Robotics Final Design Project:

The Programmable Safety Arm

Kanwarpal Brar

Abstract

The robotics final design project is an exploration of the world of robotics in an open, student-dependent, self-managed, project creation format. It revolves around the user of the sensors and actuators commonly used in the robotics implementations of today's industry. The Programmable Safety Arm is a robotics implementation designed around user experience, adaptability, and safety. The following is the design process for this implementation, and follows all aspects of it from software, hardware, circuity, down to schematics. It also includes the testing and investigation of the requirements of the robot, corrections made, as well as further possible improvements. The report ends off with a summarization of skills learned, and the benefits of such.

Introduction

The Programmable Safety Arm, or PSA, is an intelligent robotics system for use in a mixed variety of applications, ranging from industrial to hobby. It can, kinematically, be classified as a stationary robot. Running on a partially closed-loop system, the PSA's use is dictated by the end effector attached to it. However, the most important focus of the PSA is safety. The Programmable Safety Arm is designed to be used in assembly or personal assistance scenarios while reducing the risk of bodily harm to biological organisms within it's working area. As such, the PSA is best-suited for jobs in which it will be working in close proximity to humans. This encompasses scenarios such as on a robot-human mixed assembly line, or even in a hobby situation where the robot acts as an assistant to a primary worker. Unlike many robotics implementations of a similar sort, the Programmable Safety Arm is designed not to replace the human worker, but to aid them in a non-destructive--safe--manner. As such, it is not treated as its own standalone device, but as a specific type of robotic implementation: a human-assistance robot.

The basic concept of the PSA is similar to that of an angular architecture robotic arm. The prototype version runs off of three miniature servo motors powered by an Arduino, providing it movement within three degrees of freedom (excluding an end effector). However, Perhaps the most important distinguishing feature of the PSA to other angular architecture robotic implementations, is that of its focus on safety. The PSA, as opposed to modern industrial robotics which are built of metals and/or plastics, is built of cardboard; a material that is--relatively--soft in comparison to traditional choices. To even further enhance the safety of the PSA, incorporated into its design is a PIR sensor, which enhances the robot's ability to work

with humans. The sensor plays an important role in preventing indirect injuries. When the PSA's human assistant comes within a certain proximity of the robot, the sensor, detecting their movement, halts the robot's operations to drastically reduce the possibility for human injury. The Programmable Safety Arm also allows for, as the name indicates, programmability; allowing users to set arm positions and then save them to memory for later use. This provides great functionality for repetitive procedures and enhances the PSA's ability to work as a human-assistance robot by allowing it to handle such gritty procedures while its human assistant works on the more delicate operations. Also added to enhance the design of the PSA are rotary potentiometers. These potentiometers not only allow manual control of the robot, for any case where such is required, they also make it easier to position the robot in the specific manner that the user aims to save to memory for later use. To enhance user experience, the robot also makes use of various LEDs to indicate robot status; allowing the PSA to communicate with the user. As a whole, the Programmable Safety Arm is a human-assistance robot with an especial focus on improving human interactions with robotics implementations in a work or home scenario through a design that focuses primarily on enhancing the user experience and customizability over pre-programmed autonomous functionality.

Key Features:

- 3 Degrees of Freedom for flexible movement.
- Programmability; allowing users to save a variety of positions and let the robot return to them at any time.
- Manual and Precision control to allow for setup of positions to save, or manual operation of robot.

- LED feedback used to indicate robot status to user.
- Safety Motion Sensor (PIR Sensor) to prevent human injury by stopping robot functions when movement is detected.
- Soft material design to enhance user experience and further decrease risk of injury.

Research and Requirements

The Programmable Safety Arm is a robotics implementation designed--in its entirety--to appeal to business and social issues prevalent in robotics-utilizing industries today. Primarily, it focuses on the belief of workers that robots will take away their jobs, as well the risks, mainly the safety hazards, brought about through business' attempts to rectify such an issue by creating work environments where humans and robots collaborate. A large majority of industrial robotics implementations are not free thinking; they are unable to react to unpredictable situations. This minor flaw, or rather, a missing feature, becomes a greater issue when these robotics implementations are made to work alongside humans. A human worker is unpredictable; a robot is unable to know when they might enter its work space, putting the human at risk of injury, and, potentially, the robot at risk of being damaged. In either outcome, there is a loss for the company, and possibly even social outcry for workers' rights. This creates a circularity to the problem: people are afraid that robots will take their jobs, companies, in an attempt to save money and please the people, create a robot and human work space, and this then leads to injury, which once again leads to issues regarding robots in the workplace. Broken down, the key issues (along with sub-problems) that the PSA deals with are:

- The social issue of robots replacing humans in the workplace

    - Social Issues that arise from solutions to primary issue: Some solutions that are used in industrial robotics to solve the issue of robots replacing humans can-in fact raise new social issues. For example, if a worker is injured because a business creates a robot-human mixed workplace, then there is the possibility of social outcry against the business for placing robots in close vicinity to humans.

- The business-oriented issue of workplace injuries due to robots in close proximity to humans (which is a popular solution to the social issue)
    - Price: Price plays a big factor in both the social and business issue. Only the business can solve the social issue, and businesses are for-profit. Therefore, any solutions to the social and business issue must be affordable; equal to a business's current profits, or able to increase them
    - Features: A very important factor in influencing a business to buy a robotics implementation to solve the business and social issue is to make sure that the new implementation is able to hold up to the old on. No business would want to replace a robot with 3 degrees of freedom with one that has any less, or is weaker, less flexible, or overall with inferior features.

There are, however, those that attempt to break this vicious cycle through a variety of methods. As mentioned before, a majority or industrial robotics implementation are not able to react effectively to human workers. The implementations that are able to do so are usually quite expensive, making it harder for business to implement them due to costs. There is also the option of separating the machines' and humans' workplaces altogether, however this defeats the purpose of making humans and machines work together and could potentially worsen the

efficiency of the business. There are also implementations that exist which can be integrated with existing industrial robotics, allowing them to better work with humans. However, these types of products often have functional and compatibility constraints, such as required space or only working on specific models of robot. Below are listed prominent types of solutions to the identified problems (mainly solutions to prevent human injury due to robots), along with key flaws in their design:

1. SICK: Safe Sequence Monitoring, using SICK Relays

The SICK solution attempts to make it easier for humans to work in the vicinity of robots by using their SICK relay system to set up a connected mesh sensor system, that detects human movement within it. When a human is detected within the mesh, depending on how close they are to the robot, it will either slow down or stop entirely, and will not resume until the human is confirmed to be out of the working zone of the robot. While, at first, this seems like a good solution; and it is in fact very effective at preventing injury, the design of the SICK system quickly falls apart when considering the various factors that play into its implementation into a workspace. The key issues with the SICK Safety System are compatibility and cost. While the system may work, it is up to the business to consider the buy, and no business will invest into something that does not work with their current robots. The SICK system has limited compatibility and required a fair setup process in order to integrate it with a robot's software. This means that, for companies with incompatible robots, the SICK is not a valid option. Furthermore, the SICK system depends on SICK relay's, which, depending on the workspace size, could require a mesh of more than 3. This means that the cost of this implementation rises quickly, especially if it must be implemented for multiple robots. Take this ballpark estimate for

example. A SICK relay sells for about $130 USD on ebay, the actual price requires a quote from the company. At about 3 relays per robot, and with 3 robots, this comes out to a price of $1170 USD, not including taxes, shipping, or setup costs. And this is just for three robots, with only three relays. As can be seen, quickly rising costs mixed with a lack of compatibility, result in the SICK Relay system becoming a lackluster solution to the safety problem, one that is--likely--not affordable, or worthwhile, for any but the most affluent businesses.

Sources provided in references: *Refer to Safety relays, and SICK Safety Relays*

2. Lidar Robotics:

Lidar is a method of surveying an area through the use of laser which illuminate the target. The target then reflects these pulses of light back to the sensor, which are measured. The differences in laser return times, and the wavelengths of the returned lasers, allow the sensor to create a digital 3D representation of the target. Lidar stands for Light Detection and Ranging. In the robotics industry today, some companies have begun implementing Lidar sensors into their products, allowing their robots to become smarter. Using 3D model data, the robot is maneuver around its surroundings, and even recognize persons, allowing the implementation to autonomous make decisions through this information, and actively prevent hazards to humans in its environment. While the Lidar is able to provide large amounts of information to a robot, allowing it to be safer, it suffers some problems of its own, which prevent it from becoming a viable solution to the safety problem, and thus the social issue of robots in the workplace over humans. Mainly, Lidar implementations suffer from issues regarding cost, as well as implementation. Good Lidar devices, due to their complexity, are extremely expensive, making it very difficult to buy the sensor alone, much less buy a robot which already has one implemented.

For reference, a top-of-the-line Lidar sensor from Velodyne costs $75,000 USD, with cheaper models running as low as $6,000 USD. This astronomical cost means that it is very difficult for a business to consider the Lidar as a viable solution; it is sure to leave a large hole their wallet. Past the atrocious cost of the lidar, implementation raises another issue. A Lidar device only provides information, it is up to the implementation to choose what it wants to do with this information. This means that a robot cannot just be hooked up to a Lidar sensor, it needs a dedicated control system that distinguishes the digital 3D data. The Lidar alone is not enough to do anything, the additional required systems, as well as the cost associated with them, makes Lidar devices difficult to implement into a robotics system. Combined with cost, Lidar devices lose their value as a practical solution to the social and safety issue, especially when factoring in the effort a business is willing to go to, to rectify these issues.

Sources provided in references: Refer to *What is Lidar…* and *Velodyne Lidar*

The Programmable safety arm is designed with the shortcoming of its alternatives in mind, and is built from the viewpoint of the business, who is the primary decision maker in whether or not to enhance their work by using humans and robots together. Primarily, it is understood that the business, while there might be social outcry for such, is not inclined to help solve the social issue of robots replacing humans, especially if it's not worth their time. This means that any products used to solve this problem are required to be affordable, efficient, and easy to set up; to coerce business into using the product to solve this social issue. Moreover, the Programmable Safety Arm aims to enhance the business's workflow alongside its previous task, especially by severely reducing the chance of workplace injury due to robots; reducing costs for businesses in the long run. The PSA aims to solve the identified issues in a variety of manners.

Through a soft-shell design as well as an infrared motion sensor to detect human workers within a certain vicinity, the Programmable Safety Arm is able to severely reduce the chance of injury to workers in close proximity of the robot; solving the social issue and allowing humans to better work alongside robots, without a risk of injury and the issues that arise for a business due to it. Through its various efficiency-focused features such as position memory, precision manual control, 3 degrees of freedom, and it's easy to understand LED-based information system, the PSA provides an excellent alternative to current robotics implementations with little to no sacrifice for the business to work efficiency, as well with the potential of usability over pre-existing implementations. This also helps in coercing businesses to solve the social issue mentioned before.

