MeArm Pi
Technical Overview

Table of Contents

Revision History....3
Disclaimer....3
Introduction....4
Servos....4
  Servos in General....4
  MeArm Pi Servos....5
I2C....6
  I2C in General....6
  MeArm Pi PCF8591 I2C Device....6
Buttons and LEDs....7
Kinematics....7
  Kinematics in General....7
  MeArm Pi Kinematics....7
Inverse Kinematics....9
  Inverse Kinematics in General....9
  MeArm Pi Inverse Kinematics....9
Dynamics....10
  Dynamics in General....10
  MeArm Pi Dynamics....10
MeArm Pi Setup and Calibration....10
  Foreword....10
  Warning....10
  Setup....10
  Can You Get There From Here?....12
  Calibration in General....12
  MeArm Pi Calibration....12
For Further Consideration....13
  Issues....13
  Additional Information....14
Figures....15
  Figure 1 – Servo Control....15
  Figure 2 – Base Dimensions....16
  Figure 3 – Lower Arm Vertical....17
  Figure 4 – Lower Arm Vertical, Upper Arm Fully Extended....18
  Figure 5 – Lower Arm Vertical, Upper Arm Fully Retracted....19
  Figure 6 – Lower Arm Fully Retracted....20
  Figure 7 – Lower Arm Fully Retracted, Upper Arm Fully Extended....21
  Figure 8 – Lower Arm Horizontal....22
  Figure 9 – Lower Arm Horizontal, Grip Resting On Mounting Surface....23
  Figure 10 – Arms Positioned for Maximum Reach on Mounting Surface....24

Figure 11 – Arms Positioned for Minimum Reach on Mounting Surface.....25
Figure 12 – Simple Kinematic Model.....26
Figure 13 – Positioning Template (not to scale).....27
Figure 14 – Setup Graph (Conceptual).....28
Appendix 1: Sample Code.....29
Commented Code.....29
Code (for cutting and pasting).....33
How to Run the Code.....34

Revision History

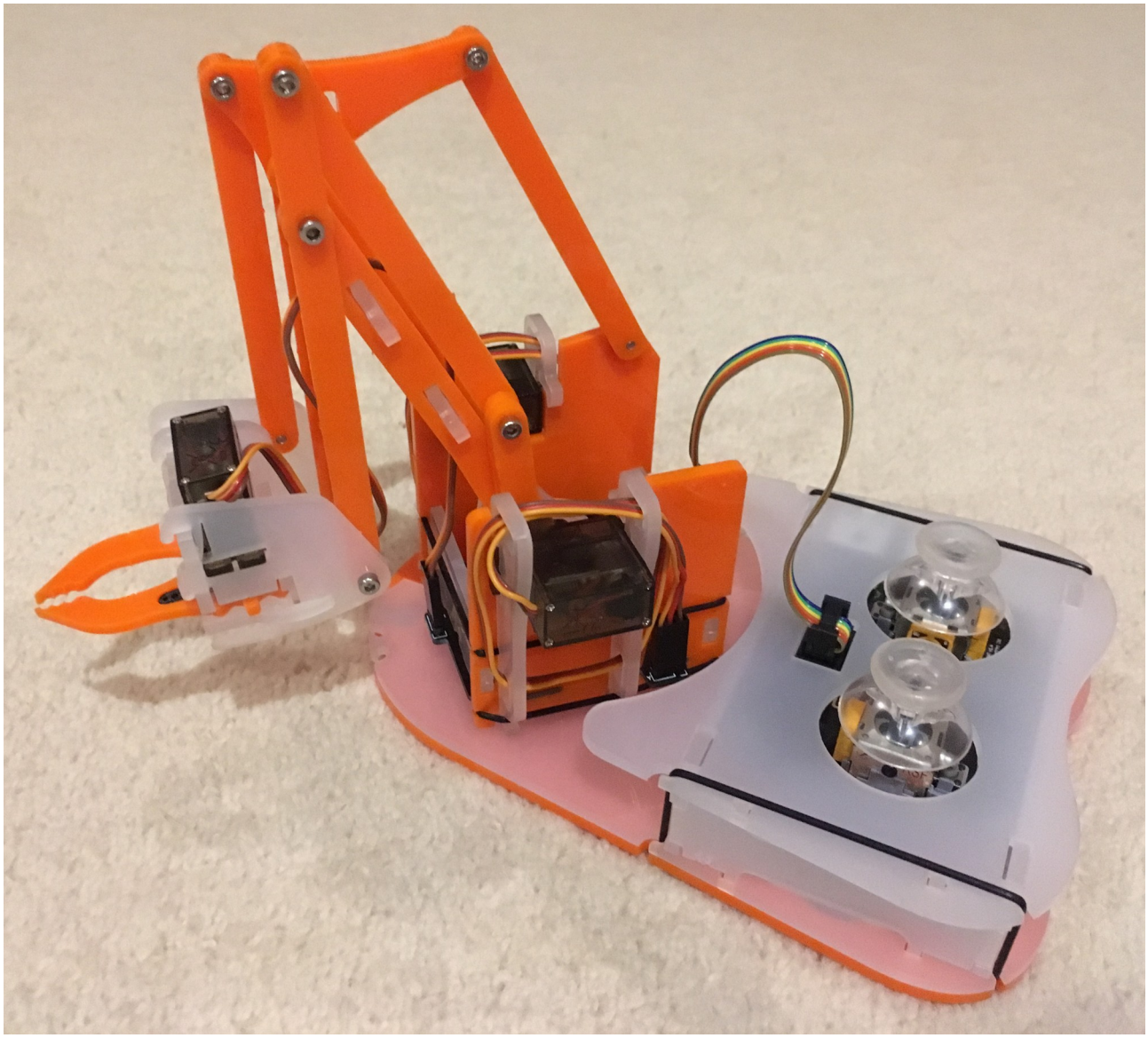
Date	Who	Version	Description
1/12/2017	Neil Higgins	0.1DRAFT	First draft
14/1/2018	Neil Higgins	0.2DRAFT	Changed “python” to “python3” in Appendix 1 run instructions

Disclaimer

This document is intended to supplement, not supersede, upstage or override, official MeArm Pi documentation. It is correct, E&OE (Errors and Omissions Excepted), i.e. only to the extent that it is not incorrect. Where this document conflicts with official MeArm Pi documentation, the latter should be assumed to be correct. This document is provided free of charge and without encumbrance, provided that the user assumes all responsibility for consequent outcomes. The author *may* issue updates and/or corrections based reader feedback to the blog posting where this document was originally posted. All constructive comments are welcome.

Introduction

The MeArm Pi is the synthesis of a Raspberry Pi computer and a robot arm (a MeArm). The MeArm Pi includes a Raspberry Pi-compatible HAT (printed circuit board used for interfacing and interconnection). The Raspberry Pi completely controls the motion of the arm; as such the arm cannot function without the Raspberry Pi and associated software.



Servos

Servos in General

The MeArm Pi HAT “passes through” four servo control signals from the Raspberry Pi’s general purpose input/output (GPIO) pins to the robot arm. One signal controls the angular position of the base, two control the angular positions of the lower and upper arms (hence the position of the grip), and the other controls the opening of the grip.

Each servo control signal comprises a train of pulses. A pulse is emitted every twenty milliseconds (i.e. 50 pulses per second) and its duration (width) varies between a minimum of 0.5 – 1.0 millisecond and a maximum of 2.0 – 2.5 milliseconds. The angular position of the servo's output shaft is determined by the pulse width – the longer the pulse, the larger the angular position:

- The minimum position of a servo corresponds with the minimum control signal pulse width to which the servo will respond (a smaller pulse width may be received, to no effect).
- The maximum position of a servo corresponds with the maximum control signal pulse width to which the servo will respond (a larger pulse width may be received, to no effect).
- The difference between the maximum and the minimum position is called the range. It is not necessary to use the whole range, and indeed this may be impossible due to mechanical limitations (as we will see).

See Figure 1.

Servos are only roughly standardised. Operation is usually linear, with the "neutral" position at the midpoint of the range, corresponding with a pulse width of about 1.5 milliseconds. Each make and model has characteristic values for the minimum and maximum control signal pulse widths, the corresponding minimum and maximum angular positions, the direction of rotation from minimum to maximum, speed, torque, dimensions, electrical ratings, etc.

The absolute angular position of a servo horn depends on how the horn is positioned on the shaft. The shaft has a spline which allows the horn to be positioned at regular intervals over a 360° circle. Not surprisingly, the number of teeth on the spline also varies between makes and models – less teeth makes for coarser setup and vice versa.

Each "application" (specific use of a servo in a robot arm or elsewhere) has different requirements for the minimum and maximum angle. Put these together and we get a specific set of theoretical minimum and maximum pulse widths for each servo in the application. These may need to be tweaked by "calibration" depending on manufacturing precision and repeatability.

MeArm Pi Servos

The author has been unable to identify the make and model of the MeArm Pi servos, hence has been unable to access the manufacturer's data sheets.

The four servos appear to be identical, except that the base servo has a double-ended horn. The following characteristics have been inferred from observations and inspection of a software sample provided by Mime Industries:

- The angular position increases from minimum to maximum in an anticlockwise direction, viewed from the horn side.
- The minimum pulse width is 530 microseconds / 0.53 milliseconds.
- The maximum pulse width is 2 400 microseconds / 2.4 milliseconds.
- The range is 180°.
- The number of teeth on the spline is 20, for a setup precision of 18°.

It is tempting to assume that each servo is manufactured with the spline "precisely" aligned to the major axes of the housing when in the minimum position, i.e. such that the horn can be positioned at "precisely" 12 o'clock, 3 o'clock, 6 o'clock and 9 o'clock (and intermediate 18° increments) relative to the housing. Observations suggest that this is not a safe assumption.

Conventional nomenclature for a robot arm would have the upper arm nearest to the “shoulder” and the lower arm nearest to the “hand”. The MeArm defies this convention, referring to the lower arm as that which is physically lowest, and the upper arm as that which is physically highest. To avoid confusion, I will use MeArm nomenclature. The “base” is the rotating platform on which the arm is mounted.

Low-level details of servo control are handled by the Raspberry Pi’s GPIO hardware in combination with the chosen software library.

I2C

I2C in General

The MeArm Pi HAT also contains a PCF8591 chip which acquires left-right and up-down signals from the two joysticks. The PCF8591 chip uses the I2C protocol to communicate with the Raspberry Pi. Specific GPIO pins on the Raspberry Pi can be configured as I2C signals.

Each I2C transaction transfers data between a master device and a slave device. Typically the master will send a command, and the slave will respond. In the MeArm Pi the Raspberry Pi is the master and the PCF8591 chip on the HAT is the slave.

Transferring data involves manipulating the I2C physical signals to:

- Signify the start of a data transmission
- Address the slave device
- Transfer data from master to slave and/or slave to master, and
- Signify (or not) successful transfer of the data (called acknowledgment)

MeArm Pi PCF8591 I2C Device

The PCF8591 is a four channel analog-to-digital (A/D) converter, also having a single digital-to-analog (D/A) channel. It has a fixed base address and a configurable 3-bit address offset which allows up to 8 identical devices to share a common I2C bus. The base address is hexadecimal 48 (0x48). On the MeArm Pi there is only one such device; its offset is configured to zero by strapping all three address offset pins to 0V.

