremes will result in inaccuracies in the force measurement.

Secondly, the data points recorded as zero Newton were, in fact, set to zero, while the adjacent measured data points are subject to the errors inherent in the system.

Improvements to the test rig should include a way to set the PMA to any desired length. This would require a robust frame with a continuous length adjustment that can hold $6\mathrm{kN}$ (or the maximum force that a PMA would exert). One should be able to set this electronically. A pulse width modulation (PWM) output was foreseen for this, and designed into the device.

6 Conclusion

The aim of this thesis was to develop a test rig for the automated measurement of the static force characteristic of Pneumtic Muscle Actuators. This was achieved with the help of a BeagleBone Black SBC, a four line LCD display, a compressed air valve, pressure and force sensors and the development of a bespoke printed circuit board (PCB). The PCB contained a power regulation sub-circuit, sensor signal conditioning hardware, and interfaces between the BeagleBone Black and the CAN bus, the LCD display, a rotary encoder, and push buttons.

A discrete PID controller was programmed in the device to regulate the pressure in a Festo pneumatic muscle. A user interface and application for the automated measurement of exerted force at different pressures was built around the controller. The software was rounded off with means to store the data on a USB stick and have this evaluated using MATLAB on an external device.

Two Festo pneumatic muscles were tested with the device and their static force characteristics plotted. This demonstrated its suitability for rapidly generating large quantities of high resolution data for use in further applications, such as "sensorless" joints based around an antagonistic muscle pair.

References

[1] H.F. Schulte, "The characteristics of the McKibben artificial muscles". In *The Application of External Power in Prosthetics and Orthotics*, vol. 874. Washington, DC: National Academy of Sciences, 1961, pp.94-115.

[2] E. Kelasidi, et al. "A survey on pneumatic muscle actuators modeling". In *Journal of Energy and Power Engineering*, vol 6. 2011, pp. 1442-1452.

[3] S. Davis, D. G. Caldwell. "Braid Effects on Contractile Range and Friction Modeling in Pneumatic Muscle Actuators." In *The International Journal of Robotics Research* vol. 25, no. 4, 2006, pp. 359-369

[4] M. Doumit, et al. "Analytical modeling and experimental validation of the braided pneumatic muscle." In *IEEE transactions on robotics* vol. 25 no. 6, 2009, pp. 1282-1291.

[5] I. Boblan. "Modellbildung und Regelung eines fluidischen Muskelpaares." Diss. PhD., Fakultät III Prozesswissenschaften der Technischen Universität Berlin, 30, 2009.

[6] N. Vitiello, et al. "A sensorless torque control for Antagonistic Driven Compliant Joints." *Mechatronics* vol. 20, no .3, 2010, pp. 355-367.

[7] G. Coley, *BeagleBone Black System Reference Manual, Revision C.1*, May 2014

[8] Texas Instruments, *LM2596 Simple Switcher ® Power Converter 150kHz 3A Step-Down Voltage Regulator*, 2013

[9] Vishay General Semiconductor, Appl. Note 88436, *What is a Silicon Transient Voltage Suppressor and how does it work?*, 2007

[10] Analog Devices, Appl. Note 1368, *Ferrite Bead Demystified*, 2015

[11] Fairchild Semiconductor, *SMBJ5V0(C)A - SMBJ170(C)A 600 Watt Transient Voltage Suppresssors*, SMBJ24A datasheet, 2003

[12] NXP Semiconductors, Appl. Note 10441, *Level shifting techniques in I^2C-bus design*, 2007

[13] H. Zumbahlen, *Linear Circuit Design Handbook*. Amsterdam, Netherlands: Elsevier Ltd, 2008

[14] Texas Instruments Technical Staff, *AM335x ARM® Cortex™-A8 Microprocessors (MPUs) Technical Reference Manual*. Texas Instruments, Oct. 2011 [Revised Apr. 2013]

[15] Texas Instruments, Appl. Note SLLA270, *Controller Area Network Physical Layer Requirements*, Jan. 2008

[16] Texas Instruments, *SN65HVD23x 3.3-V CAN Bus Transceivers*, Mar. 2001 [Revised Jul. 2015]

[17] *Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, ISO 11898-2:2003

[18] Texas Instruments, *AM335x Sitara™Processors*, AM335x datasheet, Oct. 2011 [Revised Apr. 2016]

[19] D. Molloy, *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux*, 2015

[20] devicetree.org, "The Device Tree Specification," 2016 [Online]. Available: http://www.devicetree.org [Accessed: Nov. 07, 2016].

[21] eLinux.org, "Capemgr," Sep. 20, 2015 [Online]. Available: http://www.elinux.org/Capemgr [Accessed: Nov. 07, 2016]

[22] *Information technology – Portable Operating System Interface (POSIX®)*, ISO/IEC/IEEE Standard 9945, 2009

[23] Oliver Hartkopp, "can-utils: SocketCAN user space applications," Oct. 10, 2016 [Online]. Available: https://github.com/linux-can/can-utils [Accessed: Nov. 14, 2016]

[24] Festo, "Proportional directional control valves" Document number 00962321_DBL_001, Mar. 03 2016

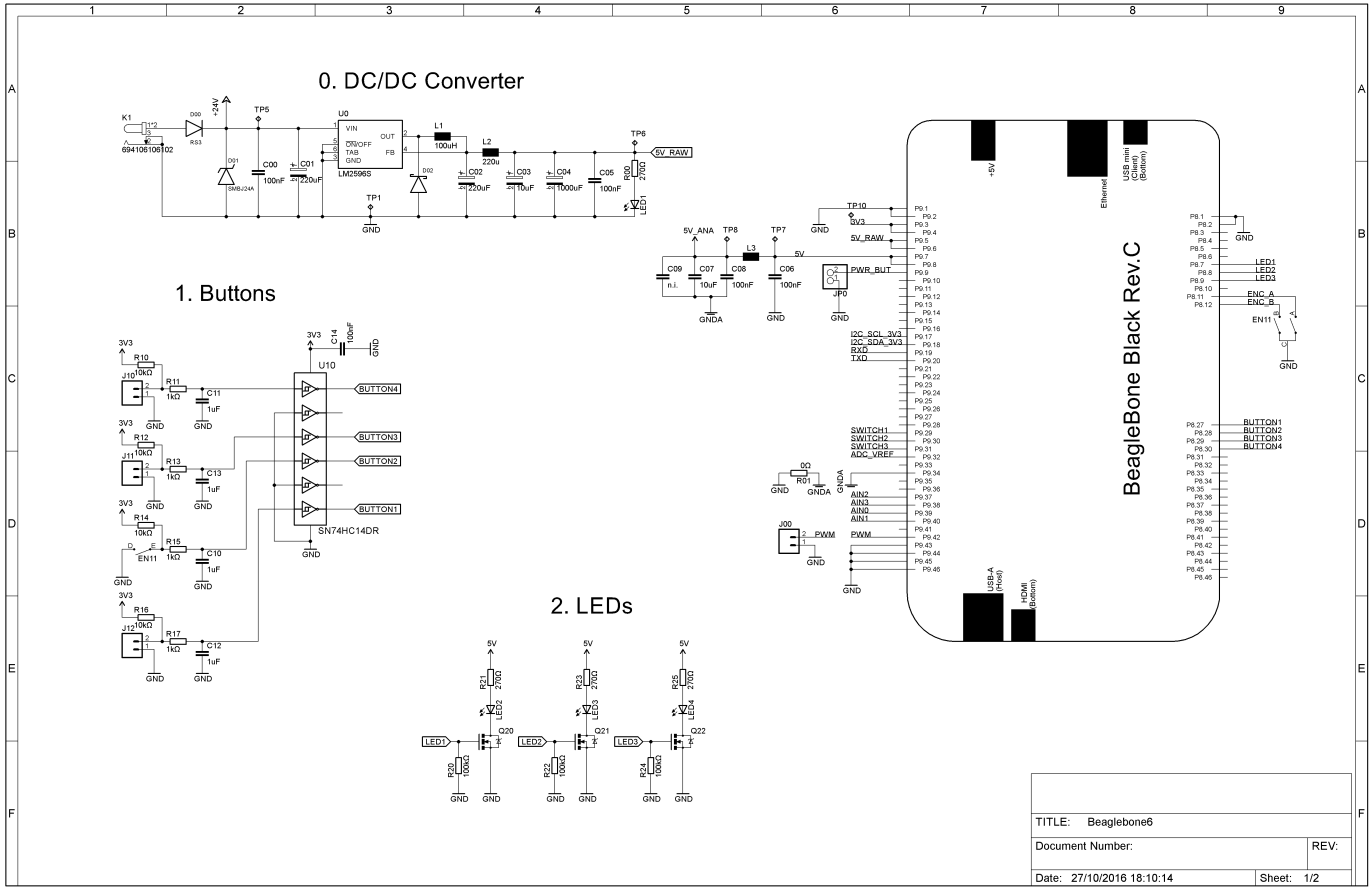
[25] aqax, "SockenCAN example with read timeout," Jun. 07, 2013 [Online]. Available: https://lnguin.wordpress.com/ [Accessed: Nov. 14, 2016]