The specific design of the Programmable Safety Arm relies on a variety of factors in its surroundings to remain a valid solution to the identified issues. On the environment end, the PSA is designed expressly for human-robot mixed workplaces, especially those where the two are in close proximity. The robot does not prove to be a good solution for any uses where it is used solely with other robots, as other implementations offer this specialization. Although it is designed primarily for the industrial sector, the features and design of the robot allow it to easily be used in environments outside of this, such as medical (where extreme precision [such as for surgeries] is not required), general workplaces (for instance, a mechanic's assistant), as well as hobby/enthusiast situations. As long as there are humans who could use general robot assistance, it is the correct working environment for the Programmable Safety Arm. In terms of economic sustainability, the PSA, due to its reliance on humans, depends on a strong economy where people are being hired more than robots, as well as one where businesses are able to spend

money to solve social issues (such as those identified), and are willing to pay for products such as the PSA. Finally, in terms of social sustainability, the core of the Programmable Safety Arm is to solve the social issue presented through the human resistance to the rise of robots in the workplace. As such, this issue needs to continue to be important to society for the PSA to be considered an effective solution; if there is no problem, then no-one will buy the solution. The Programmable Safety Arm also depends on a positive societal opinion on robots working alongside humans, as this is what the PSA aims to do; it acknowledges the efficiency and superiority of robots for certain types of jobs, but also understands the needs of the person to make a living. As a whole, the general requirements and constraints put in place by the issues that the PSA has to solve are as follows.

General Functional Requirements:

- 3 Degrees of freedom for enhanced efficiency in movement and maneuverability
- Programmability to allow for ease of use, repeatability, as well efficiency for repetitive tasks
- Motion Sensor technology to detect human workers and enhance safety
- Soft-Shell design to enhance user experience and reduce risk of injury
- LED Feedback
- Precision Manual control

Must Haves:

- 3 degrees of freedom: Required to make robot a worthwhile investment for businesses by not hindering workflow. Most industrial robotics have at least three joints for three degrees of freedom, some with extra maneuvering joints as well.

    - Solves the business problem of workplace efficiency required for robotics implementations that aim to help businesses solve the social issue through robot-human mixed workplaces.
- Motion Sensor technology: Is core to the design by enhancing user safety without severely hinder robot performance.
    - Solves the social problem, as well as the business problem by reducing injury risk for humans, and negatives effects of such for the business.
- LED Feedback: Crucial for allowing the user to communicate with the robot. Without visual feedback, errors in the robot cannot be understood, and it therefore cannot be troubleshooted.

Desirables:

- Programmability, while extremely helpful, is not required for the core issues that the PSA aims to solve. It is entirely possible to make a safe, efficient robot without the ability to manually set robot positions.
- Precision Manual control, while extremely helpful, is not required for the core issues that the PSA aims to solve. It is entirely possible to make a safe, efficient robot without manual control.

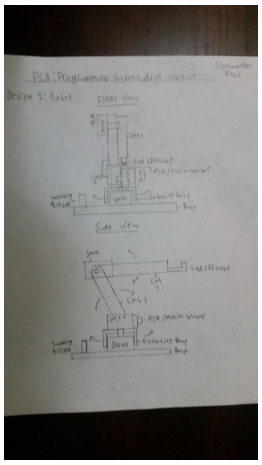
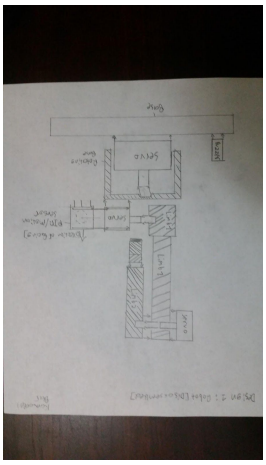
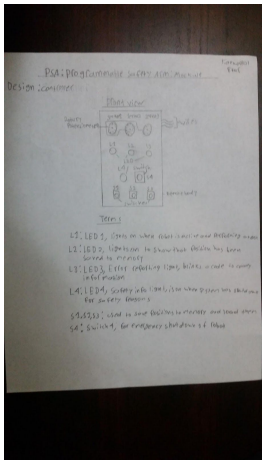
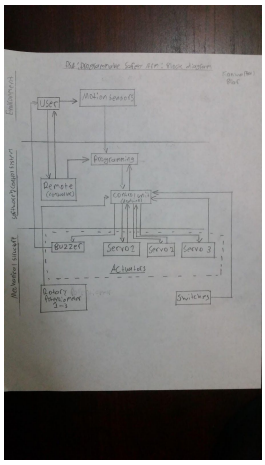
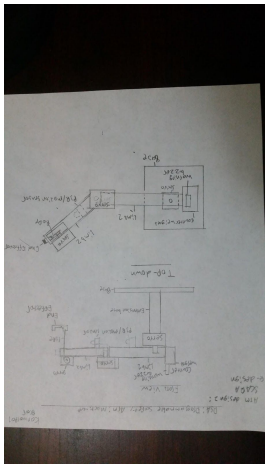
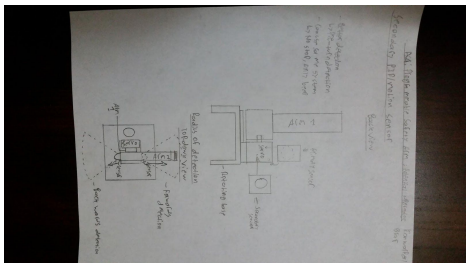
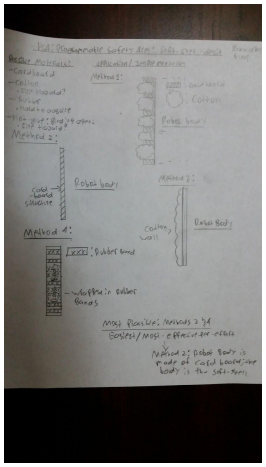
Design Constraints:

- Cannot be too expensive, especially in comparison to superior robots, otherwise the product becomes a bad investment for businesses, resulting in an inability to solve the identified issues.

- Robot will not be very sturdy due to its soft-shell design. Structural stability is present, but weak resistance to shock.
    - This issue is dependent on if a soft-shell design is used, as well as the material (A sturdy material coated in a soft material would not have this problem).
- Robot cannot be too large in size, as this makes it harder to use in limited workspace environments alongside humans.
- Material and cost restrictions only allow for a miniature-sized model to be created for prototype, this is slower, and weaker in terms of speed and payload, but still has all key features.
    - An expanded, production line model would not have these constraints.

Preliminary Designs

1. Structure/Design Mock-up of Arm, Remote, alternative designs:

2. PseudoCode

// Global variables, these need to be referenced throughout the code

#include <Servo.h> // Allows servo's to be used; servo module

// Assign servo variables, this designates that the variables are servos

Servo baseServo;

Servo shoulderServo;

Servo elbowServo;

//----------

// Assign LED's to chosen out pin's

```
const int activityLight = choosenOutPinNumber;
const int posSaveLight = choosenOutPinNumber2;
const int errorLight = choosenOutPinNumber3;
const int safetyStopLight = choosenOutPinNumber4;
//-----------------------------------------------------------------------------------------
// Assign Buzzer pin, and PIR pin
const int buzzerPin = choosenOutPinNumber8;
const int pirPin = choosenPinNumber;
int pirState = LOW;
boolean pirSafety = false;
boolean motionCheck = true;
//-----------------------------------------------------------------------------------------
// Sets up variable for buttons, which are used for commands
const int posSaveButton = choosenOutPinNumber5;
const int posRunButton = choosenOutPinNumber6;
const int safetyButton = choosenOutPinNumber7;


int posSaveButtonPresses = NumberOfPresses;  // Starting value must be 0
boolean posRunButtonIsPressed = false;
boolean safetyButtonVal = false;
//-----------------------------------------------------------------------------------------
// Rotary Potentimeter setup to allow for manual control of servos
```

```
const int basePot = choosenAnalogPin;
const int shoulderPot = choosenAnalogPin2;
const int elbowPot = choosenAnalogPin3;


int basePotVal;
int basePotAngle;
int shoulderPotVal;
int shoulderPotAngle;
int elbowPotVal;
int elbowPotAngle;
//--------------------------------------------------------------------------------------------
// Below are the saved positions for the three servo's, these are cycled when
posRunButtonIsPressed = true
int baseServoSavedPos[] = ListOfSavedPositions;
int shoulderServoSavedPos[] = ListOfSavedPositions2;
int elbowServoSavedPos[] = ListOfSavedPositions3;
// Defualts for all are {1,1,1,1,1} (five positions, for five saves in memory)
//--------------------------------------------------------------------------------------------
void setup() {
//--------------------------------------------------------------------------------------------
  // Attach the servo's to an analog pin on the Arduino
  baseServo.attach(PinOnArduino);
```