Acquiring analog input data involves, firstly, configuring the device:

- Addressing the device in write mode (master → slave, one byte)
- Sending a control byte to set the conversion mode and select the required channel (master → slave, one byte)

Secondly, acquiring the data:

- Addressing the device in read mode (master → slave, one byte)
- Reading a channel (slave → master, two* or more bytes); multiple channels can be read if the auto-increment bit in the control byte has been set

* Note that the first byte returned after starting acquisition is the value of the *previous* channel (or a dummy value of 0x80, after a reset). This is due to the nature of the A/D conversion process in the PCF8591.

Low-level details of I2C signal manipulation are handled by the Raspberry Pi's GPIO hardware in combination with the chosen software library.

Buttons and LEDs

The MeArm Pi HAT "passes through" three outputs and two inputs from the Raspberry Pi's GPIO pins to the robot arm. The three outputs drive the tricolour LED and the two inputs receive the joystick buttons.

Kinematics

Kinematics in General

"Kinematics" is defined as the branch of mechanics concerned with the motion of objects without reference to the forces which cause the motion; the features or properties of the motion of an object.

In this section we will derive a formula for the position of the grip relative to a reference point on the MeArm base plate, as a function of the servo positions.

In the next section we will look at the "inverse kinematic" question: What servo positions should we choose in order to put the grip at a desired position?

MeArm Pi Kinematics

The MeArm is a "double parallelogram" mechanism. The advantage of this arrangement is that the servos for both the lower and the upper arm can be mounted on the base while still allowing them to be moved independently.

The following figures were drawn using an basic CAD package (Autosketch 3 in DOSBox under Ubuntu Linux). The MeArm is viewed from the upper servo side. A quick search of the web has revealed free kinematics software that could potentially do the same job much more efficiently.

Dimensions were taken off a MeArm to an accuracy of about ±0.5 mm. Exact dimensions could be taken from the cutting file (see Further Information). Derived dimensions are represented with vastly higher precision than the mechanism can achieve.

The angles of the servo horns are indicated as follows:

- Lower arm servo relative to 12 o'clock, increasing anticlockwise. "Extension" is associated with increasing rotation.
- Upper arm servo relative to 6 o'clock, increasing clockwise (anticlockwise viewed from the horn side). "Extension" is associated with increasing rotation.

The horizontal dimension for the grip is roughly to the centre of the space between the jaws. For the distance to the serrations, add another 10mm.

Figure 2 shows the critical dimensions of the base. Note that pivot points / axes are shown, not the base outline.

Figure 3 shows the lower arm in the vertical position. Figure 4 adds the upper arm at full extension. Extension of the upper arm is limited by interference between the top and bottom upper arms. The upper arm rotor could (without other constraints) extend until it interferes with the base servo

housing, as shown by the dashed lines; astute observers will have noted a recess in the base servo clamp that catches the screw connecting the upper arm rotor and actuator, preventing this from happening. Figure 5 adds the upper arm at full retraction. Retraction of the upper arm is limited by interference between the lower arm and the upper arm actuator.

Figure 6 shows the lower arm at full retraction. Retraction is limited by interference between the rear lower arm and the bottom upper arm, at the bell crank. Figure 7 adds the upper arm at full extension. Retraction of the upper arm in this case is limited by interference between the grip and the body of the base (the servo positions cannot been derived because this kinematic model only deals with movable parts).

Figure 8 shows the lower arm in the horizontally extended position. Figure 9 adds the upper arm with the grip resting on the mounting surface.

Figure 10 shows that by depressing the lower arm below the horizontal it is possible to increase the overall reach slightly.

Figure 11 shows the upper arm at full retraction with the grip resting on the mounting surface. Retraction of the upper arm is limited by interference between the top and bottom upper arms.

Because the forward and rear lower arms are the same length and are fixed to points with the same separation, they form a parallelogram; hence the orientation of the bell crank remains constant. Similarly, because the top and bottom upper arms are the same length and are fixed to points with the same separation, they form another parallelogram, so that the orientation of the grip (which is tied to the orientation of the bell crank) also remains constant.

We can now derive formulae for the horizontal and vertical positions of the grip, based on the servo angles.

The following symbols will be used:

l	–	the angle of the lower servo, increasing anticlockwise from 12 o'clock
u	–	the angle of the upper servo, increasing clockwise (anticlockwise viewed from the horn side) from 6 o'clock
r	–	the radial position of the grip relative to the centreline of the base servo
z	–	the vertical position of the grip relative to the mounting surface
θ	–	the angle of the base (not the base servo) relative to neutral, increasing anticlockwise viewed from above
x	–	the position of the grip parallel to the neutral axis
y	–	the position of the grip perpendicular to the neutral axis
cos()	–	cosine function
sin()	–	sin function
sqrt()	–	square root function
arctan()	–	inverse tan function
arcsin()	–	inverse sin function
temp1	–	temporary variable
temp2	–	temporary variable

See Figure 12 and Figure 13.

Then:

$$r = 15 + 80 \sin(l) + 80 \sin(u) + 50 = 65 + 80 (\sin(l) + \sin(u)) \ldots (1)$$

$z = 53 + 80 \cos(l) – 80 \sin(u) = 53 + 80 (\sin(l) – \sin (u)) \ldots (2)$
$x = r \cos(\theta) \ldots (3)$
$y = r \sin(\theta) \ldots (4)$

Note that these formulae work perfectly well for negative values of l, as is the case when the lower arm is fully retracted.

There are some mechanical constraints:

- l cannot be less than -48.8°or more than 98.2°.
- u cannot be less than -9.4°or more than 126.5°.
- z cannot be negative (unless the MeArm Pi is mounted near the edge of a table so that the grip can descend below the surface of the table).

Inverse Kinematics

Inverse Kinematics in General

“Difficult” sums it up.

The term “degrees of freedom” means the number of independently variable factors affecting the range of states in which a system may exist, in particular any of the directions in which independent motion can occur. As applied to a robot arm, it means the number of independently controllable distances or angles in the arm.

A higher number of degrees of freedom increases the likelihood that an object can be reached and manipulated by the arm; it also increases, disproportionately, the complexity of positioning and motion planning. For positioning, this is because there may be several combinations of controllable distances and/or angles that put the grip in the desired position. Some of these alternative solutions may be “better” than others for a range of reasons.

For motion planning, it is because (even ignoring obstructions) some pathways between the start point and the end point may be highly non-optimal, or even physically impossible to implement.

MeArm Pi Inverse Kinematics

In a real-world application, we may wish to derive the servo angles (l, u, θ) from the Cartesian coordinates (x, y, z) of the grip in order to position the arm. To do so, we can proceed as follows:

$r = \text{sqrt} (x^2 + y^2) \ldots (5)$
$\theta = \arctan(y / x) \ldots (6)$

then solve equations (1) and (2) for l and u using r and z as follows:

$temp1 = l – u = 2 * \arctan((53 – z) / (r – 65)) \ldots (7)$
$temp2 = l + u = 2 * \arcsin((r – 65) / (160 * \cos(temp1 / 2)) \ldots$ (8a), or alternatively
$temp2 = l + u = 2 * \arcsin((53 – z) / (160 * \sin(temp1 / 2)) \ldots$ (8b)
$l = 0.5 * (temp1 + temp2) \ldots (9)$
$u = 0.5 * (temp2 - temp1) \ldots (10)$

The derivation of the above formulae relies on the following trigonometric identities:

$\tan(x) = \sin(x) / \cos(x) \ldots (11)$
$\sin(l) + \sin (u) = 2 * \sin((l + u) / 2) * \cos((l – u) / 2) \ldots (12)$
$\cos(l) – \cos (u) = -2 * \sin((l + u) / 2) * \sin ((l – u) / 2) \ldots (13)$

Dynamics

Dynamics in General

“Dynamics” is defined as the branch of mechanics concerned with the motion of objects under the action of forces. Mechanical forces are associated with:

- Friction
- The acceleration of massive bodies (the arm itself, and the object being manipulated)
- Gravity (which for the purists is equivalent to an acceleration)

Although Newton’s laws straight forwardly cover all mundane earthly situations, there is again room for the use of advanced mathematical techniques to improve computational simplicity and accuracy.

As applied to robot arms, the central problem is often that of moving an object along a desired path without damaging the object (by rough handling) or exceeding the ratings of the robot arm.

MeArm Pi Dynamics

This document does not address Me Arm Pi dynamics.

MeArm Pi Setup and Calibration

Foreword

The following may seem an unnecessarily elaborate analysis, given that the MeArm Pi is a hobby-class robot arm with obvious limitations in the achievable performance. It was carried out for the author’s education and is repeated here for what it is worth.

Warning

The Mime Industries setup (it is assumed) will have been conservatively designed to give satisfactory out-of-the-box performance with no end-user adjustment.

The following setup is independently derived from the kinematics described above. It is not based on the Mime Industries setup, and actually differs from it. It requires the end user to adjust the positions of the servo horns. Compatibility with the Mime Industries software is not guaranteed.

Setup

“Setup” (in this case) means the deciding:

- The as-assembled positions of the servo horns.
- The default values of software-based constants for controlling the position of the arm.

The servo positions inherent in the design are as follows.

Element	Relative to	Full retraction (degrees)	Full extension (degrees)	Design range (degrees)
Lower arm	12 o’clock	-48.8	98.2	147.0
Upper arm	6 o’clock	-9.4	126.5	135.9

Base	Neutral	-90	+90	180
Jaws	Open	0	90	90

To keep things simple initially, we will assume (contrary to observation) that each servo is indeed manufactured with the spline “precisely” aligned to the major axes of the housing when in the minimum position.

In cases where the design range is less than the range available from the servo, alternative setups may be possible. The simplest setups are derived first. The pros and cons of alternative setups will be considered later under Calibration.

For the lower arm:

- To allow the full retraction of -48.8° to be achieved, the horn must be assembled at least three spline pitches (54°) behind 12 o’clock with the servo in its minimum position.

1. The following rationale applies to all setups. Assuming the servo responds linearly, it will deliver 180 / (2.40 – 0.53) = 96.26 degrees of rotation per millisecond of pulse width. The minimum position of -54° will correspond with a pulse width of 0.53ms. To prevent the servo from retracting behind -48.8°, the pulse width should never be less than 0.53 + (-48.8 - (-54)) / 180 * (2.40 – 0.53) = 0.58ms.

- To prevent the servo from extending beyond 98.2 degrees, the pulse width should never be greater than 0.53 + (98.2 - (-54)) / 180 * (2.40 – 0.53) = 2.11ms.
- See Figure 14.

For the upper arm:

- To allow the full retraction of -9.4° to be achieved, the horn must be assembled at least one spline pitch (18°) behind 6 o’clock with the servo in its minimum position.
- To prevent the servo from retracting behind -9.4°, the pulse width should never be less than 0.53 + (-9.4 - (-18)) / 180 * (2.40 – 0.53) = 0.62ms.
- To prevent the servo from extending beyond 126.5 degrees, the pulse width should never be greater than 0.53 + (126.5 - (-18)) / 180 * (2.40 – 0.53) = 2.03ms.

For the base, because the design range is equal to the available range, only one setup is possible:

- The horn must be assembled on the relevant major axis with the servo in its minimum position.
- The minimum pulse with of 0.53 milliseconds will enable full negative rotation.
- The maximum pulse width of 2.40 milliseconds will enable full positive rotation.

For the jaws:

- If opting for a maximum opening of 90°, the horn must be assembled on the relevant major axis with the servo in its minimum position.
- The minimum pulse with of 0.53 milliseconds will enable full opening.
- The prevent the jaws from closing more than 90° and clashing, the pulse width must never be greater then 0.53 + 90 / 180 * (2.40 – 0.53) = 1.47ms.