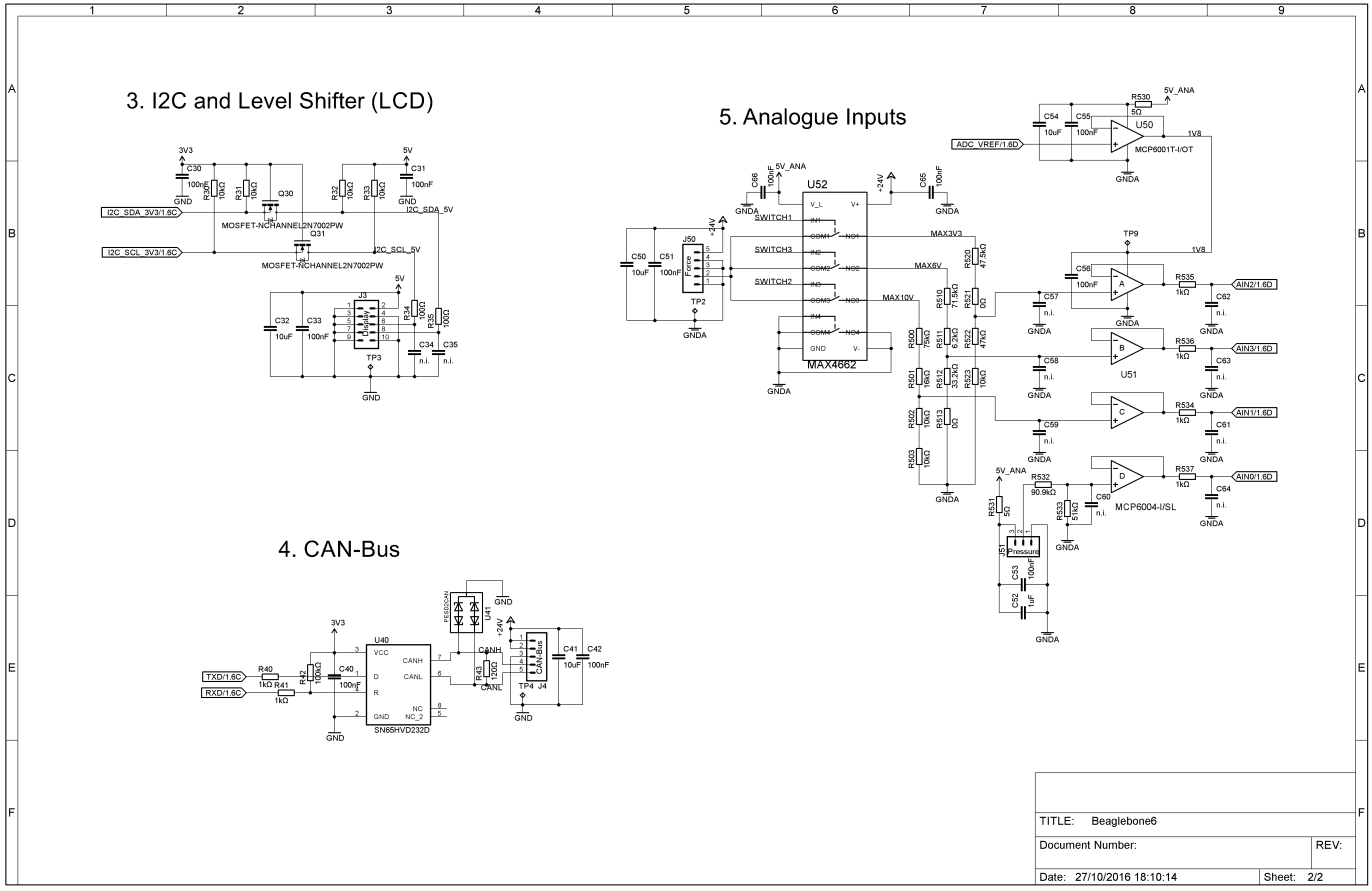
[26] Captain Stouf, "Raspberry Pi: using a 4x20 characters display," Mar. 20, 2014 [Online]. Available: http://hardware-libre.fr/2014/03/en-raspberry-pi-using-a-4x20-characters-display [Accessed: Mar. 24, 2016]

[27] Hitachi, *HD44780U (LCD-II), Dot Matrix Liquid Crystal Display Controller/Driver*, HD44780U datasheet, Sep. 1999

[28] Nathaniel Lewis, "C++ eQEP API," May. 15, 2014 [Online]. Available: https://github.com/Teknoman117/beaglebot/tree/master/encoders/api/c%2B%2B [Accessed: Apr. 21, 2016]