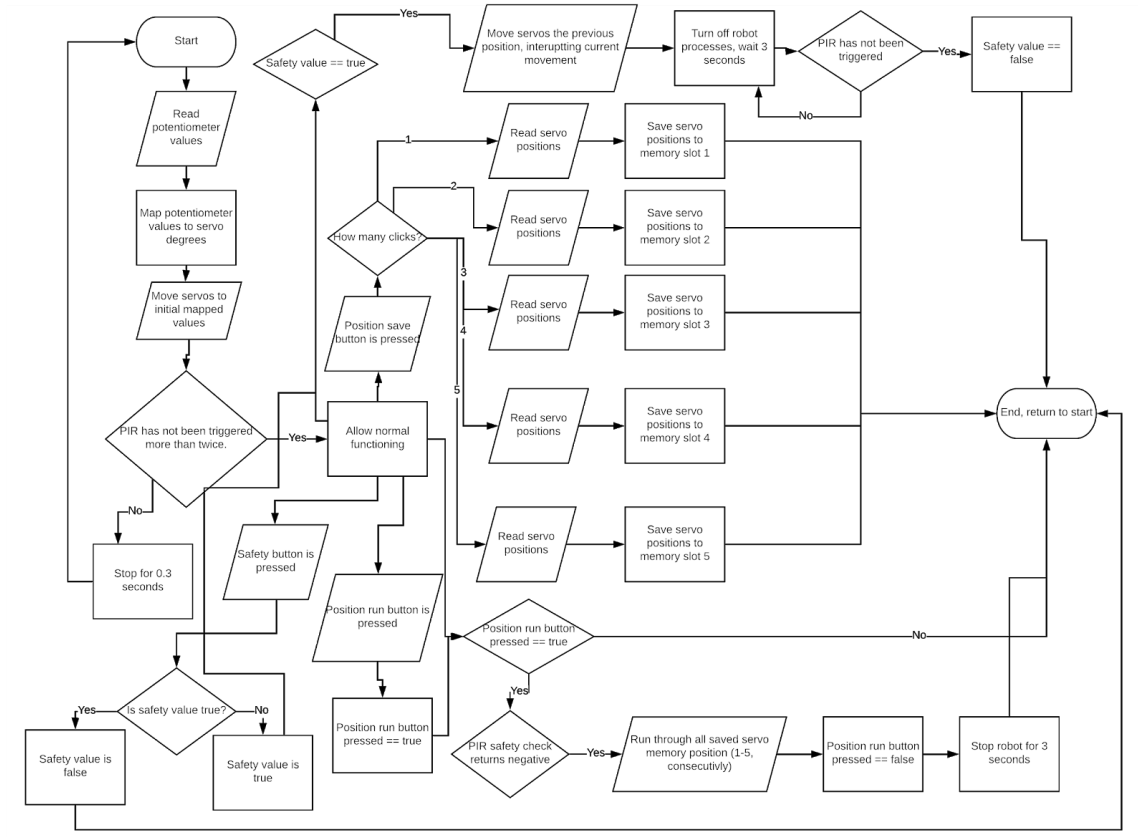
```
  shoulderServo.attach(PinOnArduino2);
  elbowServo.attach(PinOnArduino3);
//-------------------------------------------------------------------------------------------
  // Setup pins for the LED's (outputs) and the pins for the buttons (inputs)
  pinMode(activityLight, OUTPUT);
  pinMode(posSaveLight, OUTPUT);
  pinMode(errorLight, OUTPUT);
  pinMode(safetyStopLight, OUTPUT);
  pinMode(posSaveButton, INPUT);
  pinMode(posRunButton, INPUT);
  pinMode(safetyButton, INPUT);
  // Setup pins for PIR and buzzer
  pinMode(buzzerPin, OUTPUT);  // This is for the warning buzzer
  pinMode(pirPin, INPUT);
//-------------------------------------------------------------------------------------------
  set serial data communication rate to 9600
}
//-------------------------------------------------------------------------------------------
void loop() {
//-------------------------------------------------------------------------------------
  void pirSafetyCheck() {
        int pirTriggers = 0; // This is the number of low to high triggers
```

```
// this function checks the PIR value
if pirState is low {
if pirState is high {
increment pirTriggers
}
}
// This function is supposed to run only after every baseServo run, to ensure proper detection
// If function runs before baseServo, then it might stop the machine
if pirTriggers is more than 2 {
ring the buzzer
set safetyStopVal to true
set pirSafetyVal to true
delay for 5 seconds
}
if pirSafetyVal is true {
check for movement
if no movement, or heat {
safetyStopVal is false
pirSafetyVal is false
}
else (if there is movement) {
```

```
        safetyStopVal stays true, pirSafety stays true
        }
        }
  }
//-------------------------------------------------------------------------------------------
  if safetyStopButton is pressed {
        if safetyStopVal is false, then set to true
        if safetyStopVal is true, then set to false
  }
//-------------------------------------------------------------------------------------------
  read values of all potentiometers and set them to their variable ([blank]PotVal)
  map potentiometer values to values for the servos ([blank]PotAngle)
  move servos to the values dictated by mapped potentiometer values
//-------------------------------------------------------------------------------------------
  while safetyStopVal is false (and the safety button has not been pressed) {
//-------------------------------------------------------------------------------------------
        if posSaveButton is pressed {
        check how many times it is pressed, increment posSaveButtonPresses
        depending on how any times the button is pressed, run through a different case. each case
is a saved set of positions
        assign the iterated servoSavedPosVal to servoPotAngle for each servo
        light up posSaveLight at beginning of every memory save, turn off after
```

```
        }
//-------------------------------------------------------------------------------------------
        if posRunButton is pressed {
        set posRunButtonPressed to true
        }
//-------------------------------------------------------------------------------------------
        if posRunButtonPressed has been pressed {
        for number of saved positions in servo positions {
        run servos to their i position, but, set motionCheck to false before base servo, and true
after
        stall movement for a second
        set posRunButtonPressed to false
        }
        }
//-------------------------------------------------------------------------------------------
  }
//-------------------------------------------------------------------------------------------
  while motionCheck is true {
        use pirSafetyCheck() to check for motion
  }
//-------------------------------------------------------------------------------------------
}
```

3. Flow Chart



a

Final Design Plan

1. Theory of Operation

Initial Information:

- The current drawn for all servos reaches a maximum of 360mA. It varies with load. Voltage generally hangs around 4-4.8 volts.
- Whenever any LED's are powered, they are receiving 5 volts through a 300-ohm resistor from the Arduino control unit.

- Whenever any servos receive motion data, the Arduino is putting out a voltage that is dependent on the degree (in the servos range of motion) that the software has decided to output.
    - All servo signal pins are connected to Pulse Width Modulation pins, which means that voltages are either high or low, but the degree (of servo movement) is determined by the length of the low signal. (ex: 255 degrees would have no lows, whereas 0 would be all low).
- Whenever the PIR sends a signal to the Arduino (through the signal pin) it should be approximately 3.3 volts at most.
    - The sensor itself is powered by 4.5-5 volts
- If information is needed on what pins are connected where, refer to circuit diagram section.
- Potentiometers are powered by 5 volts and provide a base resistance of 10k ohms. The wiper pin output to the Arduino varies depending on degrees of turn (on it).
- Whenever a servo is read from (the Arduino grabs information from the servo) it is actually just referencing the latest servo write command that has been made.

The general concept and operation of the programmable safety arm is not very complicated; further adding to its viability as a simple, inexpensive solution to the issues identified in the preliminary research section. The PSA has its subsystems split into three main categories: The environment system, the software and control system, as well as the mechanical system. These three systems are incorporated together and, as such, communicate with each other frequently to accomplish the PSA's tasks. For more information on the flow of information

in the Programmable Safety Arm, refer to the block diagram in the preliminary designs section. Inside each primary system exists the core subsystems. Within the environment system are the motion sensors. Within the software and control system there is the Arduino Uno used to process tasks, running on its software, as well as the user interface/controller. In the mechanical system there are the three servos, the buzzer, as well as parts of the controller such as the rotary potentiometers and the switches. These the subsystems that will be referred to within the theory of operation.

There are various operations that the Programmable Safety Arm can perform, and for each operation there are different interactions in the subsystems. These operational modes consist of manual control mode, memory control mode, and safety control mode. Manual control mode and memory control mode cannot run together, whereas safety control mode is always running.

Manual Control Mode:

In manual control the subsystems of interest are the rotary potentiometers, the control unit (Arduino), as well as the servos. Manual control mode is always enabled unless memory control is activated (via button on controller), overriding it. In manual control, as the name suggests, the user is able to precisely control the movement of the PSA with the rotary potentiometers. Each of the three potentiometers acts as a control for the position of a respective servo. When, for example, potentiometer one is turned, the change in resistance will be detected by the Arduino control unit via a direct connection. The software then processes this change in resistance, and maps the value, digitally, to a servo value. Then, the control unit sends out this digital servo movement value to the servo affiliated with the inputting potentiometer, turning it

to the indicated position. This is true for any potentiometer in the system. The manual control function runs in a loop, and therefore offers continuous control of the servos through the potentiometers. The order of interaction in the subsystems can be described as so: potentiometer changes resistance, resistance is detected by control unit, software processes control unit input, produces an output, control unit feeds output to servo, servo uses data to turn a specific amount, resulting in robot motion.

Memory Control Mode:

Memory control mode is an autonomous feature of the PSA. The subsystems of interest in memory control mode are the user interface/controller, the Arduino control unit, as well as the three servos. Memory control mode, while executing autonomously, relies on manual control mode as the user must maneuver the robot to a position and then use the controller to save this position to memory for use in memory mode. This function of the PSA starts when the user clicks the memory control switch on the user interface/controller. Depending on the number of clicks, ranging up to five, the current position of all the servos are saved as the nth position, where n is the number of clicks. In the saving process, the servos send their positional data to the control unit. The programming interprets this data and saves it to memory in one of five position data sets, depending on the number of clicks to the switch. Once the memory saving process have been finished, the program, through the Arduino control unit, sends a logic high into the designated memory save successful light on the user interface. This high completes the LEDs circuit and causes it to light up. The program holds this position for a few seconds and then returns its output to low, resulting in the breaking of the flow of electricity. (Electricity from the Arduino first flows through a 300-ohm resistor and then to the LED, this reduces voltage and

makes it safer for the LED). For the execution of memory control mode, the user must click the memory cycle button, beginning the process. The clicking of the switch bridges the electrical connection to the control unit, feeding it a logic high. The program interprets this logic high as the “ok” signal, running the memory control loop. In this loop, the software references all five sets of positional data for every servo, running through each consecutively. This means that the software retrieves the positional data of all three servos for saved position data set one, and then, through the Arduino control unit, sends this data out to the servos, starting first with the base servo, then the shoulder servo, and then the elbow servo (from bottom to top on the robot). The positional data, now arriving at the servos, causes them to rotate to the position dictated by it, reproducing the robot position that was previously saved to memory. The software, control unit, and servos then repeat this process for the remaining four position data sets, until each of the five positions have been reproduced. When the process for executing the saved positions begins, the control unit sends out a logic high to the activity light, which turns on green to indicate that the robot is in motion. The Arduino’s high completes the LEDs circuit, resulting in the flow of electricity. (Electricity from the Arduino first flows through a 300 ohm resistor and then to the LED, this reduces voltage and makes it safer for the LED) The order of interaction in the subsystems can be described as so: switch bridges connection to control unit, activating memory control as dictated by software, software retrieves position data, outputs through control unit into servos, servos move in accordance with position data producing robot movement.

Safety Control Mode:

Safety control mode, as it is always running, encompasses activity over the previous two control modes in accordance with its own function. Safety control mode is the primary feature of

the Programmable Safety Arm, due to the robot's focus on enhancing user safety in the robotics workplace. The subsystems of interest in Safety control mode are the motion sensor, the control unit (as well as the associated programming), the user interface/controller, the three servos, as well as the robot's buzzer. The forefront subsystem in safety control mode is the motion sensor, as that is the autonomous activation of safety mode. The motion sensor is in fact a PIR Motion/Infrared Sensor and is able to detect motion through heat. It operates using a two-piece sensor on a control loop. These two sensor portions are always comparing the heat level in front of them. The PIR activates whenever there is a difference in heat between the two parts of the sensor, indicating a motion, as well as the presence of a biological organism giving off heat in the form of infrared. When this change in heat is detected, the PIR bridges the connection between its power and ground pins, and then outputs power from its signal pin. This positive signal is fed into the Arduino control system and interpreted by the software. In the design of the PSA, every time this high signal is sent, the software increases the value of a counter variable. Once the PIR has been triggered twice in a short period of time, the counter will hit a value of above two, indicating the definite presence of a biological organism through fluctuating motion/infrared heat (this value can be changed to adjust sensitivity). This activates the autonomous safety control. The software, through the Arduino control unit's connections to the servos, grabs the position of every servo in the robot. It then processes this data and, once again through the Arduino control unit's connection to the servos, sends servo movement data out that is identical to the servos' current positions. In essence, the software outputs servo position data, overriding the old data that was being executed, and resulting the robot stopping as the servos are given a command to move 0 units. Once all the servos have been stopped, and the robot is at a

standstill, the control unit sleeps for three seconds (this value can be changed to speed up or slow down the loop [decreased values offer less safety, but higher work efficiency]), then once again checks the values of the PIR. If the PIR's state remains high, and the software continues to see a tripping counter value, the control unit sleeps another three seconds. This process continues until the PIR is deactivated, indicating that the biological organism in the robot's way has moved, at which point the robot resumes normal activity, and the software counter is reset. In the process of detecting heat, every time the PIR is tripped, the software is sent a high. It processes this data and the Arduino sends out a high to the error LED, completing its circuit and turning it. This is a cautionary light that indicates the robot is detecting something. (Electricity from the Arduino first flows through a 300-ohm resistor and then to the LED, this reduces voltage and makes it safer for the LED). If, however, the robot's safety does trip fully, then the Arduino sends out a logic high to the safety LED, completing its circuit and lighting it up. This indicates that the robot has halted because it has detected something. (Electricity from the Arduino first flows through a 300-ohm resistor and then to the LED, this reduces voltage and makes it safer for the LED). The order of interaction between subsystems can be described as so: PIR sensor outputs high signal to Arduino control unit, this signal is processed by the software, when the software has identified activation of PIR more than twice (through the digital counter and fluctuating signals). The software, through the Arduino, sends out servo positional data equal to the servos' position at the time, overriding current servo movement with a movement of zero units; stalling robot motion until the PIR is no longer active.