We are fortunate that Mime Industries has provided example Python code, including a servo constructor of the form:

Servo({'pin': , 'min': , 'max': , 'minAngle': , 'maxAngle': });

where

- pin is the relevant GPIO pin number

- min and max are pulse widths in microseconds
- minAngle and maxAngle are angles in degrees

We can apply this to setup the MeArm Pi as follows:

```
lower = Servo({'pin': 17, 'min': 580, 'max': 2110, 'minAngle': -48.8, 'maxAngle': 98.2});
upper = Servo({'pin': 22, 'min': 620, 'max': 2030, 'minAngle': -9.4, 'maxAngle': 126.5});
base = Servo({'pin': 4, 'min': 530, 'max': 2400, 'minAngle': -90.0, 'maxAngle': 90.0});
grip = Servo({'pin': 10, 'min': 530, 'max': 1470, 'minAngle': 0, 'maxAngle': 90.0});
```

Can You Get There From Here?

If we apply these setups – not forgetting the horns (requiring partial disassembly and reassembly of the MeArm) – we can move the arm from one “cardinal” position to the next as shown in the figures. Appendix 1 contains the relevant stand-alone Python code and documentation.

Well, almost ...

Amongst other problems, we cannot move the arm smoothly, in one step, between the positions in Figures 10 and 11. This is because retraction of the upper servo causes the grip to (try to) go below the level of the mounting surface, before the upper arm has had time to retract. This problem can be partly addressed though adequate motion planning; however it is made worse on the author’s MeArm Pi because the lower spline is very poorly aligned to the major axes of the housing, pushing the grip even lower. For guaranteed outcomes, we also need to calibrate the arm.

Calibration in General

“Calibration” (in this case) refers to adjustment of the values of software based constants for controlling the position of the arm, in order to optimise the arm’s performance.

The need for calibration arises from several causes:

- Manufacturing variations in the components of the arm, including the servos. As previously mentioned, the spline does not appear to be precisely aligned, during manufacture, with the major axes of the servo housing. In a conventional application of this kind of servo, e.g. a model aircraft, this would not matter much – the whole range would not be used, and any error would be trimmed out before and/or during flight (trimming being a form of calibration).
- Backlash in the servo gears.
- “Slop” (for want of a better term) and deflection under load that are inherent in the mechanism.
- Task-specific adjustments.

By definition, then, calibration is specific to each arm, and the task to be performed.

MeArm Pi Calibration

Consider the situation where spline is not aligned at all during manufacture. Its minimum position could be anywhere in a range of ±9° from a major axis. For the lower arm, a nominal setting of three spline pitches behind 12 o’clock could put the horn anywhere between 45° and 63° from the relevant major axis. The latter could be calibrated out; the former is within the design operating range!

To overcome this, the horn could be set at four spline pitches behind 12 o'clock, but only if there is enough unused room at the opposite end of the available range to accommodate the extra offset. For cases like the base servo, where the design uses the entire available range, any form of calibration (say to ensure that the arm is centred for a given control signal pulse width) will result in a loss of working range.

In other words, depending on the quality of the components used, there may be a cyclic process of setup and calibration – hopefully one that eventually produces a satisfactory outcome.

Clearly calibration involves situation-specific adjustment of the setup constants derived above. This may be done by trial and error or, in a more professional situation, by using special tools and gauges.

For Further Consideration

Issues

Issue	In General	MeArm	Comment
There is nothing to prevent human injury this might result from a person interacting with the MeArm Pi.	In industrial settings, human intrusion into a robot's workspace is detected by light beams (and the like) and triggers a safety contingency such as an immediate shutdown.	It is doubtful that the MeArm has enough strength to cause serious injury.	Users are advised to keep themselves and their non-sentient offspring, pets, etc. clear while the MeArm Pi is operating.
There is nothing to prevent damage to the MeArm that might result from commanding servos to move beyond their kinematically allowable ranges. There is nothing to prevent damage to the MeArm that might result from attempting to move excessively massive objects.	In industrial settings, robots will be include "layered" protections such as limit switches, mechanical and electrical overload protections, and independent supervisory subsystems to prevent self-harm.	It is doubtful that the MeArm has enough strength to self-harm, albeit attempting to move excessive masses could result in overheated servo motors, stripped gears or broken structural members.	Users are advised to exercise forethought and judgement in their use of the MeArm Pi, respecting its limited capabilities.
When the MeArm to is commanded move to a new position, each servo operates independently, resulting in a jerky and non-optional path.	Motion control includes approaching, grasping, repositioning releasing and leaving an object. High quality motion control often requires position feedback from each	The MeArm has no position feedback. Provided that the MeArm is not overloaded, each servo should come close to its commanded position eventually; however	By breaking the desired path into segments, and by explicitly taking into account the mass of the object, a semblance of smooth operation may be achievable.

	degree of freedom, allowing the robot to compensate for the effects of inertia and gravitation, and thus achieve a predictable path for the object.	there is no innate coordination between them.	There is no limit to the types and degrees of refinement that the user may experiment with, but it might be worth doing some research before hand, in order to avoid “dry gulches”.
The MeArm Pi has no means of navigating spaces containing obstructions.	Many industrial robots are the same! They require a carefully controlled environment in which to work. Advanced robots can detect obstructions by “feel” (e.g. proximity sensors), LIDAR, SONAR or image processing and navigate around them.	A quick spin through the Mime Industries code provides a hint of future integration with Minecraft Pi. Minecraft Pi has the ability to model a space, and also the ability to detect collisions between objects within this space (i.e. before the physical arm is moved).	There’s nothing to prevent the user from trying his/her own approach to workspace observation and/or modelling. Go boldly!

Additional Information

(Links were correct at the time of access)

Mime Industries MeArm Pi - https://mime.co.uk/products/mearm-pi
MeArm Pi Google Group - https://groups.google.com/forum/#!forum/mearm
MeArm Pi cutting file - https://github.com/mimeindustries/MeArm/blob/v2/MeArm.dxf
Raspberry Pi HAT specification - https://github.com/raspberrypi/hats
Servo tutorial - https://learn.sparkfun.com/tutorials/hobby-servo-tutorial
Servo rotation direction - https://www.servocity.com/servo-direction-information
I2C tutorial - https://learn.sparkfun.com/tutorials/i2c
PCF 8591 device - https://www.nxp.com/docs/en/data-sheet/PCF8591.pdf
MeArm Pi laser cutting file - https://github.com/mimeindustries/MeArm/blob/v2/MeArm.dxf
Free kinematics software - http://www.kinematics.com/products/robotassist.php
Inverse kinematics - https://en.wikipedia.org/wiki/Inverse_kinematics
Python - https://www.manning.com/books/the-quick-python-book-second-edition