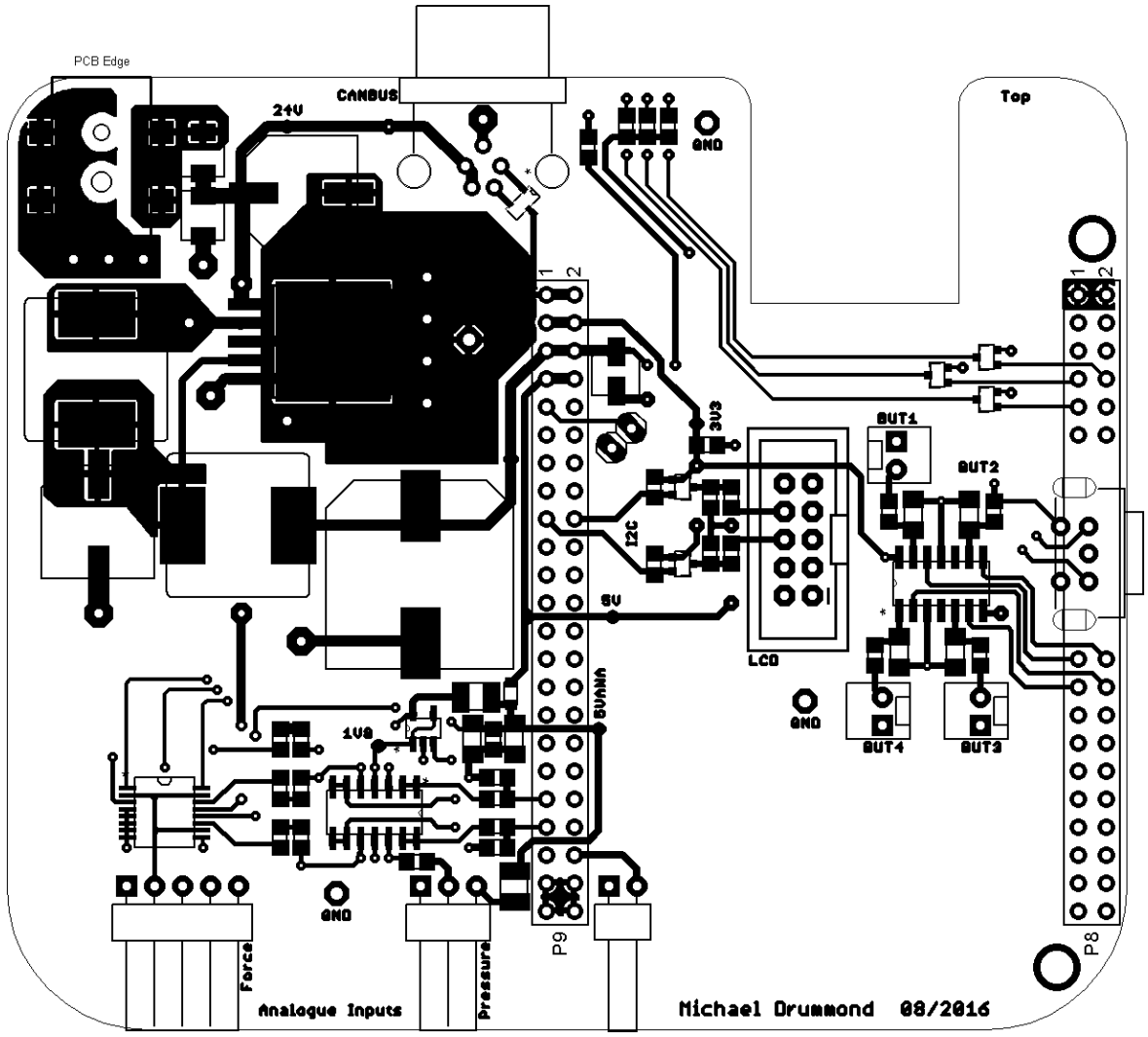
A Circuit Schematic



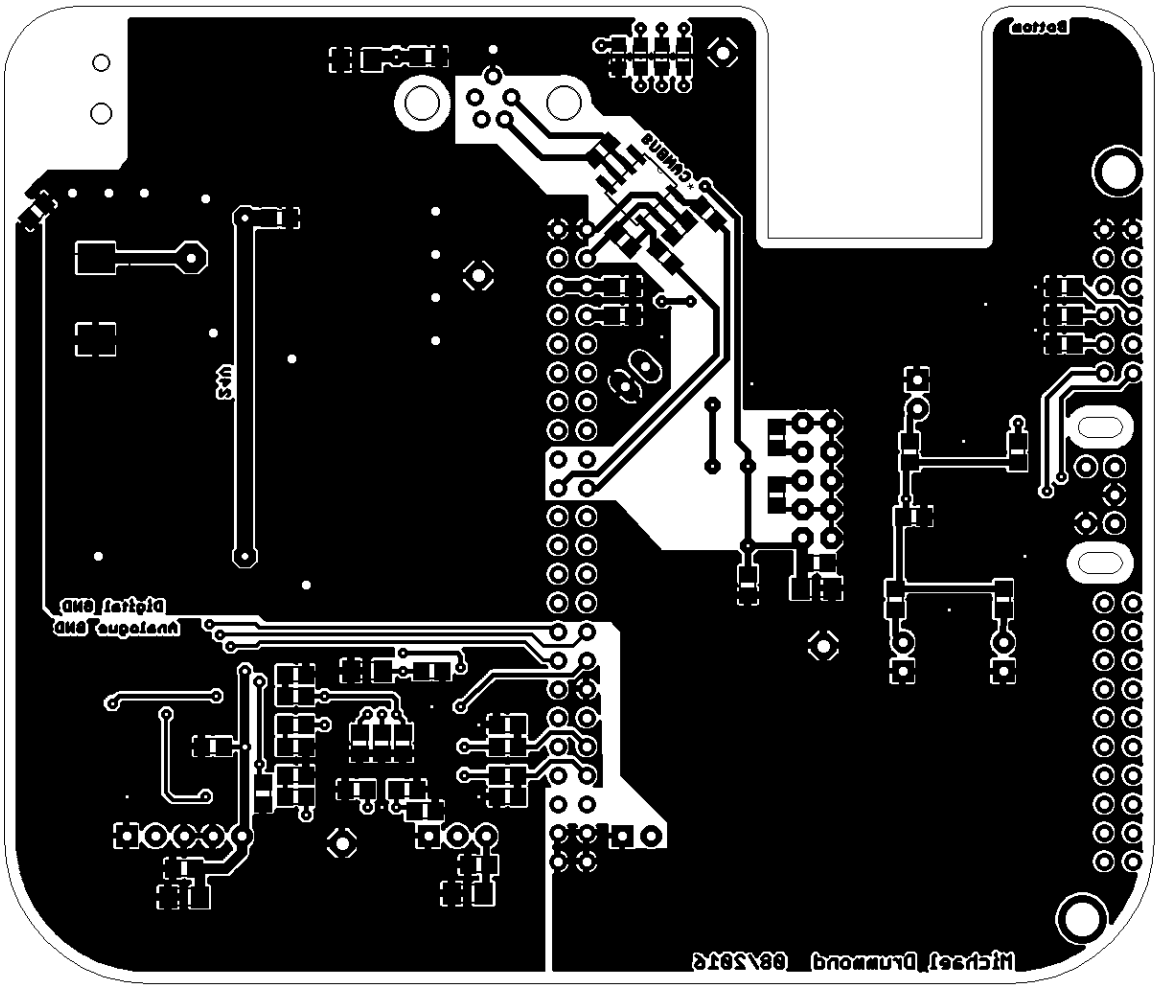

3. I2C and Level Shifter (LCD)
5. Analogue Inputs
4. CAN-Bus
MOSFET-NCHANNEL2N7002PW
MAX4662
MCP6001T-I/OT
MCP6004-I/SL
SN65HVD232D
TITLE: Beaglebone6
Document Number:
REV:
Date: 27/10/2016 18:10:14
Sheet: 2/2