The software control mode can also be activated through its non-autonomous method using the user interface/controller. If the user presses the safety switch on the controller, it

bridges the connection of electricity providing a logic high to Arduino control unit. This input is processed by the software resulting in it triggered the same servo override as before, and a complete standstill of the robot. This time however, the switch has caused the change of a boolean variable in the software, resulting in the robot staying off even if the switch is let go of, un-bridging the connection and producing a logic low/0. Manual Safety stop is only turned off when the switch has been pressed once again, this time resulting in the flipping back of the boolean variable, and the resumption of normal robot activity. Also occurring after the initial button press, the Arduino sends out a logic high to the safety LED, indicating that the robot has been paused, as well as a logic high to a piezo buzzer, as a warning sound to nearby users. The high signal competes the LED circuit and lights it up, and the buzzer circuit; producing noise. (Electricity from the Arduino first flows through a 300-ohm resistor and then to the LED, this reduces voltage and makes it safer for the LED). The order of interaction between subsystems can be described as so: safety switch is pressed, bridging connection and producing logic high to Arduino. Arduino, through its software, process the logic high and sends out positional data to servos, equal to their current positions. In doing so, the servos are overridden with a movement of 0 units, bringing the robot's motion to a halt.

Overview:

When viewing theory of operation for the Programmable Safety Arm it is important to remember that the three main control modes, and their focus subsystems often overlap to form one large operational pattern. The general rule to remember is that safety control mode can always be activated, both autonomously and manually, while manual mode is the default mode of the robot. Memory mode requires calibration through manual mode but cannot run at the same

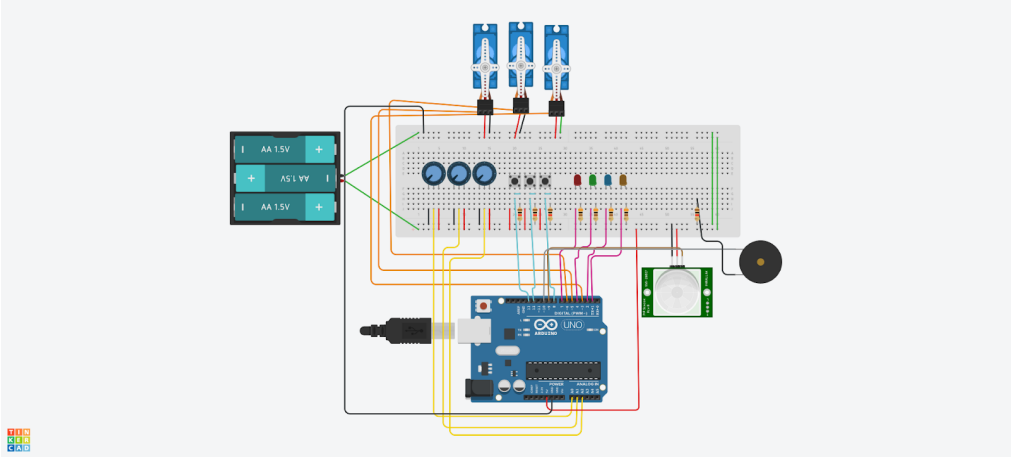
time as manual mode. Any attempt at manual control during execution of memory positions will result in either no effect, or an overriding/interference with servo movement.

2. Mechanical Drawings

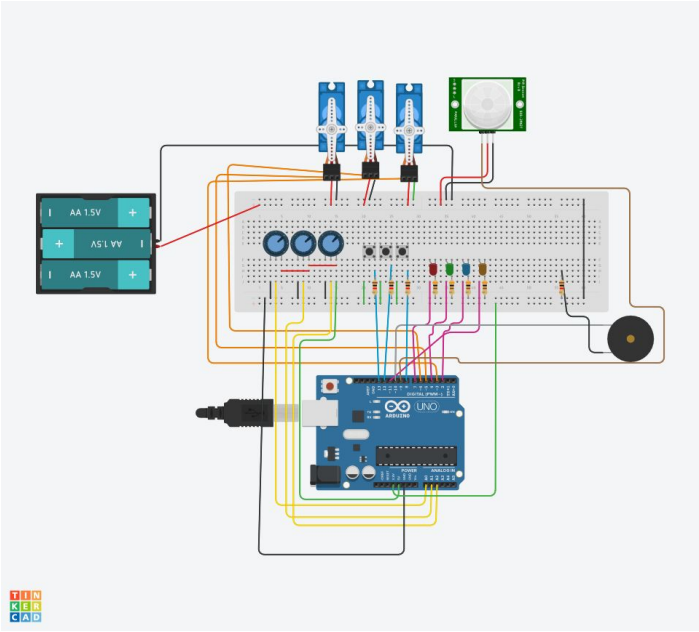
Refer to this link for a pdf of mechanical drawings:

https://drive.google.com/file/d/1f13Cf3hMS9YEwTnYY1lwHyLVi0puWnAk/view?usp=sharing

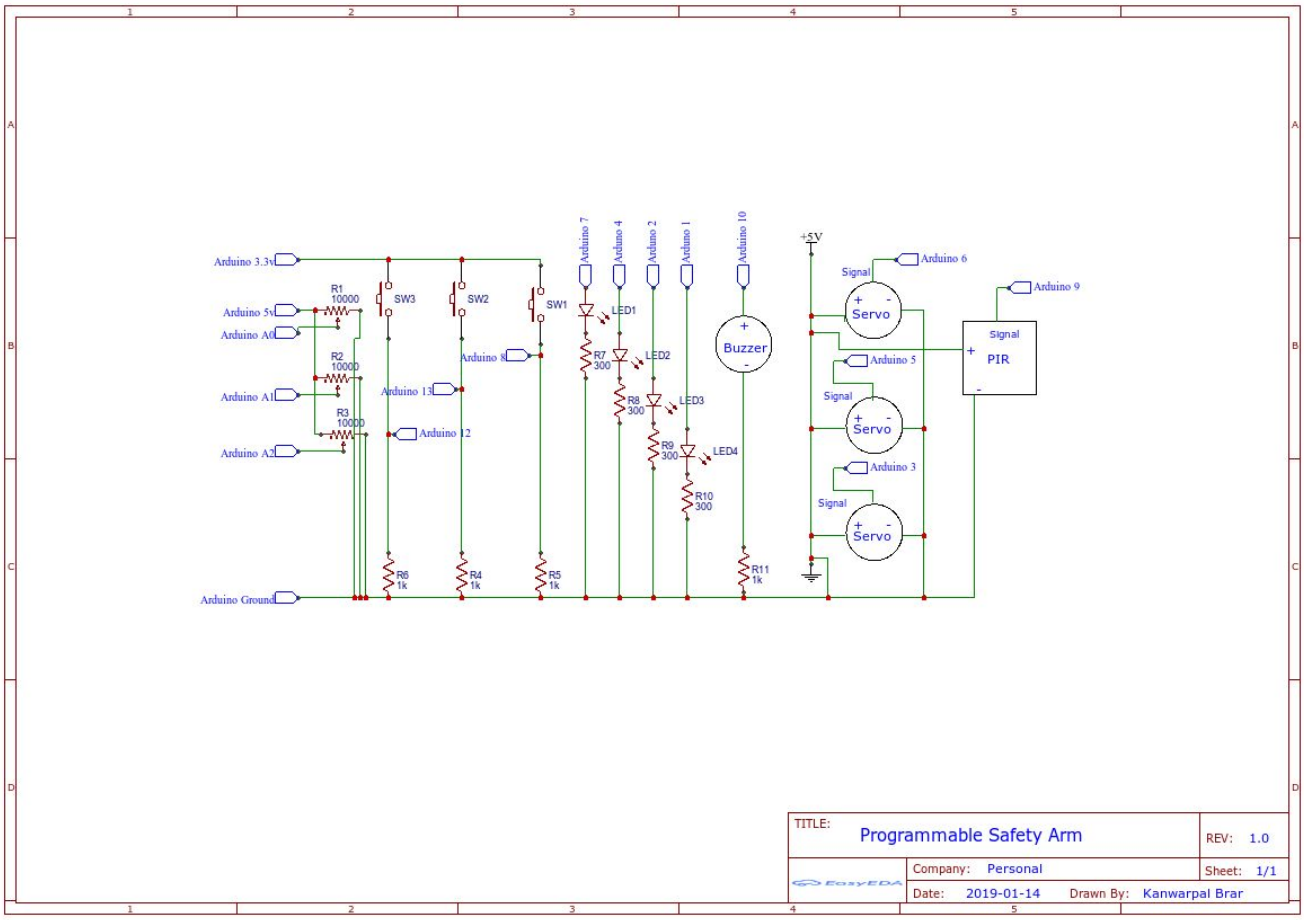
3. Circuit Diagram



Revised Circuit Diagram:



Schematic Diagram:



4. Bill of Materials

Part	Model Number	Description	URL reference	Quantity	Price
10K Ohm Adjustment Single Linear Rotary Taper Potentiometer	a13061400ux0444	A rotary potentiometer can be used to provide resistance in a circuit. Resistance can be manipulated via knob.	https://amzn.to/2SJSs4y	x3	$1.78 (per 1), $8.89 (per 5)
TowerPro SG90 9G micro small servo motor	B01GGANFZY	A miniature servo motor that can be used to provide movement as	https://amzn.to/2RBVtqf	x3	$3.38 (Per 1) $16.89 (Per 5)

		well as position data. Stall Torque (4.8v): 1kg/cm			
Pyroelectric Infrared PIR Motion Sensor	STK0114001941	A PIR sensor for detecting motion through infrared	https://amzn.to/2sj8tmE	x1	$8.13 (Per 1)
Piezo Electronic Tone Buzzer	Akozon2nqsy0t8cd	A piezo buzzer used to produce noise when powered	https://amzn.to/2RfQ2Ow	x1	$9.99 (Per 1)
Coroplast Sign Board, Corrugated Fluted Plastic Sheet	COR-WT-4MM/1248	Corrugated plastic sheets, used as the material for building the structure of the robot	https://amzn.to/2RjUoEr	x1	$25.49 (Per 12”x 48” sheet)
UNO R3 Board	EL-CB-001	Arduino microcontroller, acting as main control unit for robot	https://amzn.to/2TE1QqK	x1	$13.85 (Per 1)
Momentary SPST Push Button Switch	a12081400ux0336	Momentary switch. Bridges connections when pressed	https://amzn.to/2H1d4UD	x4	$1.63 (Per 1) $8.11 (Per 5)