Figures

Figure 1 – Servo Control

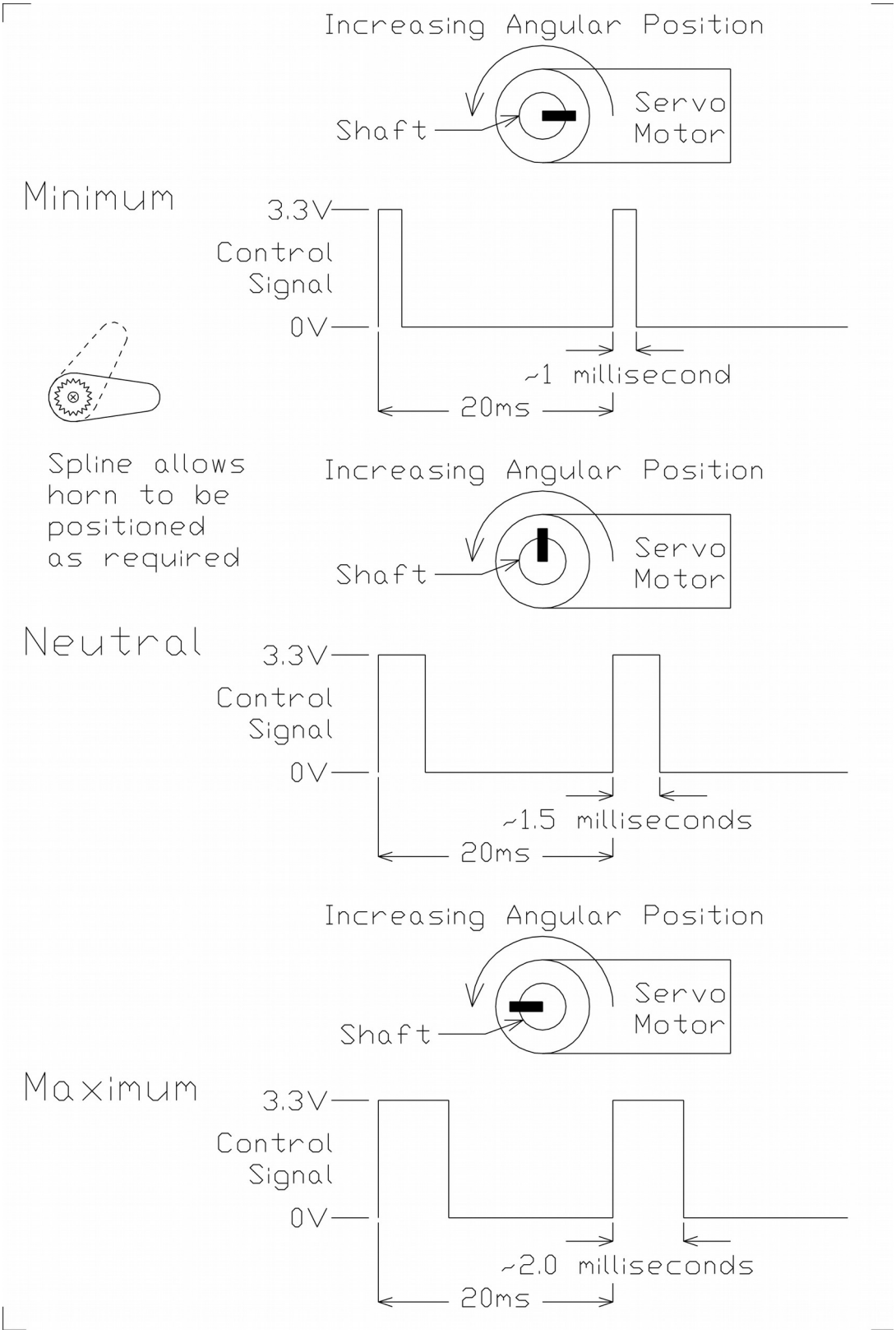


Figure 2 – Base Dimensions

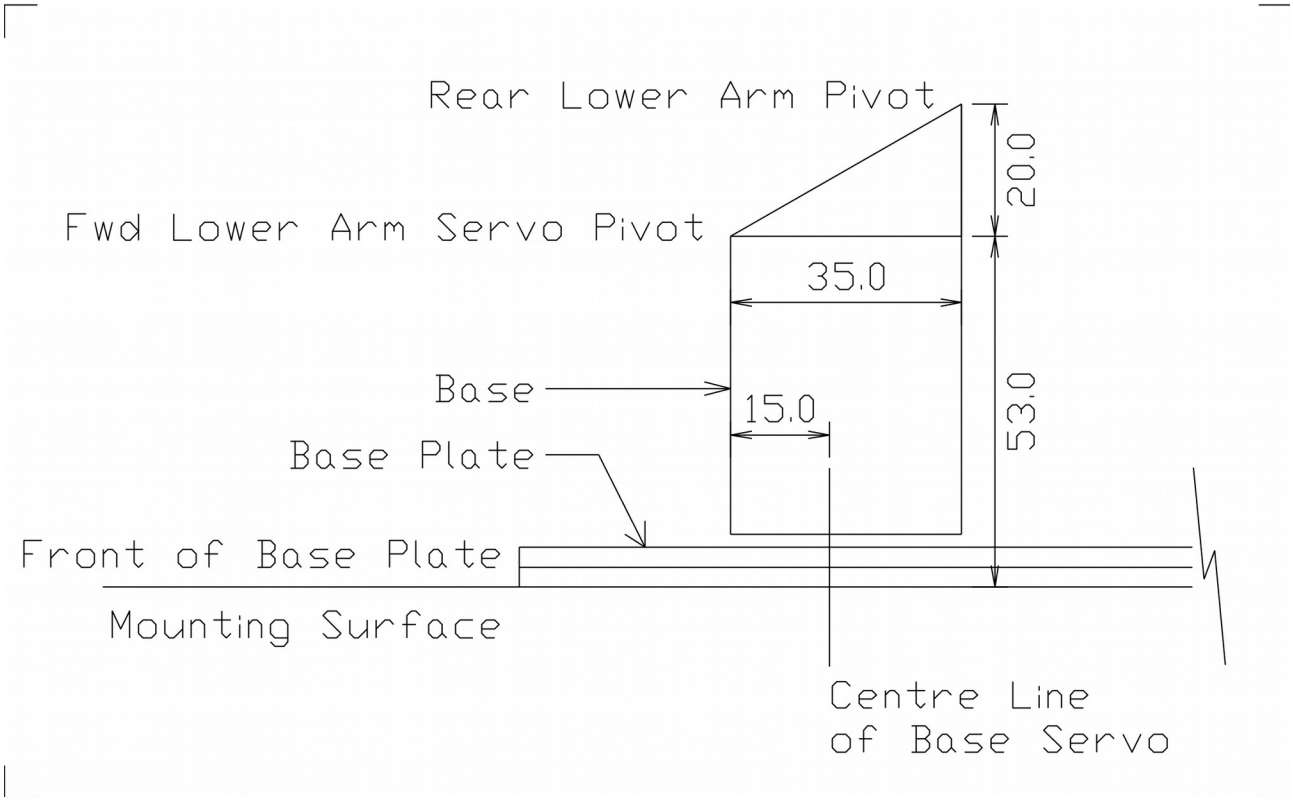


Figure 3 – Lower Arm Vertical

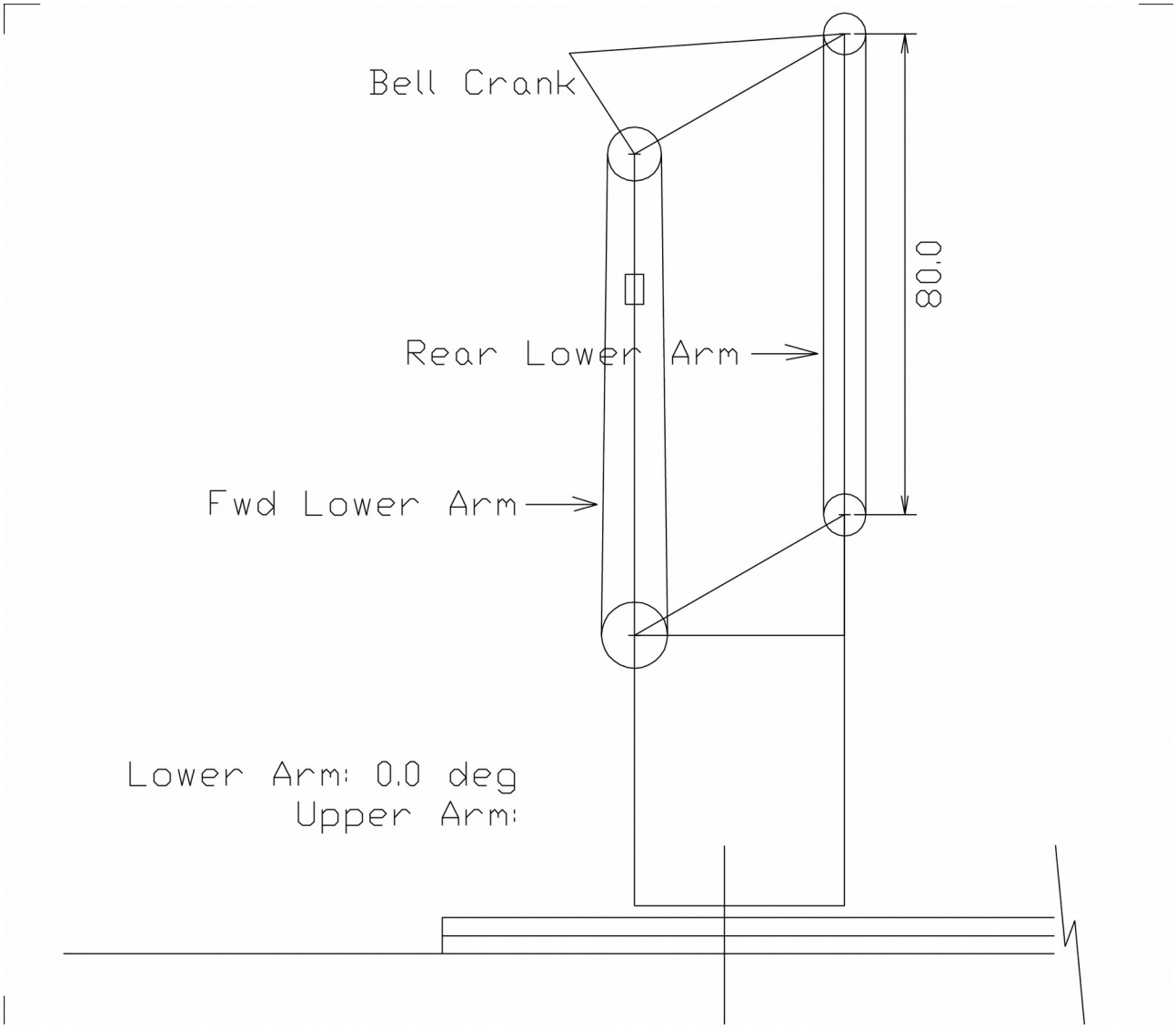


Figure 4 – Lower Arm Vertical, Upper Arm Fully Extended

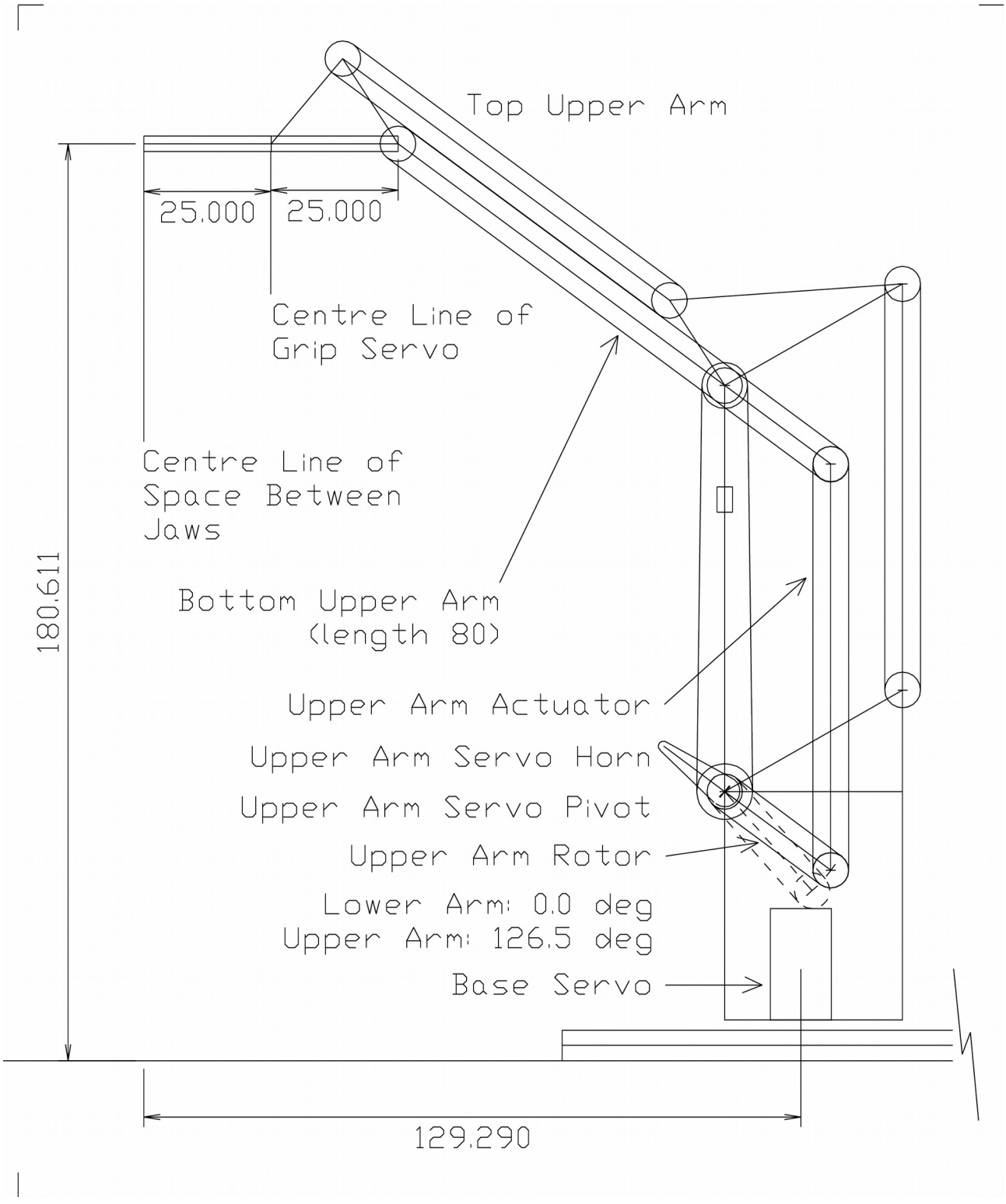