B Board Layout

B.1 Top



B.2 Bottom



C Bill of Materials

Sub-circuit	Part	Value	Device	Package	Description	Manufacturer	Manufacturer Product #
0. Power supply	C00	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
0. Power supply	C01	220uF	WCAP-ASLI_10X10.5(DXL)	4.4X1.9X4.3(BXCXA)	WCAP-ASLI Aluminum Electrolytic Capacitors	Würth Elektronik	865080657018
0. Power supply	C02	220uF	WCAP-ASLI_10X10.5(DXL)	4.4X1.9X4.3(BXCXA)	WCAP-ASLI Aluminum Electrolytic Capacitors	Würth Elektronik	865080657018
0. Power supply	C03	10uF	WCAP-ASLI_4X5.5(DXL)	2.6X1.6X1(BXCXA)	WCAP-ASLI Aluminum Electrolytic Capacitors	Würth Elektronik	865080440002
0. Power supply	C04	1000uF	WCAP-ASLI_16X17(DXL)	6.5X3.5X6(BXCXA)	WCAP-ASLI Aluminum Electrolytic Capacitors	Würth Elektronik	865060663015
0. Power supply	C05	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
0. Power supply	C06	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
0. Power supply	C07	10uF	C-EUC1206	C1206	CAPACITOR (X5R, 16V general purpose ceramic cap)	Kemet	C1206C106K4PAC
0. Power supply	C08	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
0. Power supply	C09	n.i.	C-EUC1206	C1206	CAPACITOR		
0. Power supply	D00		RS3G-E3/57T	DO-214AA	Surface Mount Fast Switching Rectifier	Vishay	RS3G-E3/57T
0. Power supply	D01		SMBJ24A	DO-214AA	600 Watt Transient Voltage Suppressor	Fairchild	SMBJ24A
0. Power supply	D02		SS34	DO-214AB	Schottky Rectifier	Fairchild	SS34
0. Power supply	J00		PSS 254/2W	MOLEX-1X2_ANGLED	CONNECTOR (Socket, angled, 2-pole)	Würth Elektronik	61900211021
0. Power supply	JP0		PIN HEADER - 1X2	1X02	Pin header		
0. Power supply	K1		69410106102	694108000000	DC/DC Power Jack, right angled, SMT, center pin ⌀2.0, with 4 Soldering Pads & Peg	Würth Elektronik	694108000000
0. Power supply	L1	100uH	SRR1208-101YL	SRR1208	Shielded High Power Inductor	Bourns	
0. Power supply	L2	220uH	SRR1208-221KL	SRR1208	Shielded High Power Inductor	Bourns	SRR1208-221KL
0. Power supply	L3		WE-CBF_0805(WB/HS)	WE-CBF_0805(W=3)	EMI Suppression Ferrite Bead	Würth Elektronik	742792043
0. Power supply	R00	270Ω	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F2700T5E
0. Power supply	R01	0Ω	ERJ-6	R0805	Anti-Surge Thick Film Chip Resistors	Panasonic	ERJ-6GEY0R00V
0. Power supply	U00		LM2596S	TO263-5	SIMPLE SWITCHER®Power Converter 150 kHz 3A Step-Down Voltage Regulator	Texas Instruments	
1. Buttons	C10	1uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Yageo	CC1206JKX7R7BB105
1. Buttons	C11	1uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Yageo	CC1206JKX7R7BB105
1. Buttons	C12	1uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Yageo	CC1206JKX7R7BB105
1. Buttons	C13	1uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Yageo	CC1206JKX7R7BB105
1. Buttons	C14	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
1. Buttons	J10		PSS 254/2G	MOLEX-1X2	CONNECTOR (Socket, straight, 2-pole)	Würth Elektronik	61900211121
1. Buttons	J11		PSS 254/2G	MOLEX-1X2	CONNECTOR (Socket, straight, 2-pole)	Würth Elektronik	61900211121
1. Buttons	J12		PSS 254/2G	MOLEX-1X2	CONNECTOR (Socket, straight, 2-pole)	Würth Elektronik	61900211121
1. Buttons	R10	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
1. Buttons	R11	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
1. Buttons	R12	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
1. Buttons	R13	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
1. Buttons	R14	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
1. Buttons	R15	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
1. Buttons	R16	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
1. Buttons	R17	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
1. Buttons	U10		SN74HC14DR	SOIC-14	HEX SCHMITT-TRIGGER INVERTERS	Texas Instruments	SN74HC14DR
1. Buttons	U11		1EN11-VSM1BQ15	EN11	Rotary Encoder, 1EN11-VSM1BQ15	TT Electronics	9302520100
2. LEDs	LED1		CHIP-LED	805	SMD-LED 0805 Blue 100 mcd 120° 20mA 3.3V	Kingbright	KP-2012QBC-D
2. LEDs	LED2		CHIP-LED	805	SMD-LED 0805 Blue 100 mcd 120° 20mA 3.3V	Kingbright	KP-2012QBC-D

2. LEDs	LED3		CHIP-LED	805	SMD-LED 0805 Blue 100 mcd 120° 20mA 3.3V	Kingbright	KP-2012QBC-D
2. LEDs	LED4		CHIP-LED	805	SMD-LED 0805 Blue 100 mcd 120° 20mA 3.3V	Kingbright	KP-2012QBC-D
2. LEDs	Q20		2N7002PW	SOT323	60 V, 310 mA N-channel Trench MOSFET	NXP	2N7002PW,115
2. LEDs	Q21		2N7002PW	SOT323	61 V, 310 mA N-channel Trench MOSFET	NXP	2N7002PW,115
2. LEDs	Q22		2N7002PW	SOT323	62 V, 310 mA N-channel Trench MOSFET	NXP	2N7002PW,115
2. LEDs	R20	100kΩ	ERJ-P06	R0805	Anti-Surge Thick Film Chip Resistors		ERJ-P06J104V
2. LEDs	R21	270Ω	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F2700T5E
2. LEDs	R22	100kΩ	ERJ-P06	R0805	Anti-Surge Thick Film Chip Resistors	Panasonic	ERJ-P06J104V
2. LEDs	R23	270Ω	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F2700T5E
2. LEDs	R24	100kΩ	ERJ-P06	R0805	Anti-Surge Thick Film Chip Resistors	Panasonic	ERJ-P06J104V
2. LEDs	R25	270Ω	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F2700T5E
3. I2C, level shifter & LCD	C30	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
3. I2C, level shifter & LCD	C31	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
3. I2C, level shifter & LCD	C32	10uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Kemet	C1206C106K4PAC
3. I2C, level shifter & LCD	C33	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
3. I2C, level shifter & LCD	C34	n.i.	C-EUC0805	C0805	CAPACITOR		
3. I2C, level shifter & LCD	C35	n.i.	C-EUC0805	C0805	CAPACITOR		
3. I2C, level shifter & LCD	J3		M05X2SHD	2X5-SHROUDED	CONNECTOR (10 pole box header 2,54mm)	BKL Electronic	10120554
3. I2C, level shifter & LCD	Q30		2N7002PW	SOT323	63 V, 310 mA N-channel Trench MOSFET	NXP	2N7002PW,115
3. I2C, level shifter & LCD	Q31		2N7002PW	SOT323	64 V, 310 mA N-channel Trench MOSFET	NXP	2N7002PW,115
3. I2C, level shifter & LCD	R30	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
3. I2C, level shifter & LCD	R31	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
3. I2C, level shifter & LCD	R32	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
3. I2C, level shifter & LCD	R33	10kΩ	CR0805	R0805	Chip Resistors	Panasonic	CR0805-FX-1002ELF
3. I2C, level shifter & LCD	R34	100Ω	0805S8	R0805	Thick Film Chip Resistors	Royalohm	0805S8F1000T5E
3. I2C, level shifter & LCD	R35	100Ω	0805S8	R0805	Thick Film Chip Resistors	Royalohm	0805S8F1000T5E
4. CAN-Bus	C40	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
4. CAN-Bus	C41	10uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Kemet	C1206C106K4PAC
4. CAN-Bus	C42	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
4. CAN-Bus	J4		BINDER_M9_ANGLED	BINDER_M9_ANGLED	CONNECTOR		09-0416-55-05
4. CAN-Bus	R40	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
4. CAN-Bus	R41	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
4. CAN-Bus	R42	100kΩ	ERJ-P06	R0805	Anti-Surge Thick Film Chip Resistors	Panasonic	ERJ-P06J104V
4. CAN-Bus	R43	120Ω	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3J0121T5E
4. CAN-Bus	U40		SN65HVD232DR	SOIC-8	3.3-V CAN Bus Transceivers	Texas Instruments	SN65HVD232DR
4. CAN-Bus	U41		PESD2CAN	SOT23	CAN bus ESD protection diode	NXP	PESD2CAN,215
5. Analogue inputs	C50	10uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Kemet	C1206C106K4PAC
5. Analogue inputs	C51	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
5. Analogue inputs	C52	1uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Yageo	CC1206JKX7R7BB105
5. Analogue inputs	C53	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
5. Analogue inputs	C54	10uF	C-EUC1206	C1206	CAPACITOR (X7R, 16V general purpose ceramic cap)	Kemet	C1206C106K4PAC
5. Analogue inputs	C55	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
5. Analogue inputs	C56	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
5. Analogue inputs	C57	n.i.	C-EUC0805	C0805	CAPACITOR		
5. Analogue inputs	C58	n.i.	C-EUC0805	C0805	CAPACITOR		
5. Analogue inputs	C59	n.i.	C-EUC0805	C0805	CAPACITOR		
5. Analogue inputs	C60	n.i.	C-EUC0805	C0805	CAPACITOR		
5. Analogue inputs	C61	n.i.	C-EUC0805	C0805	CAPACITOR		
5. Analogue inputs	C62	n.i.	C-EUC0805	C0805	CAPACITOR		
5. Analogue inputs	C63	n.i.	C-EUC0805	C0805	CAPACITOR		
5. Analogue inputs	C64	n.i.	C-EUC0805	C0805	CAPACITOR		