Assorted Resistors	HR0679	Resistors of various resistances. They can be implemented in a circuit to manage voltage and current	https://amzn.to/2H4jjqE	x6 (221-300 ohm) x2 (10k ohm)	$12.98 (Per 1 assorted kit) <$1 for 1
Assorted LED's	HR899	Various colors of standard Light Emitting Diode. They produce light when powered	https://amzn.to/2VFCGJP	x4 (One red, rest assorted)	$12.99 (Per kit of 300, assorted) <$1 for 1
(Potential Addition) 10uf capacitor	B00WS81RBS	Capacitors that store small amounts of charge. Can be used to stabilize servos.	https://amzn.to/2TzVQz1	X1 (10uF)	$11.88 (40 Pack) <$1 for 1

5. Assembly Procedure

Circuit Assembly:

1. Wire up your breadboard, jumping the positive to positive, negative to negative on the end opposite to the power supply input.
    a. Make sure that the power supply is no more than 5 volts, as more could dangerous for the Arduino.
2. Place down your basic components: Three switches, four LED's, three rotary potentiometers
    a. For the easiest wiring, group all of the same type of component together, while leaving adequate space in between

3. After uploading the software to it and disconnecting the data cable, jump the ground and power of your Arduino to the breadboard power.
    a. Make sure that the ground is to the left as possible on the returning negative live so that it is a common ground. This means that, from a schematic point of view, the ground of every component is connected to the Arduino ground.
4. For each rotary potentiometer, connect terminal 1 to ground, and terminal 2 to positive. Then, connect the wiper pin to an analog pin on the Arduino. For example, A0, A1, and A2 are analog pins.
5. For each switch, connect one terminal to positive, and one terminal to ground through a 1k resistor. From just above the resistor, between it and the switch, jump to a standard digital pin on the Arduino. For example, pins 13, 12, and 8 are standard pins.
6. For each LED, connect the anode to negative through a 300-ohm resistor, and jump the positive to a normal pin on the Arduino. For example, 7, 4, 2, and 1 are normal pins (1 is a TX pin but can work like a normal pin).
7. For each servo motor connect the ground to negative, and the power to positive. Then, jump the signal pin to a PWN pin on the Arduino. For example, ~6, ~5, and ~3 are PWN pins.
8. For the buzzer, connect the negative pin to a 1k ohm resistor, and connect the positive pin to a normal pin on the Arduino. If normal pins are used, PWN pins can be used. For example, pin ~10 would work.

9. For the PIR, connect the ground pin to negative, and the power pin to positive. Then connect the signal pin to a normal pin on the Arduino. If there are no normal pins, a PWN pin would work. For example, pin ~9 would work.

Robot Assembly:

1. Set down a piece of corrugated plastic, dimensions 5” x 5” x ½”. This is the base of the arm
2. Place the servo motor down on the centre of the sheet. It can be affixed to the plastic through glue. (Do not glue down until after step 3)
    a. Screws can be used, but this would require creating a hole in the base plastic to allow the servo to fit, and the screw holes to be flush. The rest of the design would have to accommodate for this position shift.
3. Use a 2” x 2” x ½” square of plastic and cut a ⅝” x 1” hole in the centre. Place the base servo in this hole, and then place the square plastic in the centre of the base plastic.
4. Attach a 2 ¼” x 2 ¼” square to the top of base servo
    a. It can be attached using glue
5. Stack 1” x 1” pieces of corrugated plastic until they have a collective height of ¾”. Align this piece with the centre right of the 2 ¼” x 2 ¼” square. Fix it into place
6. Attach the next servo motor sideways and facing inwards (towards the centre of the square) and bind it into place. Make sure that the moving portion of the servo is free to move and is not blocked by anything. Leave enough clearance.

    a. A counterweight can be connected to the opposite end of the servo which will enhance the robot's stability when lifting heavier loads. However, it is not required and, for a prototype miniature, will hinder movement.

7. Create a rectangular prism, 1" x 1 ½" x 1" (W, L, H). Attach this to the rotating end of the servo, form the face of the smallest area side.
    a. This can be attached using glue, or can be screwed in place through the servo's included screw holes
8. Get two 1" x ½" strips of plastic, one with a height of 4", another at 3 ½". Attach these pieces to the left and right side of the rectangular prism (mentioned above).
    a. Make sure that the taller piece on the outer edge, furthest away from the second servo, with the shorter piece closest.
9. Mount a servo motor sideways, with its back-side flush to the taller strip, and its rotary piece facing inwards (away from the taller piece). This should result in the servo sitting on the top of the shorter strip of plastic. Bind it in place.
10. Get two strips of 4" x ½" (thickness) x 1" (height) strips of plastic. Bind these two together. Attach the side (1" side) to the rotary piece of the servo, leaving ½" of space behind the connection. Limb/Arm 2 is now connected
11. Next, we make the end effector. Get a 0.4" x 1" x 1" (height) piece of plastic, a 1" x 1.6" x ½ "(height) piece, as well as a 0.4" x 1" x ½ "(height) piece. These will be referred to as piece 1, 2 and 3, respectively. Affix the ½" face of piece 2 to the bottom portion of the 1" (height) of piece 1. Then, at the end of piece 2, on the side opposite to the previous connection, affix the 1" x 0.4" face to piece 2. The end result should look similar to a

small hook. Now, affix the 1” x 1” face of piece 1 to the bundled end of the joint-piece from the previous step, resulting in an arm with a hook end.

12. Now cut out three pieces of plastic, two 1” x 1” x ½” (height) pieces, and one 1 ½ “x ½ “x 1” (height) piece. These will be referred to as piece A, B, and C, respectively. Attach both piece A and B to the secondary arm, one underneath the connection between the arm and the end effector, as well as one at the very end of the arm (the side closest to the servo). Piece C is mounted to the back of the collective strips of plastic that form arm 1, in between the topmost servo and the rectangular prism that makes up the base of arm one.
    a. These arms provide support. If the arm is sagging, consider adding more, or repositioning these three.
13. Place a buzzer on the base plastic sheet, make sure it is not tall enough to interfere with robot activity.
14. Get a 1” x ½” x 1” (height) piece of plastic. Mount these to the top of the plastic sheet connected to the bottommost servo, just in front of the second servo (facing the front of the robot), affixing it by the 1” x ½” face. Make sure it does not interfere with the spinning of the servo it is next to.
15. Connect your PIR to the 1” x 1” face of the above plastic piece, making sure it faces in the direction of the robot’s facing.

6. Software

<u>Programmable Safety Arm, Version 1:</u>

```
// Global variables, these need to be referenced throughout the code
#include <Servo.h>  // Allows servos to be used; Servo module
```

```
Servo baseServo;
Servo shoulderServo;
Servo elbowServo;
// Assign LED's to choosen out pin's
const int activityLight = 4;
const int posSaveLight = 2;
const int errorLight = 1;
const int safetyStopLight = 7;

// Assign Buzzer pin and PIR ppin
const int buzzerPin = 10;
const int pirPin = 10;
int pirState = LOW;
boolean pirSafetyVal = false;
boolean safetyStopVal = false;
int pirTriggers = 0;  // This is the number of low to high triggers on the PIR

// Sets up variable for buttons, which are used for commands
const int posSaveButton = 13;
const int posRunButton = 12;
const int safetyButton = 8;

int posSaveButtonPresses = 0;
boolean posRunButtonIsPressed = false;
boolean safetyButtonVal = false;

// Rotary potentiometer setup to allow for manual control of servos
```

```
const int basePot = A0;  // Pin for base potentiometer
const int shoulderPot = A1;  // Pin for shoulder potentiometer
const int elbowPot = A2;  // Pin for elbow potentiometer
// Angles and values for each servo-pot combo. Angles for servos, vals for potentiometers
int basePotVal;
int basePotAngle;
int shoulderPotVal;
int shoulderPotAngle;
int elbowPotVal;
int elbowPotAngle;

// Below are the saved positions for the three servo's, these are cycled when posRunButtonIsPressed = true
int baseServoSavedPos[] = {1,1,1,1,1};
int shoulderServoSavedPos[] = {1,1,1,1,1};
int elbowServoSavedPos[] = {1,1,1,1,1};
// Defaults for all are {1,1,1,1,1} (five positions, for five saves in memory)

void setup() {
  // Setup pins for LED's (outputs) and pins for buttons (inputs)
  pinMode(activityLight, OUTPUT);
  pinMode(posSaveLight, OUTPUT);
  pinMode(errorLight, OUTPUT);
  pinMode(safetyStopLight, OUTPUT);
  pinMode(posSaveButton, INPUT);
  pinMode(posRunButton, INPUT);
  pinMode(safetyButton, INPUT);
  // Setup pins for PIR and buzzer
```

```
  pinMode(buzzerPin, OUTPUT);  // This is for the warning buzzer
  pinMode(pirPin, INPUT);  // Sets the PIR input pin as an input

  Serial.begin(9600);
}

// Function used to check if the robot workspace is safe for movement
// three variables passed to function are the servo positions before run
void pirSafetyCheck(int baseInitialPos, int shoulderInitialPos, int elbowInitialPos) {
  // Check if pir has been triggered
  if(pirState = LOW) {
            if(pirState = HIGH) {
            pirTriggers++;
            }
  }
  // This function is supposed to run only after every baseServo run, to ensure proper detection
  // If function runs before baseServo, then it might stop the machine

  //Below interupts servo movement
  if(pirTriggers >= 2) {
            digitalWrite(buzzerPin, HIGH);  // Rings buzzer
            safetyStopVal = true;  // Triggers safety values
            pirSafetyVal = true;  // Triggers safety values
            baseServo.write(baseInitialPos);  // Overwrites servo position
            shoulderServo.write(shoulderInitialPos);  // Overwrites servo position
            elbowServo.write(elbowInitialPos);  // Overwrites servo position
            delay(3000);  //Stops for 3 seconds
```

```
  }
  if(pirSafetyVal == true) {
        if(pirState == LOW) {
        delay(100);
        if(pirState == LOW) {
        safetyStopVal = false;
        pirSafetyVal = false;
        }
        }
        else {
        safetyStopVal = true;
        pirSafetyVal = true;
        }
  }
}

void loop() {
//-------------------------------------------------------------------------------------------
  if(digitalRead(safetyButton) == HIGH) {
        // This function is a toggle for the safety stop button; toggles true or false
        if(safetyStopVal == false) {
        safetyStopVal = true;
        }
        if(safetyStopVal = true) {
        safetyStopVal = false;
        }
  }
```

```
  // Below reads the value of potentiometers and then maps them to degrees for the servos to move to
  basePotVal = analogRead(basePot);
  basePotAngle = map(basePotVal, 0, 1023, 0, 179);
  shoulderPotVal = analogRead(shoulderPot);
  shoulderPotAngle = map(shoulderPotVal, 0, 1023, 0, 179);
  elbowPotVal = analogRead(elbowPot);
  elbowPotAngle = map(elbowPotVal, 0, 1023, 0, 179);
  // Below runs the servos to the potentiometer positions
  baseServo.write(basePotAngle);
  shoulderServo.write(shoulderPotAngle);
  elbowServo.write(elbowPotAngle);
//-----------------------------------------------------------------------------------------
  if(digitalRead(posSaveButton) == HIGH) {
        // Checks if the position save button has been pressed
        posSaveButtonPresses++;
        switch (posSaveButtonPresses) {  // Executes a different case depending on how many times the button is
pressed
        case 1:  // Saves potentiometer positions to servo angles memory 1
        baseServoSavedPos[0] = basePotAngle;
        shoulderServoSavedPos[0] = shoulderPotAngle;
        elbowServoSavedPos[0] = elbowPotAngle;
        break;
        case 2:  // Saves potentiometer positions to servo angles memory 2
        baseServoSavedPos[1] = basePotAngle;
        shoulderServoSavedPos[1] = shoulderPotAngle;
        elbowServoSavedPos[1] = elbowPotAngle;
        case 3:  // Saves potentiometer positions to servo angles memory 3
```

```
            baseServoSavedPos[2] = basePotAngle;
            shoulderServoSavedPos[2] = shoulderPotAngle;
            elbowServoSavedPos[2] = elbowPotAngle;
            case 4:  // Saves potentiometer positions to servo angles memory 4
            baseServoSavedPos[3] = basePotAngle;
            shoulderServoSavedPos[3] = shoulderPotAngle;
            elbowServoSavedPos[3] = elbowPotAngle;
            case 5:  // Saves potentiometer positions to servo angles memory 5
            baseServoSavedPos[4] = basePotAngle;
            shoulderServoSavedPos[4] = shoulderPotAngle;
            elbowServoSavedPos[4] = elbowPotAngle;
            }
}
if(digitalRead(posRunButton) == HIGH) {
            posRunButtonIsPressed = true;
}

if(posRunButtonIsPressed == true) {
            // Runs through memory positions
            // Checks safety of work environment, passes the first memory saved position as the initial position
            pirSafetyCheck(baseServoSavedPos[0], shoulderServoSavedPos[0], elbowServoSavedPos[0]);
            if(safetyStopVal == false) {
            for(int i = 0; i<5; i++) {  // Cycles through each memory position for servo
            // Moves each servo to it's ith memory position
            baseServo.write(baseServoSavedPos[i]);
            shoulderServo.write(shoulderServoSavedPos[i]);
            elbowServo.write(elbowServoSavedPos[i]);
```

```
            delay(1050);  // Stops robot operations for 1.5 secons)
            }
            }
  }
  delay(300);  // Stops robot for 0.3 seconds
}
```