Figure 5 – Lower Arm Vertical, Upper Arm Fully Retracted

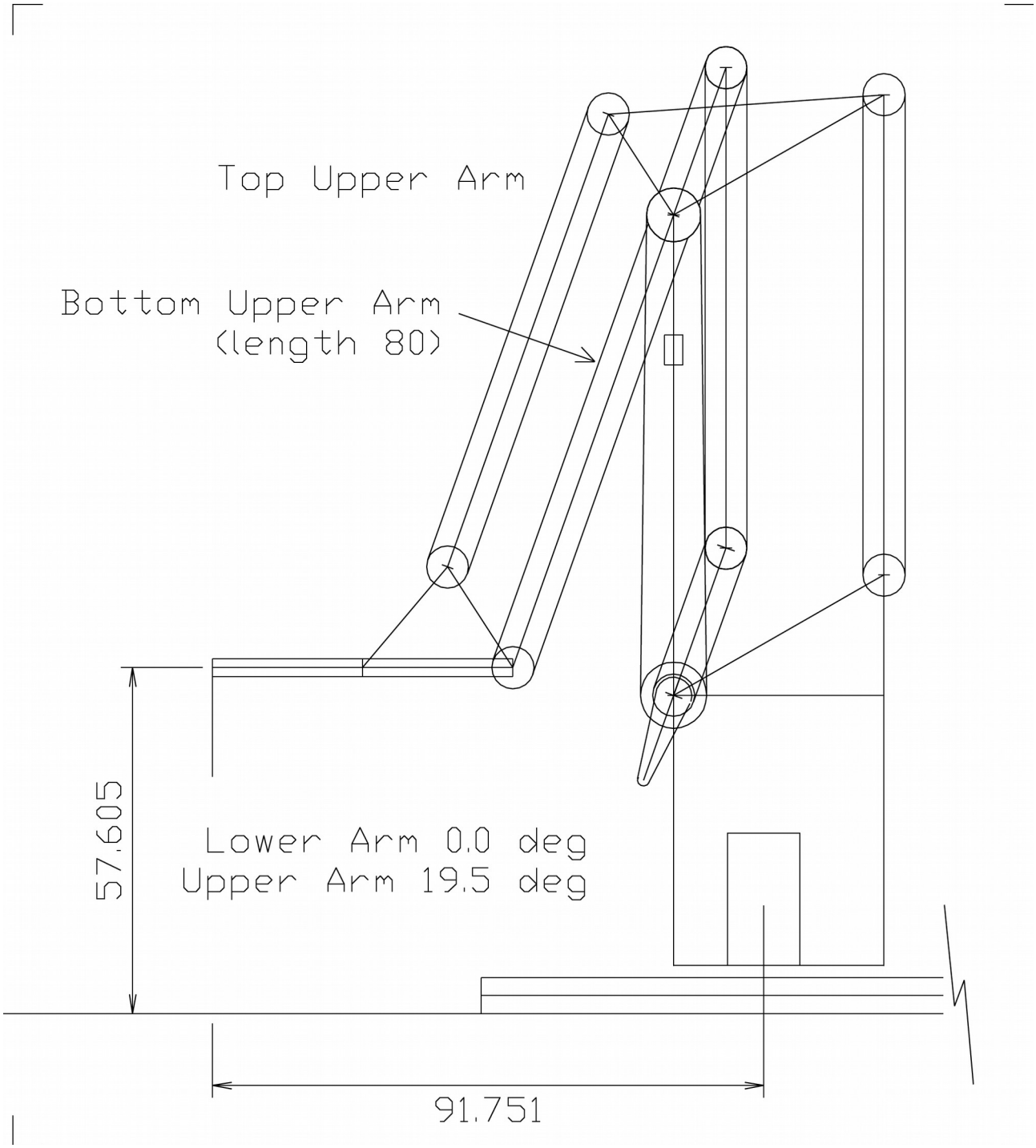


Figure 6 – Lower Arm Fully Retracted

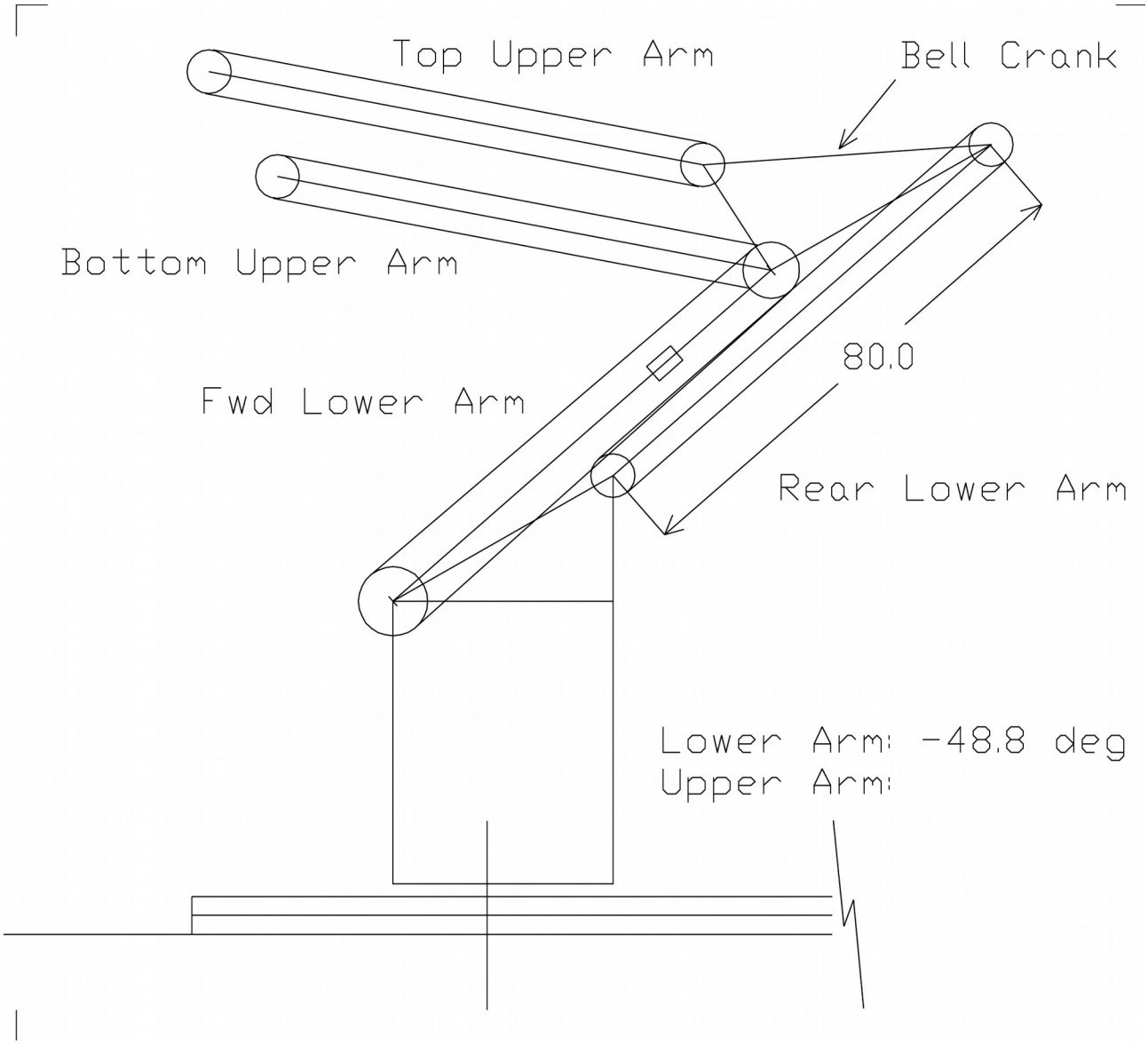


Figure 7 – Lower Arm Fully Retracted, Upper Arm Fully Extended

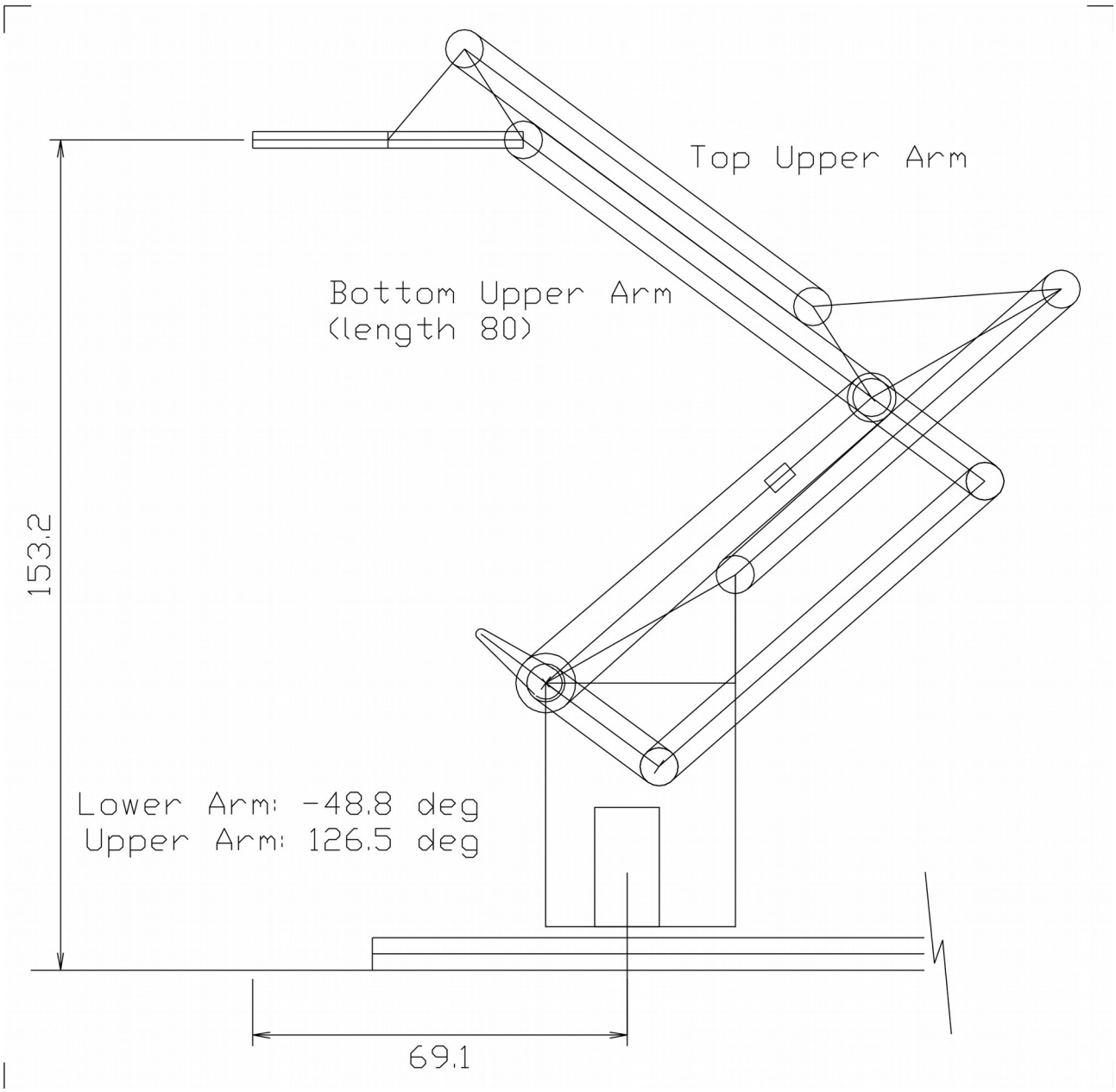