5. Analogue inputs	C65	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
5. Analogue inputs	C66	100nF	C-EUC0805	C0805	CAPACITOR (X7R, 50V general purpose ceramic cap)	Yageo	CC0805KRX7R9BB104
5. Analogue inputs	J50		PSS 254/5W	MOLEX-1X5_ANGLED	CONNECTOR (Socket, angled, 5-pole)	Würth Elektronik	61900511021
5. Analogue inputs	J51		PSS 254/3W	MOLEX-1X3_ANGLED	CONNECTOR (Socket, angled, 3-pole)	Würth Elektronik	61900311021
5. Analogue inputs	R500	75kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6APB753V
5. Analogue inputs	R501	16kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6AEB163V
5. Analogue inputs	R502	10kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6YEB103V
5. Analogue inputs	R503	10kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6YEB103V
5. Analogue inputs	R510	71,5kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6AEB7152V
5. Analogue inputs	R511	6.2kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6AEB622V
5. Analogue inputs	R512	33.2kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6AEB3322V
5. Analogue inputs	R513	0Ω	R-EU_R0805	R0805	RESISTOR	Panasonic	ERJ-6GEY0R00V
5. Analogue inputs	R520	47.5kΩ	R-EU_R0805	R0805	RESISTOR	Microtech	CMF080547k50.110
5. Analogue inputs	R521	0Ω	R-EU_R0805	R0805	RESISTOR	Panasonic	ERJ-6GEY0R00V
5. Analogue inputs	R522	47kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6AEB473V
5. Analogue inputs	R523	10kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6YEB103V
5. Analogue inputs	R530	5Ω	ERJ-P14	R1210	Anti-Surge Thick Film Chip Resistors	Panasonic	ERJ-P14J4R7U
5. Analogue inputs	R531	5Ω	ERJ-P14	R1210	Anti-Surge Thick Film Chip Resistors	Panasonic	ERJ-P14J4R7U
5. Analogue inputs	R532	90.9kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERJ-1GEF9092C
5. Analogue inputs	R533	51kΩ	R-EU_R0805	R0805	RESISTOR	Panasonic	ERA-6APB513V
5. Analogue inputs	R534	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
5. Analogue inputs	R535	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
5. Analogue inputs	R536	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
5. Analogue inputs	R537	1kΩ	HP05W3	R0805	Thick Film Chip Resistors	Royalohm	HP05W3F1001T5E
5. Analogue inputs	U50		MCP6001T-I/OT	SOT23-5	Single low-power OPAMP	Microchip Tech.	MCP6001T-I/OT
5. Analogue inputs	U51		MCP6004-I/SL	SOIC-14	Quad low-power OPAMP	Microchip Tech.	MCP6004-I/SL
5. Analogue inputs	U52		MAX4662	SSOP16	2.5W, Quad, SPST, CMOS Analog Switches	Maxim Integrated	

D Example Device Tree Overlay

Listing 1: GPIO.dts

```
/dts-v1/;
/plugin/;

/{
    compatible = "ti,beaglebone", "ti,beaglebone-black";

    part-number = "GPIOs";
    version = "00A0";

    fragment@0 {
        target = <&am33xx_pinmux>;
        __overlay__ {
            gpios: pinmux_gpios{
                pinctrl-single,pins = <
        // LEDs:
        0x090 0x07    // P8_07 output, pulldown, mux-mode 7
        0x094 0x07    // P8_08 output, pulldown, mux-mode 7
        0x09c 0x07    // P8_09 output, pulldown, mux-mode 7
        // Buttons:
        0x0e0 0x27    // P8_27 input, pulldown, mmode 7 GPIO2_22
        0x0e8 0x27    // P8_28 input, pulldown, mmode 7 GPIO2_24
        0x0e4 0x27    // P8_29 input, pulldown, mmode 7 GPIO2_23
        0x0ec 0x27    // p8_30 input, pulldown, mmode 7 GPIO2_25
                >;
            };
        };
    };

    fragment@1 {
        target = <&ocp>;
        __overlay__ {
            gpios {
                compatible = "gpio-of-helper";
                pinctrl-names = "default";
                pinctrl-0 = <&gpios>;
            };
        };
    };
};
```

E Software API

E.1 Menu Class Reference

`#include <menu.h>`

Public Member Functions

- Menu ()

  Default constructor.
- Menu (eQEP *encoder, exploringBB::GPIO *button, LCD *lcd)

  Constructor that defines the encoder, button and LCD.
- ~Menu ()

  Class destructor.
- std::unordered_map< std::string, MenuNode * > initMenu ()
- int displayMenu (MenuNode *current)
- int updateDisplay (MenuNode *current)

Public Attributes

- eQEP * encoder

  Pointer to object to control and read the encoder.
- exploringBB::GPIO * button

  Pointer to object to read the button input.
- LCD * lcd

  Pointer to object to control the LCD display.
- std::unordered_map< std::string, MenuNode * > map

  Map of the MenuNodes.
- MenuNode * current

  Keeps track of the users position in the menu.

E.1.1 Detailed Description

The Menu class groups MenuNodes into an unordered map, keeps track of which MenuNode is currently being displayed, and acts as the interface to the encoder and button that make navigating the menu possible.

E.1.2 Constructor & Destructor Documentation

E.1.2.1 Menu()

```
Menu::Menu (
            eQEP * encoder,
            exploringBB::GPIO * button,
            LCD * lcd )
```

Parameters

encoder	Pointer to eQEP object
button	Pointer to GPIO object
lcd	Pointer to LCD object

E.1.3 Member Function Documentation

E.1.3.1 displayMenu()

```
int Menu::displayMenu (
            MenuNode * current )
```

Displays the current MenuNode v vector to the LCD

Parameters

current	The MenuNode whose v vector is to display

Returns

An integer, 0 is success

E.1.3.2 initMenu()

```
std::unordered_map< std::string, MenuNode * > Menu::initMenu ( )
```

Loads the MenuNodes into the Menu unordered map

Returns

unordered map with the MenuNodes that make up the menu

E.1.3.3 updateDisplay()

```
int Menu::updateDisplay (
            MenuNode * current )
```

Updates the display with the latest position of the cursor in the v vector. If the cursor has moved passed the bottom of the display, then the elements of the vector that are displayed are also shifted.