Programmable Safety Arm, Version 2:

```
// Global variables, these need to be referenced throughout the code
#include <Servo.h>  // Allows servos to be used; Servo module
Servo baseServo;
Servo shoulderServo;
Servo elbowServo;

// Assign LED's to choosen out pin's
const int activityLight = 4;
const int posSaveLight = 2;
const int errorLight = 1;
const int safetyStopLight = 7;

// Assign Buzzer pin and PIR ppin
const int buzzerPin = 10;
const int pirPin = 10;
int pirState = LOW;
boolean pirSafetyVal = false;
boolean safetyStopVal = false;
int pirTriggers = 0;  // This is the number of low to high triggers on the PIR
```

```
// Sets up variable for buttons, which are used for commands
const int posSaveButton = 13;
const int posRunButton = 12;
const int safetyButton = 8;

int posSaveButtonPresses = 0;
boolean posRunButtonIsPressed = false;
boolean safetyButtonVal = false;

// Rotary potentiometer setup to allow for manual control of servos
const int basePot = A0;  // Pin for base potentiometer
const int shoulderPot = A1;  // Pin for shoulder potentiometer
const int elbowPot = A2;  // Pin for elbow potentiometer
// Angles and values for each servo-pot combo. Angles for servos, vals for potentiometers
int basePotVal;
int basePotAngle;
int shoulderPotVal;
int shoulderPotAngle;
int elbowPotVal;
int elbowPotAngle;

// Below are the saved positions for the three servo's, these are cycled when posRunButtonIsPressed = true
int baseServoSavedPos[] = {1,1,1,1,1};
int shoulderServoSavedPos[] = {1,1,1,1,1};
int elbowServoSavedPos[] = {1,1,1,1,1};
// Defaults for all are {1,1,1,1,1} (five positions, for five saves in memory)
```

```
void setup() {
  // Setup pins for LED's (outputs) and pins for buttons (inputs)
  pinMode(activityLight, OUTPUT);
  pinMode(posSaveLight, OUTPUT);
  pinMode(errorLight, OUTPUT);
  pinMode(safetyStopLight, OUTPUT);
  pinMode(posSaveButton, INPUT);
  pinMode(posRunButton, INPUT);
  pinMode(safetyButton, INPUT);
  // Setup pins for PIR and buzzer
  pinMode(buzzerPin, OUTPUT);  // This is for the warning buzzer
  pinMode(pirPin, INPUT);  // Sets the PIR input pin as an input
  // Setup servos
  baseServo.attach(3);
  shoulderServo.attach(6);
  elbowServo.attach(5);
  // setup pots
  pinMode(basePot, INPUT);
  pinMode(shoulderPot, INPUT);
  pinMode(elbowPot, INPUT);

  Serial.begin(9600);
}

void motorRun(int sPos1, int i) {
  if(pirState = LOW) {
```

```
            if(pirState = HIGH) {
            pirTriggers++;
            digitalWrite(errorLight, HIGH);
            delay(200);
            digitalWrite(errorLight, LOW);
            }
}
baseServo.write(baseServoSavedPos[i]);
digitalWrite(activityLight, HIGH);
delay(100);
if (pirTriggers <3) {
            shoulderServo.write(shoulderServoSavedPos[i]);
            elbowServo.write(elbowServoSavedPos[i]);
            delay(200);
            digitalWrite(activityLight, LOW);
}
else {
            digitalWrite(activityLight, LOW);
            digitalWrite(errorLight, HIGH);
            digitalWrite(safetyStopLight, HIGH);
            delay(100);
            digitalWrite(errorLight, LOW);
            baseServo.write(sPos1);
            pirSafetyVal = true;
            safetyStopVal = true;
}
```

```
}
// Function used to check if the robot workspace is safe for movement
// three variables passed to function are the servo positions before run
// DEPRICATED FUNCTION
/*
void pirSafetyCheck(int baseInitialPos, int shoulderInitialPos, int elbowInitialPos) {
 // Check if pir has been triggered
 if(pirState = LOW) {
        if(pirState = HIGH) {
        pirTriggers++;
        }
 }
 // This function is supposed to run only after every baseServo run, to ensure proper detection
 // If function runs before baseServo, then it might stop the machine

 //Below interupts servo movement
 if(pirTriggers >= 2) {
        digitalWrite(buzzerPin, HIGH);  // Rings buzzer
        safetyStopVal = true;  // Triggers safety values
        pirSafetyVal = true;  // Triggers safety values
        baseServo.write(baseInitialPos);  // Overwrites servo position
        shoulderServo.write(shoulderInitialPos);  // Overwrites servo position
        elbowServo.write(elbowInitialPos);  // Overwrites servo position
        delay(3000);  //Stops for 3 seconds
 }
 if(pirSafetyVal == true) {
        if(pirState == LOW) {
```

```
            delay(100);
            if(pirState == LOW) {
            safetyStopVal = false;
            pirSafetyVal = false;
            }
            }
            else {
            safetyStopVal = true;
            pirSafetyVal = true;
            }
  }
}
*/
void loop() {
  while (safetyStopVal == true) {  // Loop to check safety
            digitalWrite(safetyStopLight, HIGH);
            if(pirState = LOW) {
            delay(300);
            if(pirState = HIGH) {
            pirTriggers++;
            digitalWrite(errorLight, HIGH);
            delay(200);
            digitalWrite(errorLight, LOW);
            }
            else {
            pirTriggers--;
            }
```

```
            if( pirTriggers <=0) {
            pirTriggers = 0;
            pirSafetyVal = false;
            safetyStopVal = false;
            }
            }
  }
  if (safetyStopVal == false) {
            digitalWrite(safetyStopLight, LOW);
  }
//-------------------------------------------------------------------------------------------
  if(digitalRead(safetyButton) == HIGH) {
            // This function is a toggle for the safety stop button; toggles true or false
            if(safetyStopVal == false) {
            safetyStopVal = true;
            }
            if(safetyStopVal = true) {
            safetyStopVal = false;
            }
  }
  // Below reads the value of potentiometers and then maps them to degrees for the servos to move to
  basePotVal = analogRead(basePot);
  basePotAngle = map(basePotVal, 0, 1023, 0, 179);
  shoulderPotVal = analogRead(shoulderPot);
  shoulderPotAngle = map(shoulderPotVal, 0, 1023, 0, 179);
  elbowPotVal = analogRead(elbowPot);
  elbowPotAngle = map(elbowPotVal, 0, 1023, 0, 179);
```

```
  // Below runs the servos to the potentiometer positions
  baseServo.write(basePotAngle);
  shoulderServo.write(shoulderPotAngle);
  elbowServo.write(elbowPotAngle);
//-------------------------------------------------------------------------------------------
  if(digitalRead(posSaveButton) == HIGH) {
        // Checks if the position save button has been pressed
        posSaveButtonPresses++;
        switch (posSaveButtonPresses) {  // Executes a different case depending on how many times the button is
pressed
        case 1:  // Saves potentiometer positions to servo angles memory 1
        baseServoSavedPos[0] = basePotAngle;
        shoulderServoSavedPos[0] = shoulderPotAngle;
        elbowServoSavedPos[0] = elbowPotAngle;
        break;
        case 2:  // Saves potentiometer positions to servo angles memory 2
        baseServoSavedPos[1] = basePotAngle;
        shoulderServoSavedPos[1] = shoulderPotAngle;
        elbowServoSavedPos[1] = elbowPotAngle;
        case 3:  // Saves potentiometer positions to servo angles memory 3
        baseServoSavedPos[2] = basePotAngle;
        shoulderServoSavedPos[2] = shoulderPotAngle;
        elbowServoSavedPos[2] = elbowPotAngle;
        case 4:  // Saves potentiometer positions to servo angles memory 4
        baseServoSavedPos[3] = basePotAngle;
        shoulderServoSavedPos[3] = shoulderPotAngle;
```

```
            elbowServoSavedPos[3] = elbowPotAngle;
            case 5:  // Saves potentiometer positions to servo angles memory 5
            baseServoSavedPos[4] = basePotAngle;
            shoulderServoSavedPos[4] = shoulderPotAngle;
            elbowServoSavedPos[4] = elbowPotAngle;
            }
 }
 if(digitalRead(posRunButton) == HIGH) {
            posRunButtonIsPressed = true;
 }

 if(posRunButtonIsPressed == true) {
            // Runs through memory positions
            // Checks safety of work environment, passes the first memory saved position as the initial position
            if(safetyStopVal == false) {
            for(int i = 0; i<5; i++) {  // Cycles through each memory position for servo
            // Moves each servo to it's ith memory position
            motorRun(baseServo.read(),i);  // Runs the motors
            delay(1050);  // Stops robot operations for 1.5 secons)
            }
            }
 }
 delay(300);  // Stops robot for 0.3 seconds
}
```