Figure 8 – Lower Arm Horizontal

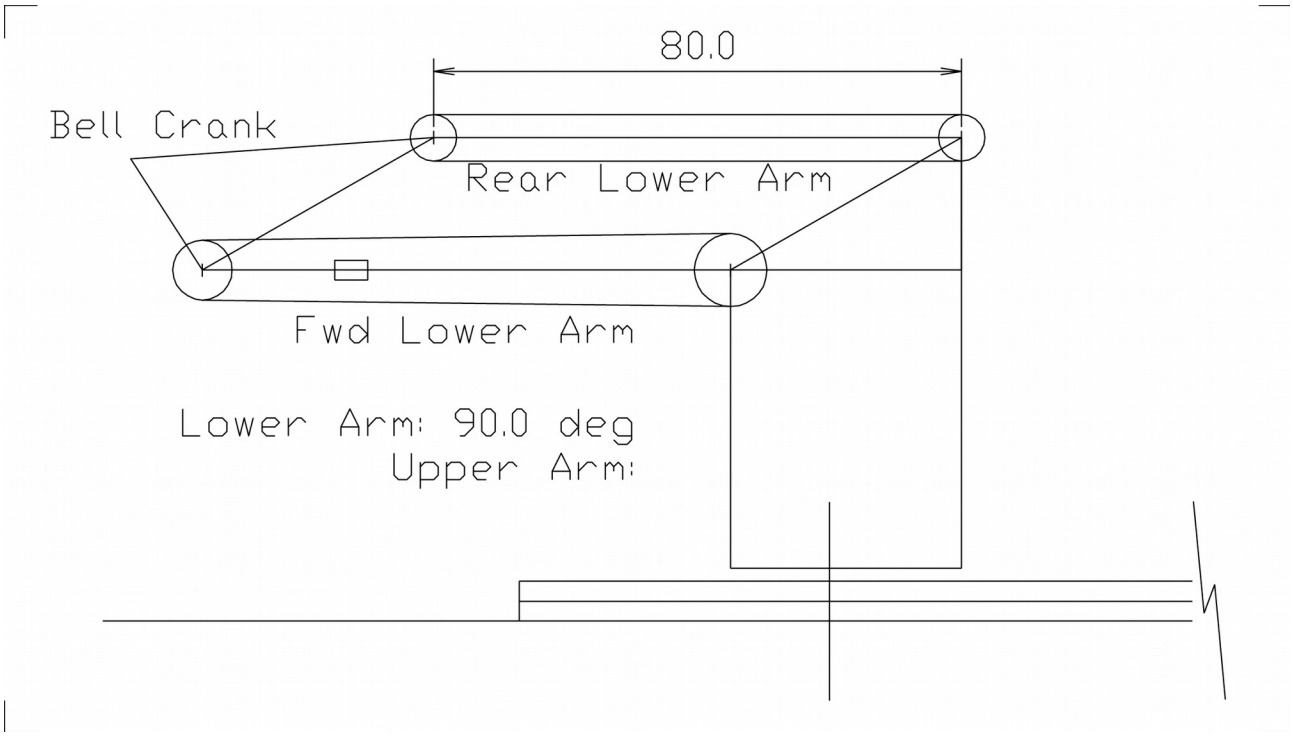


Figure 9 – Lower Arm Horizontal, Grip Resting On Mounting Surface

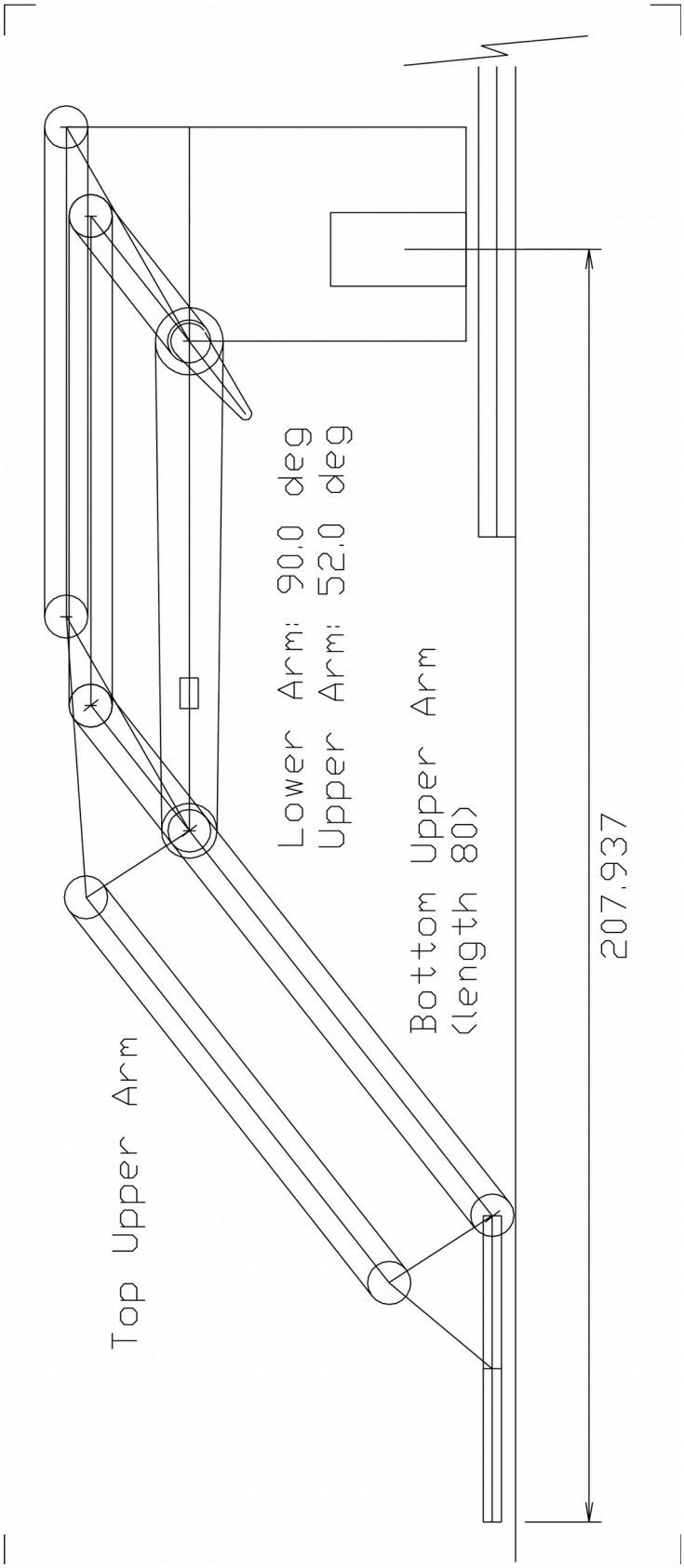


Figure 10 – Arms Positioned for Maximum Reach on Mounting Surface

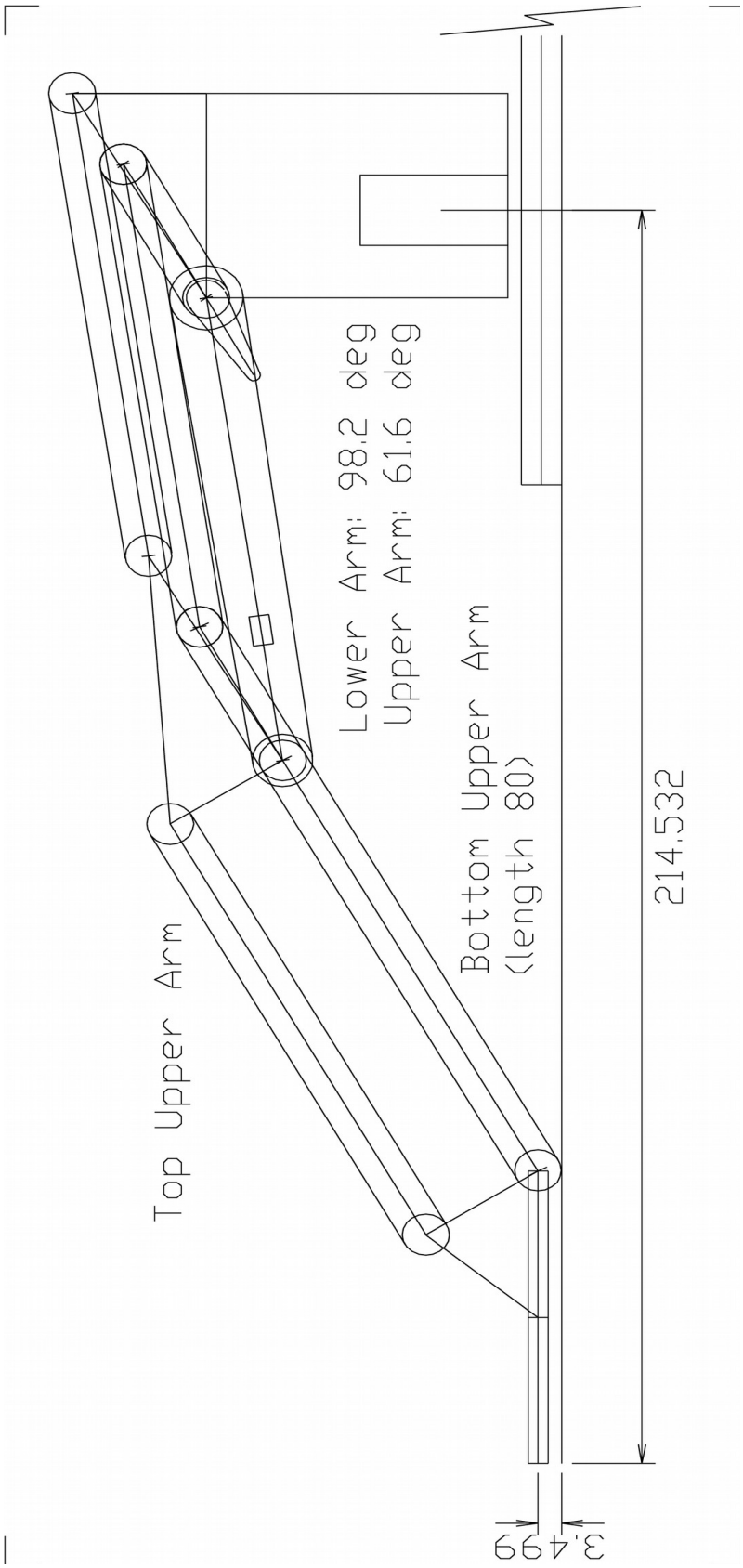