Parameters

current	The MenuNode that is being displayed on the LCD

Returns

An integer, 0 is success

The documentation for this class was generated from the following files:

- menu.h
- menu.cpp

E.2 MenuNode Class Reference

```
#include <menu.h>
```

Public Member Functions

- MenuNode (std::string name, int(∗pt2Func)(void))

  Constructor that sets its name and useNode() function.
- MenuNode (std::string name, MenuNode ∗parent, int(∗pt2Func)(void))

  Constructor that sets name, parent and useNode() function.
- ∼MenuNode ()

  Class destructor.
- void setChild (MenuNode ∗child)
- std::string getName ()
- void setName (std::string name)

Public Attributes

- std::string name

  Acts as an identifier.

- std::vector< MenuNode * > v

  Vector that holds the children of this node.

- int displayOffset

  Keeps track of the display offset.

- int vectorCursor

  Keeps track of the vector cursor position.

- int(* useNode)()

E.2.1 Detailed Description

MenuNodes are the building blocks of a menu. Conceptually, a menuNode represents a state in which the user of the menu can either move on to another node, or manipulate some other function on the device.

E.2.2 Constructor & Destructor Documentation

E.2.2.1 MenuNode() [1/2]

```
MenuNode::MenuNode (
            std::string name,
            int(*)(void) pt2Func )
```

Parameters

name	Name of the MenuNode
pt2Func	Function pointer for useNode()

E.2.2.2 MenuNode() [2/2]

```
MenuNode::MenuNode (
            std::string name,
            MenuNode * parent,
            int(*)(void) pt2Func )
```

Parameters

name	A string identifying the MenuNode
parent	A MenuNode pointer to the parent of this MenuNode
pt2Func	Function pointer for useNode()

E.2.3 Member Function Documentation

E.2.3.1 getName()

```
std::string MenuNode::getName ( )
```

Returns the name of this MenuNode
Returns

A string

E.2.3.2 setChild()

```
void MenuNode::setChild (
            MenuNode * child )
```

Adds a child MenuNode to this MenuNode

Parameters

child	Pointer to a MenuNode

E.2.3.3 setName()

```
void MenuNode::setName (
            std::string name )
```

Sets the name of this MenuNode

Parameters

name	A string

E.2.4 Member Data Documentation

E.2.4.1 useNode

`int(* MenuNode::useNode) ()`

Dereferences a function pointer

Returns

An integer

The documentation for this class was generated from the following files:

- menu.h
- menu.cpp

E.3 DeviceControl Class Reference

Controls the back-end functions of the test rig.

`#include <deviceControl.h>`

Public Member Functions

- DeviceControl (CANSocket *can)

  Constructor that defines CAN bus and GPIO pins.
- void * controlfunc (void)

  Function pointer to the control function.
- int startThread (DeviceControl *control)

  Starts a thread in the DeviceControl object.
- int terminateThread ()

  Terminates the thread.
- int initCANbus ()

  Initialises the CAN bus and the Festo valve.
- int terminateCANbus ()

  Terminates the CAN bus.
- int initValve ()

  Initialises the pressure valve.

- int initI2C ()

  Initialises an I2C bus.
- int initPRU ()

  Initialise the Programmable Real-Time Units.
- int terminatePRU ()

  Terminates the program running on the PRU.
- int readAnalog (int channel)

  Reads an ADC channel.
- int analogSwitch (int selector)

  Selects the force measurement channel.
- float measureForce (rig::muscleSize choiceOfMuscle)

  Reads the ADC and converts the value to Newtons.
- float binary2Newton (float force, int choiceOfMuscle)

  Converts an ADC measurement to force in Newtons.
- float binary2Bar (int adcValue)

  Converts an ADC measurement to pressure in milli-bar.
- int writePressurePointsToUSB ()

  Writes the pressure points used in a measurement to the USB stick.
- int writeMeasurementsToUSB (int index)

  Writes a row of measurements to the USB stick.
- int saveProgram ()

  Stores details of the program on the USB stick.
- int loadProgram ()

  Loads details of a program from the USB stick.

Static Public Member Functions

- static void * thread_helper (void *context)

  A helper function to get the thread started with controlfunc()

Public Attributes

- float pressureSetPoint

  Pressure set point in bar.

- double pressureMeasured

  Measured pressure in bar.

- std::vector< rig::Length * > vLengths

  Stores Length data.

- std::vector< rig::Measurement * > vMeasurements

  Stores measurement data.

E.3.1 Detailed Description

DeviceControl controls the back-end of the test rig. It is responsible for initialising the various elements of the control system (CAN bus, the pressure valve, the I2C bus, and the Programmable Real-time units); starting the thread that runs the PID controller in the background; reading pressure and force measurements; and storing the data.

E.3.2 Constructor & Destructor Documentation

E.3.2.1 DeviceControl()

```
DeviceControl::DeviceControl (
            CANSocket * can )
```

This constructor is passed a pointer to CAN object with which it will communicate with the valve. In addition, it defines the GPIO pins to be used with the analogue switch that chooses the voltage divider over which the force measurement will be taken.

Parameters

can	Pointer to CANSocket object for CAN communication

E.3.3 Member Function Documentation

E.3.3.1 analogSwitch()

```
int DeviceControl::analogSwitch (
```

```
            int selector )
```

Selects the voltage divider channel over which the force measurement signal reaches the ADC input pin. This happens with the appropriate switching of the DeviceControl GPIO pins which control an analogue switch.

Parameters

selector	The desired channel (0-3)

Returns

An integer: 0 is success

E.3.3.2 binary2Bar()

```
float DeviceControl::binary2Bar (
            int adcValue )
```

Converts a BeagleBone Black ADC measurement into milli-bars of pressure. Uses voltage divider settings based on the hardware, and a linear interpolation of the pressure sensor characteristic.

Parameters

adcValue	A value read directly from the BBB ADC (0-4095)

Returns

A float: pressure in milli-bar

E.3.3.3 binary2Newton()

```
float DeviceControl::binary2Newton (
            float force,
            int choiceOfMuscle )
```

Converts a BeagleBone Black ADC measurement into Newtons of force, depending on the specific muscle (and therefore voltage divider)settings are used.

Parameters

force	A value read directly from the BBB ADC (0-4095)
choiceOfMuscle	The muscle diameter setting, determines the voltage divider settings

Returns

A float: the force in Newtons

E.3.3.4 controlfunc()

```
void * DeviceControl::controlfunc (
            void  )
```

This function pointer is passed to a thread running parallel to the main thread. It runs in a loop that waits in every cycle for an interrupt from the PRU timer. Then takes the measured pressure and set point to calculate an output value with the help of a PID controller.

E.3.3.5 initCANbus()

```
int DeviceControl::initCANbus ( )
```

Enables a network interface to the CAN bus and binds the CAN socket of CAN object of DeviceControl. If successful the pressure valve will be initialised according to the data sheet with the Node ID 0x01.
Returns

An integer: 0 = success, -1 = network interface failed, 2 = network interface busy (most likely already enabled)

E.3.3.6 initI2C()

```
int DeviceControl::initI2C ( )
```

Initialises the I2C2 bus for communication with the LCD. This is done by creating and returning a file descriptor to /dev/i2c-2 and initialising communication by sending the address of the LCD.
Returns

The file descriptor of the I2C bus

E.3.3.7 initPRU()

```
int DeviceControl::initPRU ( )
```

Allocates and initialises memory for the Programmable Real-Time units; maps the PRU's interrupts; loads and executes the assembler program on the BeagleBone.