Programmable Safety Arm, Version 3:

```
// Global variables, these need to be referenced throughout the code
```

```
#include <Servo.h>  // Allows servos to be used; Servo module
Servo baseServo;
Servo shoulderServo;
Servo elbowServo;

// Assign LED's to choosen out pin's
const int activityLight = 4;
const int posSaveLight = 2;
const int errorLight = 11;
const int safetyStopLight = 7;

// Assign Buzzer pin and PIR pin
const int buzzerPin = 10;
const int pirPin = 9;
boolean pirSafetyVal = false;
boolean safetyStopVal = false;
int pirTriggers = 0;  // This is the number of low to high triggers on the PIR

// Sets up variable for buttons, which are used for commands
const int posSaveButton = 13;
const int posRunButton = 12;
const int safetyButton = 8;

int posSaveButtonPresses = 0;
boolean posRunButtonIsPressed = false;
boolean safetyButtonVal = false;
```

```
// Rotary potentiometer setup to allow for manual control of servos
const int basePot = A0;  // Pin for base potentiometer
const int shoulderPot = A1;  // Pin for shoulder potentiometer
const int elbowPot = A2;  // Pin for elbow potentiometer
// Angles and values for each servo-pot combo. Angles for servos, vals for potentiometers
int basePotVal;
int basePotAngle;
int shoulderPotVal;
int shoulderPotAngle;
int elbowPotVal;
int elbowPotAngle;

// Below are the saved positions for the three servo's, these are cycled when posRunButtonIsPressed = true
int baseServoSavedPos[] = {1,1,1,1,1};
int shoulderServoSavedPos[] = {1,1,1,1,1};
int elbowServoSavedPos[] = {1,1,1,1,1};
// Defaults for all are {1,1,1,1,1} (five positions, for five saves in memory)

void setup() {
  // Setup pins for LED's (outputs) and pins for buttons (inputs)
  pinMode(activityLight, OUTPUT);
  pinMode(posSaveLight, OUTPUT);
  pinMode(errorLight, OUTPUT);
  pinMode(safetyStopLight, OUTPUT);
  pinMode(posSaveButton, INPUT);
  pinMode(posRunButton, INPUT);
  pinMode(safetyButton, INPUT);
```

```
  // Setup pins for PIR and buzzer
  pinMode(buzzerPin, OUTPUT);  // This is for the warning buzzer
  pinMode(pirPin, INPUT);  // Sets the PIR input pin as an input
  // Setup servos
  baseServo.attach(3);
  shoulderServo.attach(6);
  elbowServo.attach(5);
  // setup pots
  pinMode(basePot, INPUT);
  pinMode(shoulderPot, INPUT);
  pinMode(elbowPot, INPUT);

  digitalWrite(errorLight, LOW);

  Serial.begin(9600);
}

void safeCheck() {
  if(digitalRead(pirPin)==LOW) {
        if(digitalRead(pirPin)==HIGH) {
        pirTriggers++;
        Serial.print('PIR Triggered');
        digitalWrite(errorLight, HIGH);
        digitalWrite(buzzerPin, HIGH);
        delay(300);
        digitalWrite(errorLight, LOW);
        digitalWrite(buzzerPin, LOW);
```

```
            }
  }
  if(pirTriggers>=1) {
            pirSafetyVal = true;
            safetyStopVal = true;
            digitalWrite(safetyStopLight, HIGH);
  }
}
// Function that runs through motors while looking at safety value
void motorRun(int sPos1, int i) {
  if(digitalRead(pirPin)==LOW) {
            if(digitalRead(pirPin)==HIGH) {
            pirTriggers++;
            Serial.print('PIR Triggered');
            digitalWrite(errorLight, HIGH);
            digitalWrite(buzzerPin, HIGH);
            delay(300);
            digitalWrite(errorLight, LOW);
            digitalWrite(buzzerPin, LOW);
            }
  }
  baseServo.write(baseServoSavedPos[i]);
  digitalWrite(activityLight, HIGH);
  delay(300);
  if (pirTriggers <1) {
            safeCheck();
            if ((safetyStopVal == false) && (digitalRead(pirPin) == LOW)) {
```

```
            shoulderServo.write(shoulderServoSavedPos[i]);
            safeCheck();
            }
            if ((safetyStopVal == false) && (digitalRead(pirPin) == LOW)) {
            elbowServo.write(elbowServoSavedPos[i]);
            }
            delay(200);
            digitalWrite(activityLight, LOW);
  }
  else {
            digitalWrite(activityLight, LOW);
            digitalWrite(errorLight, HIGH);
            digitalWrite(buzzerPin, HIGH);
            digitalWrite(safetyStopLight, HIGH);
            delay(200);
            digitalWrite(errorLight, LOW);
            digitalWrite(buzzerPin, LOW);
            baseServo.write(sPos1);
            pirSafetyVal = true;
            safetyStopVal = true;
  }

}

void loop() {
  basePotVal = analogRead(basePot);
  basePotAngle = map(basePotVal, 0, 1023, 0, 180);
```

```
  shoulderPotVal = analogRead(shoulderPot);
  shoulderPotAngle = map(shoulderPotVal, 0, 1023, 0, 180);
  elbowPotVal = analogRead(elbowPot);
  elbowPotAngle = map(elbowPotVal, 0, 1023, 0, 180);

  // Below runs the servos to the potentiometer positions
  baseServo.write(basePotAngle);
  shoulderServo.write(shoulderPotAngle);
  elbowServo.write(elbowPotAngle);

  // Check PIR
  if(digitalRead(pirPin)==LOW) {
            if(digitalRead(pirPin)==HIGH) {
            pirTriggers++;
            Serial.print('PIR Triggered');
            digitalWrite(errorLight, HIGH);
            digitalWrite(buzzerPin, HIGH);
            delay(200);
            digitalWrite(errorLight, LOW);
            digitalWrite(buzzerPin, LOW);
            }
  }
//-------------------------------------------------------------------------------------------
  // Below reads the value of potentiometers and then maps them to degrees for the servos to move to
  basePotVal = analogRead(basePot);
  basePotAngle = map(basePotVal, 0, 1023, 0, 179);
  shoulderPotVal = analogRead(shoulderPot);
```

```
  shoulderPotAngle = map(shoulderPotVal, 0, 1023, 0, 179);
  elbowPotVal = analogRead(elbowPot);
  elbowPotAngle = map(elbowPotVal, 0, 1023, 0, 179);

  // Below runs the servos to the potentiometer positions
  baseServo.write(basePotAngle);
  shoulderServo.write(shoulderPotAngle);
  elbowServo.write(elbowPotAngle);
//-------------------------------------------------------------------------------------------
  if(digitalRead(posSaveButton) == HIGH) {
        // Checks if the position save button has been pressed
        posSaveButtonPresses++;
        switch (posSaveButtonPresses) {  // Executes a different case depending on how many times the button is
pressed
        case 1:  // Saves potentiometer positions to servo angles memory 1
        baseServoSavedPos[0] = basePotAngle;
        shoulderServoSavedPos[0] = shoulderPotAngle;
        elbowServoSavedPos[0] = elbowPotAngle;
        Serial.print('case 1');
        break;
        case 2:  // Saves potentiometer positions to servo angles memory 2
        baseServoSavedPos[1] = basePotAngle;
        shoulderServoSavedPos[1] = shoulderPotAngle;
        elbowServoSavedPos[1] = elbowPotAngle;
        Serial.print('case 2');
        break;
        case 3:  // Saves potentiometer positions to servo angles memory 3
```

```
            baseServoSavedPos[2] = basePotAngle;
            shoulderServoSavedPos[2] = shoulderPotAngle;
            elbowServoSavedPos[2] = elbowPotAngle;
            Serial.print('case 3');
            break;
            case 4:  // Saves potentiometer positions to servo angles memory 4
            baseServoSavedPos[3] = basePotAngle;
            shoulderServoSavedPos[3] = shoulderPotAngle;
            elbowServoSavedPos[3] = elbowPotAngle;
            Serial.print('case 4');
            break;
            case 5:  // Saves potentiometer positions to servo angles memory 5
            baseServoSavedPos[4] = basePotAngle;
            shoulderServoSavedPos[4] = shoulderPotAngle;
            elbowServoSavedPos[4] = elbowPotAngle;
            Serial.print('case 5');
            break;
            }
}
if(digitalRead(posRunButton) == HIGH) {
            posRunButtonIsPressed = true;
            Serial.print('pos Read');
}

if(posRunButtonIsPressed == true) {
            // Runs through memory positions
            // Checks safety of work environment, passes the first memory saved position as the initial position
```

```
            if(safetyStopVal == false) {
            for(int i = 0; i<5; i++) {  // Cycles through each memory position for servo
            // Moves each servo to it's ith memory position
            motorRun(baseServo.read(),i);  // Runs the motors
            Serial.print('Motor run ' + i);
            delay(1050);  // Stops robot operations for 1.5 secons)
            }
            posRunButtonIsPressed = false;
            posSaveButtonPresses = 0;
            }
  }
//-------------------------------------------------------------------------------------------
  while (pirSafetyVal == true) {  // Loop to check safety
            digitalWrite(safetyStopLight, HIGH);
            if(digitalRead(pirPin)==LOW) {
            delay(300);
            if(digitalRead(pirPin)==HIGH) {
            pirTriggers++;
            Serial.print('PIR Triggered');
            digitalWrite(errorLight, HIGH);
            digitalWrite(buzzerPin, HIGH);
            delay(200);
            digitalWrite(errorLight, LOW);
            digitalWrite(buzzerPin, LOW);
            delay(1000);
            }
            else {  // Subtract from PIR triggers
```

```
            pirTriggers--;
            Serial.print('Untriggered PIR');
            delay(1000);
            }
            if( pirTriggers <=0) {
            pirTriggers = 0;
            pirSafetyVal = false;
            }
            }
}
if (safetyStopVal == false) {  // Turns off safety light if value is false
            digitalWrite(safetyStopLight, LOW);
}
if(digitalRead(pirPin) == HIGH) {
            digitalWrite(safetyStopLight, HIGH);
            safetyStopVal = true;
            pirSafetyVal = true;
            delay(500);
            digitalWrite(safetyStopLight, LOW);
}
if(digitalRead(safetyButton) == HIGH) {
            // This function is a toggle for the safety stop button; toggles true or false
            switch(safetyStopVal){
            case false:
            safetyStopVal = true;
            digitalWrite(safetyStopLight, HIGH);
            digitalWrite(buzzerPin, HIGH);
```

```
            break;
            case true:
            safetyStopVal = false;
            digitalWrite(safetyStopLight, LOW);
            digitalWrite(buzzerPin, LOW);
            break;
            }
 }
 if(safetyStopVal == true) {  // Turns on safety light if value is true
  digitalWrite(safetyStopLight, HIGH);
 }
 delay(300);  // Stops robot for 0.3 seconds
}
```

Test Planning and Results

1. Planned Tests

To ensure that the Programmable Safety Arm is functioning to its correct specifications, there are two tests that must be performed: the movement test, and the safety test. These will verify that the features of the PSA are working correctly.