Figure 11 – Arms Positioned for Minimum Reach on Mounting Surface

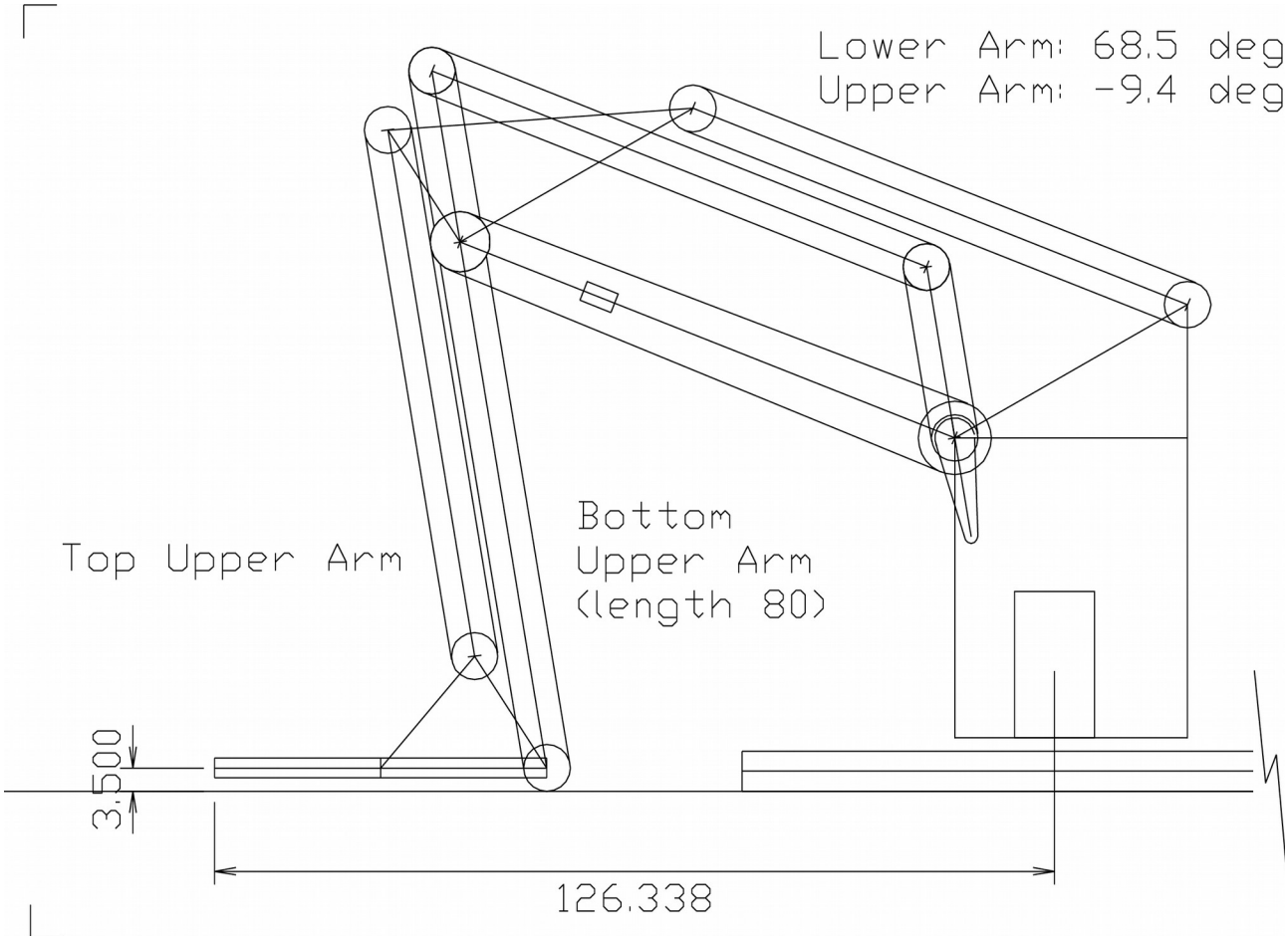


Figure 12 – Simple Kinematic Model

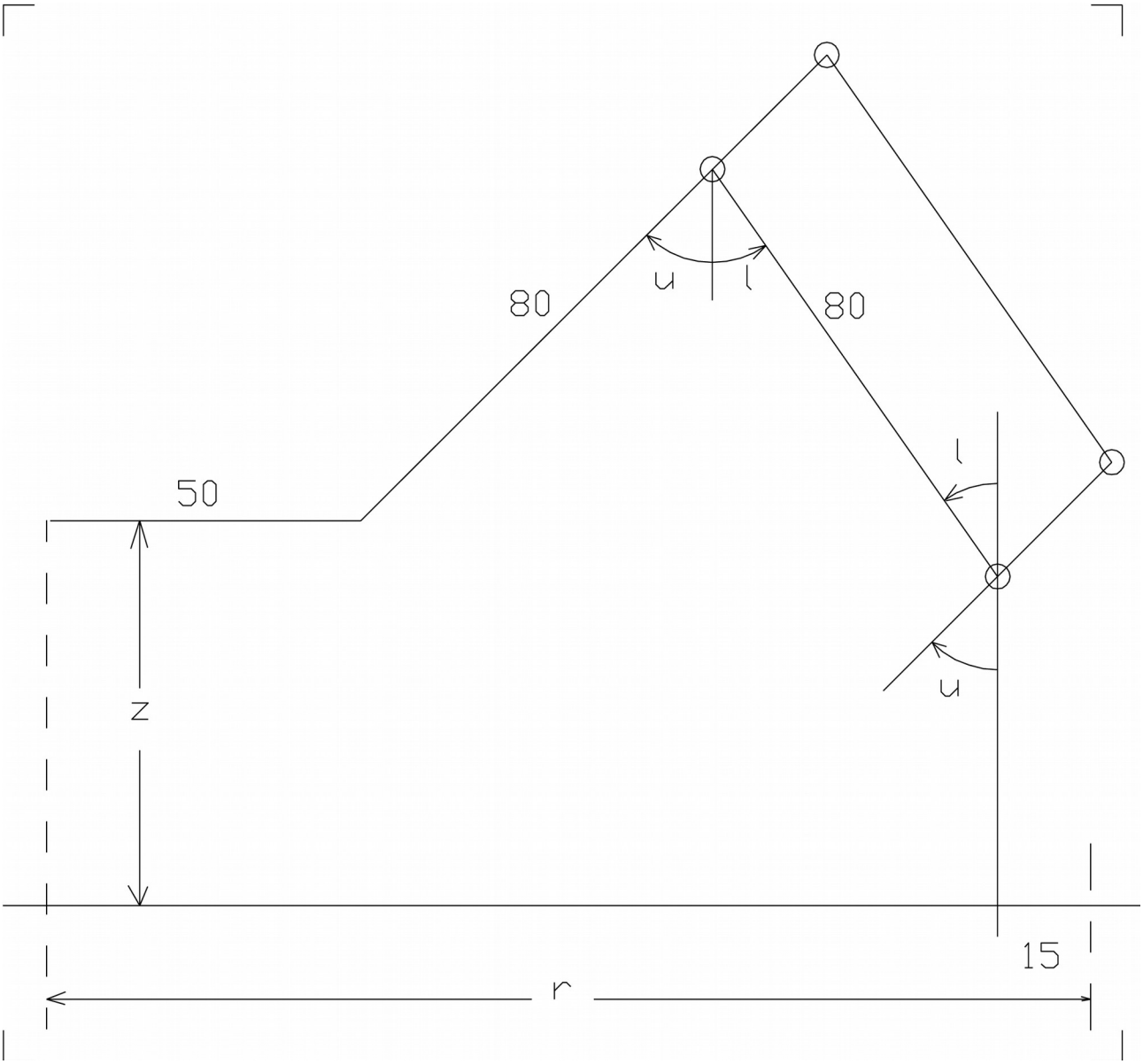


Figure 13 – Positioning Template (not to scale)

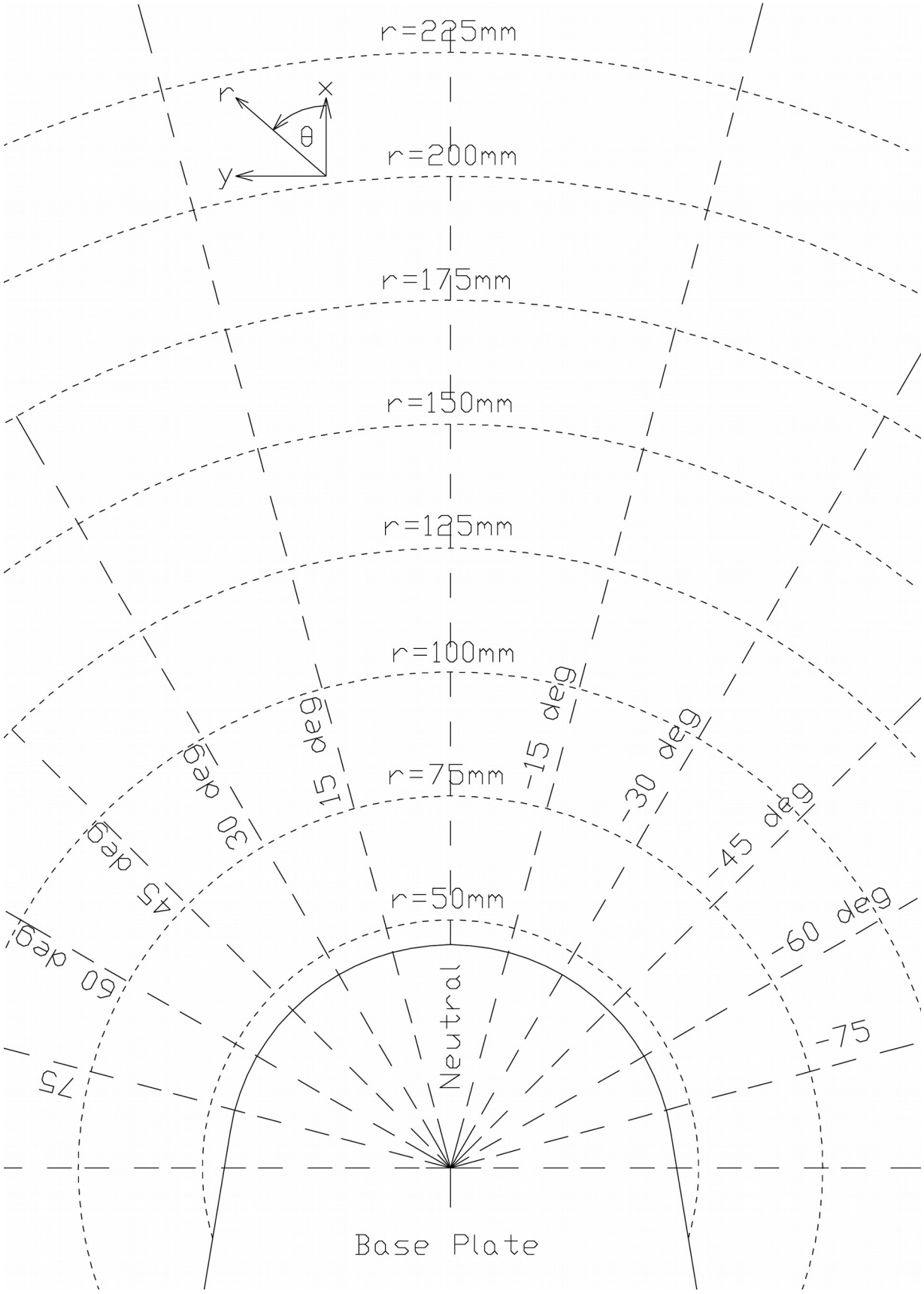
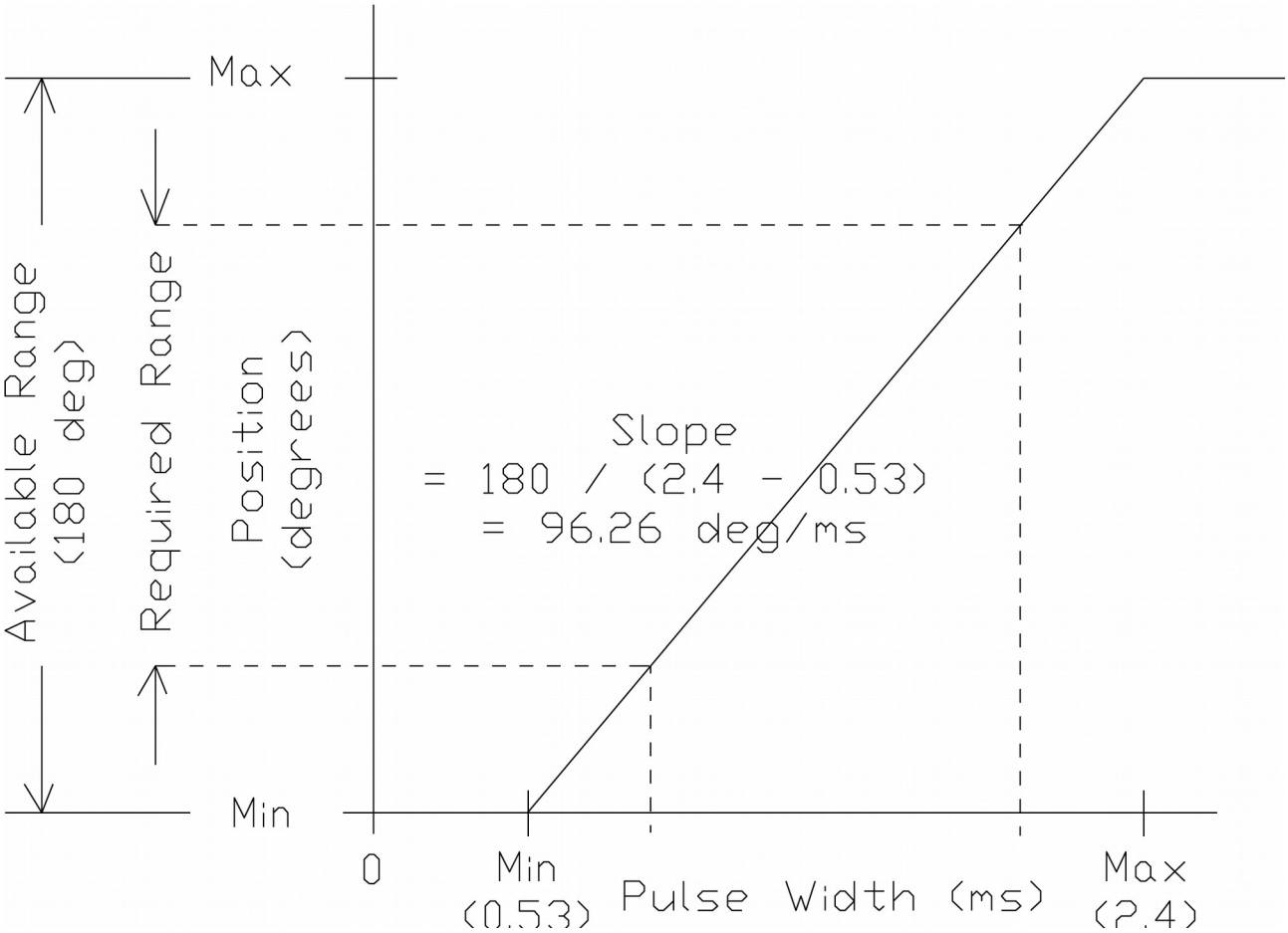