Returns

An integer: 0 is success

E.3.3.8 initValve()

```
int DeviceControl::initValve ( )
```

Initialises the pressure valve by putting a series of commands on the CAN bus that sets the valve's Node ID to 0x01 and prepares it to receive valve position values.

Returns

An integer: 0 is success

E.3.3.9 loadProgram()

```
int DeviceControl::loadProgram ( )
```

Opens a filesstream to a file called program.txt on the USB stick. If the file exist then the details contained in it are used to populate the vLengths vector.

Returns

An integer: 0 is success

E.3.3.10 measureForce()

```
float DeviceControl::measureForce (
            rig::muscleSize choiceOfMuscle )
```

Uses the choiceOfMuscle parameter to select the correct ADC input channel. Reads the input and converts it to a value in Newtons of force.

Parameters

choiceOfMuscle	

Returns

A float: the measured force in Newton

E.3.3.11 readAnalog()

```
int DeviceControl::readAnalog (
            int channel )
```

Opens a file stream to the appropriate ADC channel input and reads the calculated value.

Parameters

channel	The ADC channel to be read

Returns

An integer: the measured value at the ADC input (0-4095)

E.3.3.12 saveProgram()

```
int DeviceControl::saveProgram ( )
```

Opens a filesstream to a file called program.txt on the USB stick. The file is opened in "truncate" mode, such that any previously stored data is deleted when it is opened. The values from the Length structs in vLengths is written to the file. The filestream is closed.

Returns

An integer: 0 is success

E.3.3.13 startThread()

```
int DeviceControl::startThread (
            DeviceControl * control )
```

Starts a thread with a round robin scheduling policy and a priority as defined in THREAD_PRIORITY

Parameters

control	The deviceControl object to which the thread will belong

Returns

An integer, 0 is success

E.3.3.14 terminateCANbus()

```
int DeviceControl::terminateCANbus ( )
```

Terminates the CAN socket and closes the network interface to the CAN bus.

Returns

An integer: 0 is success

E.3.3.15 terminatePRU()

```
int DeviceControl::terminatePRU ( )
```

Exits the program running on the Programmable Real-Time Unit(s) and closes the memory mapping to the main processor

Returns

An integer: 0 is success

E.3.3.16 terminateThread()

```
int DeviceControl::terminateThread ( )
```

Terminates the control loop thread that DeviceControl is running

Returns

An integer, 0 is success

E.3.3.17 thread_helper()

```
void * DeviceControl::thread_helper (
            void * context ) [static]
```

This function is used in startThread() to get a thread started with the function controlfunc(). In startThread(), thread_helper() is passed as a parameter of pthread_create. pthread_create is a C function and does not understand the context ("this") of a function it is passed. This helper function circumvents this problem by itself being static while still calling the appropriate controlfunc().

Parameters

context	The context of the function

E.3.3.18 writeMeasurementsToUSB()

```
int DeviceControl::writeMeasurementsToUSB (
            int index )
```

Opens a filestream to a file called data.txt on the stick. The file is opened in "append" mode, such that existing data in the file is not erased. Writes the length value from the Length struct in the vLength vector that the index parameter points to to the file. This is followed by (on the same line, separated by a tab) the measurement values of the Measurement structs in vMeasurements. The line ends with a line break and the file is closed.

Parameters

index	An index of which length (from vLengths) has been measured

Returns

An integer: success is 0

E.3.3.19 writePressurePointsToUSB()

`int DeviceControl::writePressurePointsToUSB ( )`

Mounts the USB stick and opens a filestream to a file called data.txt on the stick. The file is opened in "append" mode, such that existing data in the file is not erased. The pressure values of the Measurement structs in vMeasurements are transfered to a single line in this file, separated by tabs. The line ends with a line break, the file is closed and the USB stick us unmounted.
Returns

An integer: success is o

The documentation for this class was generated from the following files:

- deviceControl.h
- deviceControl.cpp

F Test Measurement Data

F.1 Data.txt of muscle 1

	0.35	0.6	0.85	1.1	1.35	1.4625	1.575	1.6875	1.8	1.975	2.15	2.325	2.5	2.8125	3.125	3.4375	3.75	4.5	5.25	6	6.75
296:	94	132	192	250	317	356	383	401	436	488	519	564	614	707	794	864	951	1132	1343	1544	1723
296:	77	134	195	259	314	349	371	407	426	483	532	576	624	695	783	859	943	1140	1341	1534	1714
296:	72	127	194	260	328	348	378	408	438	483	520	574	618	705	781	857	947	1132	1340	1530	1727
296:	75	128	184	255	325	354	373	412	427	480	514	558	619	690	792	851	936	1147	1340	1521	1724
296:	80	132	189	253	325	343	375	405	422	481	523	570	617	700	788	867	938	1141	1342	1531	1722
296:	72	128	189	250	301	344	374	402	434	479	523	580	615	688	780	861	946	1147	1342	1541	1730
296:	69	129	177	244	315	345	373	403	430	476	521	570	622	697	779	866	950	1132	1347	1535	1728
296:	71	129	173	256	314	348	373	409	425	463	509	569	622	698	771	861	947	1134	1346	1534	1735
296:	57	119	189	250	303	341	368	400	431	466	521	568	605	697	779	862	945	1142	1341	1532	1740
296:	70	128	189	250	313	342	371	400	424	463	520	567	615	695	778	868	943	1142	1344	1536	1729
282:	0	0	0	0	85	110	120	156	181	222	255	292	335	407	470	527	614	769	935	1095	1262
282:	0	0	0	0	97	111	142	162	188	224	256	293	338	406	482	551	610	772	925	1099	1256
282:	0	0	0	0	97	124	134	177	192	239	256	307	333	409	477	544	616	776	938	1099	1250
282:	0	0	0	0	99	111	148	170	204	232	261	307	338	398	473	538	620	765	932	1098	1261
282:	0	0	0	0	102	116	151	158	196	236	280	307	353	406	482	549	610	790	928	1106	1261
282:	0	0	0	0	104	133	147	175	196	226	271	296	349	416	493	552	627	768	953	1106	1255
282:	0	0	0	0	107	130	151	186	200	236	273	311	349	403	480	540	629	782	942	1107	1282
282:	0	0	0	0	106	130	155	164	195	234	260	314	356	418	475	552	622	788	946	1109	1265
282:	0	0	0	0	108	119	145	182	202	241	276	316	351	418	488	541	623	780	947	1110	1278
282:	0	0	0	0	101	133	142	179	189	226	270	304	351	422	490	549	624	780	935	1117	1279
267:	0	0	0	0	0	0	0	0	85	99	144	176	210	256	301	352	420	549	674	802	921
267:	0	0	0	0	0	0	0	0	83	107	149	179	210	255	302	364	415	552	686	807	930
267:	0	0	0	0	0	0	0	0	88	129	162	186	202	265	306	365	415	558	674	813	935
267:	0	0	0	0	0	0	0	0	97	126	156	195	222	268	311	375	420	543	686	807	937
267:	0	0	0	0	0	0	0	0	95	127	162	194	227	277	310	377	439	559	686	803	927
267:	0	0	0	0	0	0	0	0	98	128	145	197	217	265	324	379	430	559	687	815	941
267:	0	0	0	0	0	0	0	0	100	136	166	180	219	267	318	378	417	562	675	802	939
267:	0	0	0	0	0	0	0	0	94	122	167	188	219	263	331	381	419	562	675	807	943
267:	0	0	0	0	0	0	0	0	112	130	151	188	220	274	337	382	433	563	690	823	943
267:	0	0	0	0	0	0	0	0	96	121	162	192	222	260	318	369	420	564	693	817	939
250:	0	0	0	0	0	0	0	0	0	0	0	0	93	136	176	214	262	350	453	541	632
250:	0	0	0	0	0	0	0	0	0	0	0	0	104	139	178	220	258	356	455	542	646
250:	0	0	0	0	0	0	0	0	0	0	0	0	108	141	180	207	260	355	450	544	636
250:	0	0	0	0	0	0	0	0	0	0	0	0	94	144	181	226	263	357	459	532	634
250:	0	0	0	0	0	0	0	0	0	0	0	0	91	145	190	224	249	346	452	545	638
250:	0	0	0	0	0	0	0	0	0	0	0	0	109	144	186	226	251	347	454	554	642
250:	0	0	0	0	0	0	0	0	0	0	0	0	107	146	187	235	265	364	456	548	638
250:	0	0	0	0	0	0	0	0	0	0	0	0	115	149	191	229	268	362	458	553	645
250:	0	0	0	0	0	0	0	0	0	0	0	0	101	156	176	221	270	363	458	553	631
250:	0	0	0	0	0	0	0	0	0	0	0	0	109	159	192	216	261	352	459	554	647
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	161	209	274	343
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	86	151	209	286	350
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	101	163	225	273	357
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	109	171	233	279	334
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	91	151	227	289	348
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	170	238	293	355
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	91	169	221	281	359
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	92	169	220	295	357
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	110	169	227	296	355
237:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	107	162	219	296	356
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	94
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	95
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	94
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	95
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	96
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	87
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	107
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	84
222:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	89

F.2 Data.txt of muscle 2

	0.3	0.575	0.85	1.125	1.4	1.5375	1.675	1.8125	1.95	2.125	2.3	2.475	2.65	2.95	3.25	3.55	3.85	4.575	5.3	6.025	6.75
296:	51	135	178	250	317	366	393	437	471	515	562	609	655	732	811	887	959	1149	1347	1534	1731
296:	54	116	191	238	304	352	394	418	469	497	553	602	644	730	788	882	959	1152	1350	1531	1720
296:	57	114	181	249	319	358	393	423	460	506	551	599	641	717	800	872	949	1150	1343	1529	1711
296:	40	103	173	233	318	349	380	433	451	499	543	600	638	723	808	868	957	1159	1340	1522	1708
296:	54	107	185	247	313	356	390	433	462	506	551	601	644	715	799	878	968	1141	1343	1531	1726
296:	52	105	180	239	323	357	383	424	453	498	543	601	634	720	800	866	958	1135	1328	1529	1730
296:	41	116	180	257	303	361	390	414	468	496	561	596	642	713	811	877	968	1149	1343	1525	1717
296:	49	113	166	247	315	362	390	419	446	513	549	599	641	722	802	885	948	1134	1335	1535	1717
296:	39	103	175	249	317	352	386	433	447	514	550	600	634	725	801	887	960	1160	1341	1529	1713
280:	0	0	0	0	84	99	142	164	198	231	278	320	362	416	489	545	621	776	925	1085	1250
280:	0	0	0	0	97	126	155	195	203	243	277	325	364	430	493	559	623	780	924	1092	1244
280:	0	0	0	0	87	121	167	189	216	256	277	331	374	441	494	554	630	787	940	1095	1235
280:	0	0	0	0	112	119	151	197	220	254	279	331	368	444	499	563	627	784	951	1101	1251
280:	0	0	0	0	104	133	170	191	229	249	287	333	371	435	491	558	622	778	943	1100	1254
280:	0	0	0	0	110	133	168	196	222	246	306	342	364	437	507	566	630	780	945	1099	1253
280:	0	0	0	0	107	137	152	182	225	260	303	338	382	443	503	567	632	790	953	1097	1267
280:	0	0	0	0	101	139	164	198	212	264	311	338	379	429	506	570	639	792	939	1110	1262
280:	0	0	0	0	109	141	164	186	227	270	296	330	376	436	498	572	644	788	947	1095	1256
265:	0	0	0	0	0	0	0	0	103	125	150	180	213	257	303	357	416	528	640	768	875
265:	0	0	0	0	0	0	0	0	97	134	146	195	200	274	316	359	406	521	648	772	894
265:	0	0	0	0	0	0	0	0	97	131	160	195	224	267	316	366	417	532	656	774	884
265:	0	0	0	0	0	0	0	0	101	132	162	191	222	273	321	369	404	538	657	769	900
265:	0	0	0	0	0	0	0	0	106	129	165	204	212	271	314	357	420	533	654	766	902
265:	0	0	0	0	0	0	0	0	94	125	165	186	211	275	323	373	422	551	646	779	908
265:	0	0	0	0	0	0	0	0	108	127	170	186	225	273	321	363	423	550	662	781	900
265:	0	0	0	0	0	0	0	0	118	131	169	196	219	260	316	381	423	538	649	773	906
265:	0	0	0	0	0	0	0	0	111	139	170	198	234	272	314	385	427	543	663	775	901
250:	0	0	0	0	0	0	0	0	0	0	0	0	89	135	158	211	248	329	410	494	607
250:	0	0	0	0	0	0	0	0	0	0	0	0	104	141	177	213	242	340	434	513	590
250:	0	0	0	0	0	0	0	0	0	0	0	0	105	143	179	216	250	338	413	516	603
250:	0	0	0	0	0	0	0	0	0	0	0	0	97	128	179	216	251	338	428	513	601
250:	0	0	0	0	0	0	0	0	0	0	0	0	107	143	168	210	253	338	431	517	593
250:	0	0	0	0	0	0	0	0	0	0	0	0	108	131	183	209	265	335	418	504	592
250:	0	0	0	0	0	0	0	0	0	0	0	0	95	139	173	214	244	345	433	521	599
250:	0	0	0	0	0	0	0	0	0	0	0	0	97	145	179	220	256	332	438	523	607
250:	0	0	0	0	0	0	0	0	0	0	0	0	109	148	183	230	249	331	422	523	608
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	88	139	209	270	336
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	94	162	212	267	321
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	158	215	284	336
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	96	165	221	270	336
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	167	206	271	328
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	145	206	288	338
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	151	208	278	340
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	99	162	220	282	328
235:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	101	162	229	271	339
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	81
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	89
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	76
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	94
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	89
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	87
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	90
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97
221:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	92

G MATLAB Script for Data Evaluation

Listing 2: readDataFile_PlotScript.m

```
clear all; close all; clc;
% Plot the static force map of a festo muscle actuator
% Data must be generated by the test stand of M. Drummond (TU Berlin)
% Author: Mirco Martens (TU Berlin)
% Date: 02.11.2016

%% Get data from file
filename = 'data2.txt';
data = readDataFile_StaticForceMap(filename);

%% Prepare data for 3d-plot
numberMeasurementsPerLength = sum(data(:,1)==data(2,1));
musclePressure = data(1,2:end);
muscleLength = flipud(unique(data(2:end,1)));
data = data(2:end,2:end);
muscleForce = zeros(length(muscleLength),length(musclePressure));
rowNumber = 1;
for i=1:length(muscleLength)
   midValVec = zeros(1,length(musclePressure));
   for j=1:numberMeasurementsPerLength
       midValVec = midValVec + data(rowNumber,:);
       rowNumber = rowNumber + 1;
   end
   muscleForce(i,:) = (midValVec / numberMeasurementsPerLength);
end

%% 3d plot
f = figure;
set(f,'Units', 'normalized', 'Position', [0.1, 0.1, 0.5, 0.7]);
surf(musclePressure,muscleLength,muscleForce);
xlabel('Pessure [bar]');
ylabel('Length [m]');
zlabel('Force [N]');
```