Figure 14 – Setup Graph (Conceptual)



Appendix 1: Sample Code

Commented Code

	Notes: 1. This example deals with MeArm positioning only. It does not acquire joystick data or exercise the tricolour LED. 2. Lines in some cells opposite have wrapped. If running this code, make sure they are not wrapped – use the “cut and paste” format in the next section. Remember: White space is significant in Python – a row beginning with “n” spaces is contained within the lexical scope of previous a row containing “n-1” or less spaces.
import os	Get access to operating system (os) library functions.
import math	Get access to math library functions.
import pigpio	Get access to pigpio functions.
pi = pigpio.pi()	Create an instance of the pigpio class – methods of this object will be used to exercise the GPIO pins.
class Servo:	Start the definition of the Servo class. Note: In Python and many other languages, class names are capitalised by convention.

def __init__(self, config):
  self.pin = config['pin']
  self.min = config['min']
  self.max = config['max']
  self.minAngle = config['minAngle']
  self.maxAngle = config['maxAngle']
	This is the “constructor” method for Servo. It creates and initialises instance-specific variables from a “tuple” called config, supplied by the caller.
	
	Note: Python requires each method definition to include a reference to the instance, which is called “self” by convention.

def moveTo(self, angle): self.moveToAngle(angle)	This is the moveTo method for Servo. It sets the servo angle to a specific value. It calls another method called moveToAngle. See below.

def moveBy(self, angle):
  newAngle = self.currentAngle + angle
  self.moveToAngle(newAngle)
	This is the moveBy method for Servo. It changes the servo angle by a specified amount.

def moveToCentre(self):
  centre = self.minAngle + (self.maxAngle - self.minAngle)/2
  self.moveToAngle(centre)
	This is the moveToCentre method for Servo. It sets the servo halfway between minAngle and maxAngle (nominally the neutral position).

def moveToAngle(self, angle):
  if angle > self.maxAngle:
  angle = self.maxAngle
  if angle < self.minAngle:
  angle = self.minAngle
  self.currentAngle = angle
  self.updateServo()
	This is the moveToAngle method for Servo. If the required angle is outside the range for the instance, it is reset to the minAngle or maxAngle as appropriate, and the value is remembered as the currentAngle for later use. Then the servo position is updated using another method called updateServo.

def updateServo(self):
  pulseWidth = math.floor(self.min + ((float(self.currentAngle - self.minAngle) / float(self.maxAngle - self.minAngle)) * (self.max - self.min)));
  pi.set_servo_pulsewidth(self.pin, pulseWidth)
	This is the updateServo method for Servo. It calculates the required control signal pulse width, then asks the pigpio library to set that up on the relevant GPIO pin.

class MeArm:	Start the definition of the MeArm class.

def __init__(self):
  self.lower = Servo({'pin': 17, 'min': 580, 'max': 2110, 'minAngle': -48.8, 'maxAngle': 98.2});
  self.upper = Servo({'pin': 22, 'min': 620, 'max': 2030, 'minAngle': -9.4, 'maxAngle': 126.5});
  self.base = Servo({'pin': 4, 'min': 530, 'max': 2400, 'minAngle': -90, 'maxAngle': 90});
  self.grip = Servo({'pin': 10, 'min': 530, 'max': 1470, 'minAngle': 0, 'maxAngle': 90});
	This is the “constructor” method for MeArm. It creates four Servo instances called lower, upper, base and grip with appropriate setups. In this example, the setups are hard-coded to specific GPIO pins, meaning that it is inappropriate to create more than one instance of the class.

def moveToPosition (self, lower, upper, base, grip):
  self.lower.moveTo(lower)
  self.upper.moveTo(upper)
  self.base.moveTo(base)
  self.grip.moveTo(grip)
	This is the moveToPosition method for MeArm. It sends the required angle to each of the MeArm’s servos.

myMeArm = MeArm()	Here we create an instance of the MeArm class called myMeArm.
while True:	Here we kick off an infinite loop ...

var = input ("Press <Enter> to move to the Figure 4 (0, 126.5, 0, 0) position") myMeArm.moveToPosition(0, 126.5, 0, 0) var = input ("Press <Enter> to move to the Figure 5 (0, 19.5, 0, 90) position") myMeArm.moveToPosition(0, 19.5, 0, 90) var = input ("Press <Enter> to move to the Figure 7 (-48.8, 126.5, 0, 0) position") myMeArm.moveToPosition(-48.8, 126.5, 0, 0) var = input ("Press <Enter> to move to the Figure 9 (90, 52, 0, 90) position") myMeArm.moveToPosition(90, 52, 0, 90) var = input ("Press <Enter> to move to the Figure 10 (98.2, 61.6, 0, 0) position") myMeArm.moveToPosition(98.2, 61.6, 0, 0) var = input ("Press <Enter> to move to the Figure 11 (68.5, -9.4, 0, 90) position") myMeArm.moveToPosition(68.5, -9.4, 0, 90) print ("Maybe that worked for you ... maybe")	… which repeatedly cycles through the “cardinal” positions.

Code (for cutting and pasting)

```
import os
import math
import pigpio

pi = pigpio.pi()

class Servo:
  def __init__(self, config):
    self.pin = config['pin']
    self.min = config['min']
    self.max = config['max']
    self.minAngle = config['minAngle']
    self.maxAngle = config['maxAngle']

  def moveTo(self, angle):
    self.moveToAngle(angle)

  def moveBy(self, angle):
    newAngle = self.currentAngle + angle
    self.moveToAngle(newAngle)

  def moveToCentre(self):
    centre = self.minAngle + (self.maxAngle - self.minAngle)/2
    self.moveToAngle(centre)

  def moveToAngle(self, angle):
    if angle > self.maxAngle:
      angle = self.maxAngle
    if angle < self.minAngle:
      angle = self.minAngle
    self.currentAngle = angle
    self.updateServo()

  def updateServo(self):
    pulseWidth = math.floor(self.min + ((float(self.currentAngle - self.minAngle) /
float(self.maxAngle - self.minAngle)) * (self.max - self.min)));
    pi.set_servo_pulsewidth(self.pin, pulseWidth)

class MeArm:
  def __init__(self):
    self.lower = Servo({'pin': 17, 'min': 580, 'max': 2110, 'minAngle': -48.8, 'maxAngle': 98.2});
    self.upper = Servo({'pin': 22, 'min': 620, 'max': 2030, 'minAngle': -9.4, 'maxAngle': 126.5});
    self.base =  Servo({'pin': 4, 'min': 530, 'max': 2400, 'minAngle': -90, 'maxAngle': 90});
    self.grip =  Servo({'pin': 10, 'min': 530, 'max': 1470, 'minAngle': 0, 'maxAngle': 90});

  def moveToPosition (self, lower, upper, base, grip):
    self.lower.moveTo(lower)
```

```
    self.upper.moveTo(upper)
    self.base.moveTo(base)
    self.grip.moveTo(grip)

myMeArm = MeArm()

while True:
  var = input ("Press <Enter> to move to the Figure 4 (0, 126.5, 0, 0) position")
  myMeArm.moveToPosition(0, 126.5, 0, 0)
  var = input ("Press <Enter> to move to the Figure 5 (0, 19.5, 0, 90) position")
  myMeArm.moveToPosition(0, 19.5, 0, 90)
  var = input ("Press <Enter> to move to the Figure 7 (-48.8, 126.5, 0, 0) position")
  myMeArm.moveToPosition(-48.8, 126.5, 0, 0)
  var = input ("Press <Enter> to move to the Figure 9 (90, 52, 0, 90) position")
  myMeArm.moveToPosition(90, 52, 0, 90)
  var = input ("Press <Enter> to move to the Figure 10 (98.2, 61.6, 0, 0) position")
  myMeArm.moveToPosition(98.2, 61.6, 0, 0)
  var = input ("Press <Enter> to move to the Figure 11 (68.5, -9.4, 0, 90) position")
  myMeArm.moveToPosition(68.5, -9.4, 0, 90)
  print ("Maybe that worked for you ... maybe")
```

How to Run the Code

Note: The following instructions assume your Pi is running stock Raspbian (not the Mime Industries distribution). The latest version of Raspbian includes a Python IDE (Integrated Development Environment) called Thonny, which is used in this example.

To run the above example:

Copy the code into a text file called mearm-demo.py.

Before launching the Python code, you must start the pigpio "daemon" (pronounced "demon") - a little stand-alone server which implements the low level control commands, like this:

$ sudo pigpiod

Note: sudo (superuser do) gives pigpiod the privileges it needs to exercise the GPIO pins.

Then go for broke, like this (from the directory containing the code):

$ python3 mearm-demo.py

You will need to hit the <Enter> key to move the arm between “cardinal” positions.

If you prefer, you can single-step through the code in Thonny, the Python IDE, like this:

<In the GUI file manager, right-click on mearm-demo.py and select "Thonny Python IDE">
<Thonny will open with the code on display>
<Click on the "Debug current script" icon to start>

<Then repeatedly click on the "Step over" icon to step through the code. Note that you will still need to hit the <Enter> key in the console window in order to progress>

You can modify the code if you want to play, using the existing lines as a hint about how to proceed.

When you are finished, kill the pigpio daemon. Firstly identify the daemon process, like this:

$ ps ax | grep pigpio

You will see something like:

```
 2902    ?        Slsl       0:37      pigpiod
 2957    pts/0    S+          0:00     grep --color=auto pigpio
```

2902 (or whatever the process ID turns out to be on your machine) is the number you are looking for. Then:

$ sudo kill 2902 ← substitute your number