For the movement test, it is as simple as ensuring the robot is able to move correctly. The first part of the test requires the moving of the potentiometers to check if the robot responds. For each potentiometer, when spun, a corresponding servo should respond by spinning. If this is true for all three potentiometers and servos, then the first part of the test is complete. The second portion of the test involves setting up the memory positions and running through them. Choose five different robot positions and save each one by moving the robot to the position and then

pressing the position save button n times (up to 5 for position 5). Then, click the run positions button. If the robot cycles through five positions uninterrupted (unless by the safety), then the test is a success. If issues are found in the movement test, then diagnosis¶ should be performed on the potentiometers and servos by looking at their connections to the controller and Arduino, as well as the components themselves (to check for damage).

For the next test, safety controls are focused on. First off, you must run the robot through any set of saved positions. While it is running, press the safety stop button. If the robot stops, then the test is a success. For the next test, once again run the memory positions. This time move in front of the PIR to trigger it. If the robot stops moving, stops for some time, and then continues moving, the test is a success. In the safety test, if the robot stops constantly without any actual triggering, adjust sensitivity of the PIR and number of trips required. If the robot does not resume work after PIR has been deactivated, troubleshoot code. If the PIR does not trigger, or the robot does not stop, diagnosis is required. If code is correct, check PIR connections (if servos are already proven to work).

If the above two tests are successful, then then the basic functional requirements have been met. There are three degrees of freedom (assuming that there are three servos), there is manual control, there is programmability, and there is a working motion sensor. For the remaining requirements, the following minor tests are performed:

- Check to see if the activity LED runs while robot is moving, the warning LED lights up on PIR trigger, the stop LED lights up on stop button press, and PIR triggered stop, as well as the position save light running when positions are saved.
    - If all of these tests are successful, the LED feedback requirement has been met.

- Test the softness and resistance of the robot's material. If the resistance to pressure is relatively low, as is the softness relatively high, then the soft-shell design requirement has been met.

Design constraints must also be tested. If the robot is small in comparison to conventional robots, then the size constraint has been satisfied. If the robot is made of relatively inexpensive parts, then the price constraint has been met (this can be verified using the Bill of Materials and the prices listed; these are the two satisfiable constraints.

2. Testing

Testing Date: 16/01/2019

All robot features seem to be flawed. Movement test has failed entirely; servos do not move or respond to the potentiometers. However, they are actively powered and drawing minimal current (nowhere close to stall current). Motor functionality is proven as fluctuations in electricity cause slight movement (this can likely be solved with a capacitor connected to all servos [10uf], should be implemented). This means that issue must be with the potentiometers. Upon inspection, potentiometers do not seem to be receiving power. Must be correctly re-integrated into circuit.

In terms of the safety test, it is inconclusive as it is untestable due to inability of servos to move. However, buttons do not respond to any clicks (should be signified by activity/position save/safety light). Safety light does not turn on, PIR test inconclusive as safety light does not turn on and motors do not respond. Wiring for buttons seems to be flawed, must be redone. Wiring for LEDs must be edited to match with the wiring of the buttons; correct the flow of electricity.

Buzzer does not turn on, likely due to no power (as tested to 0 volts [a.k.a. Out of circuit]. Must be integrated correctly).

In regard to minor tests, all negative. Error light appears to always be on, no other LEDs respond. This is likely due to improper wiring; must be adjusted with previous rewiring required. (see above).

Virtual Redesign Testing Date: 16/01/2019

Changes made:

- Corrected wiring for potentiometers and buttons
    - Potentiometers are now all powered by the 5-volt connection on Arduino. This means that they now share a common ground and common power source.
    - Buttons share a similar setup to potentiometers; now powered by 3.3volt connection on Arduino, sharing a common ground with the potentiometers and Arduino.
- Corrected wiring of LED
    - Were originally designed to be outputting to Arduino. This method was flawed; replaced with an Arduino output leading to led activation; joined at a common ground (shared with the potentiometers and buttons).
    - Software changes made to accommodate for corrected method of LED activation.
- Corrected wiring of PIR
    - PIR is now powered by the same power supply as motors (separate from Arduino). Shares a common ground, and is connected via signal pin to Arduino (pin 9)

- Software edit made to accommodate for PIR change to pin 9

Using a virtual simulation of the device, with the exact same circuit and software (on tinkercad), motors are now confirmed to work correctly. They respond to the potentiometers now. Motors hover around a voltage of 3.4 - 4.2 volts at most times. Current fluctuates depending on load (40mA+). Potentiometers are not very accurate in motor control however. Buttons work, positions save button. Run positions button runs positions. However, position save button seems to provide some degree of noise to the Arduino as it seems to be throwing multiple times in a single click (this could also be a software problem where the button is read from too fast, resulting in overlapping values in a single click). This needs to be rectified (through software or hardware). Overall, motor tests result in basic functionality, some tweaking is needed.

In regard to safety tests, safety button, when pressed, now triggers the safety light and prevents the robot from running memory positions. If memory positions are running when button is pressed, then they are stopped. If safety is disabled, light turns off, and positions continue. However, safety button is very sensitive. This could be solved with a delay on the button press in software, however this would interfere with other robot functioning (the delays would add up and reduce the simulated “refresh rate” of the safety check loop). PIR, when triggered, turns on safety light; enabling safety mode. However, when it is untriggered, work does not resume. Everything must be manually re-enabled by toggling the safety button. This could potentially be solved through software. However, implementation would be extensive without the use of a dedicated synchronous tasks manager/module (which is not provided in Arduino due to the hardware limitation of a single core and single thread). Software-simulated multithreading it too

advanced for Arduino board. As such, the PIR in its current form is likely stuck in its current state.

Changes to make to physical board (already made to virtual/simulated board):

- Correct wiring to match virtual board (functioning correctly)
- Update software with latest version, designed to fix LED outputs

Re-Testing Date: 17/01/2019

Changes made pre-test:

- Potentiometers, buttons, and LED's rewired to match virtual circuit
    - Attached 5v and 3.3v of Arduino to power potentiometers and buttons (respectively)
- Code updated with latest version (to accommodate for hardware changes)
    - Changed LED activation method (outputs HIGH, delays, outputs LOW)
    - Increased Sensitivity of PIR (Lowered triggered threshold for faster activation)
- PIR connected to board according to digital circuit

Motion test results are mostly positive. ⅔ potentiometers work, resulting in ⅔ servos responding. However, the one inactivated servo works when connected to a working potentiometer. Thus, there is a problem with the singular potentiometer wiring (unable to diagnose problem, wiring appears fine). 3/3 motors are active when powered, but due to there only being two servos, only two respond. When position save button is pressed, the position is successfully saved to memory. This means that when the position run button is pressed, the positions are successfully run. The issue of faulty memory values (present in the virtual test) still persists. This requires more tweaking on the position save button. While servos respond to

potentiometers, the movement is not very precise. This is likely due to the resistance in the potentiometers or the voltage being fed to them. (It is predicted that a higher voltage will result in a larger range of motion. A mechanical rotary potentiometer might also solve the problem with finer resistance adjustment). In addition, while robot is in motion, the activity light activates. However, the position save light does not. Upon further inspection, these seems to be a software flaw, and is thus fixable with an update. Overall, motion of robot works satisfactorily, despite the flaw present in the single potentiometer (must be corrected).

In terms of safety testing, further revisions are required. When PIR is not connected to board, Safety loop detects false positives, flashing the safety light. However, this does not seem to prevent robot movement. (The physical hardware is acting differently from its identical virtual counterpart). When PIR is connected, the light flashes only when triggered. However, the PIR is extremely sensitive, resulting in constant false positives. This needs to be drastically reduced to provide a reliable sensor system. Furthermore, the safety trip still does not stop motor movement (as it does for the virtual circuit). Due to the false positives of the PIR, the safety stop button becomes practically useless. In theory, it works. However, if safety is turned off, the PIR triggers it back on. PIR must be fixed for a proper safety system. Another issue is found in the buzzer. When the safety stop button is clicked, the warning buzzer does not go off. This is either due to a flaw in the code, faulty wiring, or the PIR overriding the safety switch. Upon further inspection, it seems that the buzzer it not receiving enough power to run (requires 3v[min], receiving 1 volt). This means that its wiring must be reconfigured as to allow it to use the Arduino 3.3v or 5v supply (however, when there is anything but potentiometers on the 5v power, then neither component acts correctly [the root of this issue is not known]). In addition, the warning LED

seems to always be active, despite not having any PIR triggers. Although there is a low possibility that this is a hardware/wiring issue, it is likely a conflict between the serial communication line opened by the software, as the warning led is connected to Tx pin 1. (This is converted to a serial communication pin when Serial.begin is used in the code [which it is]). Overall, the safety test results are a full negative. The only safety feature that works (in theory) is the safety stop. However, the flawed connections of the PIR and Buzzer take away from the end result. More changes are required.

As a whole, it seems that that physical circuitry does not work like the virtual circuitry, despite being the exact same (minus power supply [4.5 volts vs. 5 volts]). This can likely be chalked up to wiring issues (faulty/loose/shorting wires). As this would be the only other difference. The potentiometers used in the simulation circuit are not the same as those used in the physical circuit. This likely contributes to their lack of precision.

Changes required for complete design:

- Deletion of serial communication to allow functioning of warning light
- Rewiring of buzzer to allow sufficient power for activation.
- Reduce sensitivity of PIR in software (through triggers required for safety) to prevent false-positives.
    - It is possible that software updates will not be sufficient, in which case the PIR must be swapped out for a different sensor with similar functionality.
- Diagnose and rewire the faulty potentiometer
    - Possibly replace all potentiometers with more precise ones, to allow finer moment

        - Precision could be manually adjusted through software by editing the resistance-to-degree mapping conversion
- Overall system wiring needs major adjustment
    - Clipping of excess wire, organization of wires, untangling, finding and fixing faulty/loose connections. This will make the system more reliable
- Servos are a little finicky in movement, this could be resolved with the addition of a capacitor to handle excess charge in the circuit for the servos.

Conclusions

1. Met Design Requirements:

Overall, the prototype model of the Programmable Safety Arm met all of its design requirements to some degree. The main requirements of focus are the 3 degrees of freedom, programmability, motion sensor technology, soft shell design, LED feedback, and manual control. The servo control system manages to satisfy the 3 degrees of freedom requirement in its entirety. However, due to the quality of the parts the robot's speed and stall torque was limited, hindering its functionality in movement to some degree. In the case of programmability, the PSA satisfies what it set out to do. It allows the user to save and run robot positions, offering such a degree of programmability in its design. Thus, the design requirement has been met excellently. When looking at motion sensor technology, the PSA fulfils the bare minimum requirements. Due to the simple sensor array (consisting of a singular sensor) and the lack of programmability in the sensor itself, the robot manages to provide some degree of motion sensor technology to enhance safety. However, the sensor is prone to false-positives, reducing the functionality of the robot.

Furthermore, the robot has trouble resuming its tasks after a safety stop, requiring manual intervention. In the case of a soft-shell design, this too is only met to some degree. Although the cardboard and corrugated plastic design of the Programmable Safety Arm is relatively safe (taking into consideration that it is a prototype model). They do not provide any significant structural stability and are still able to catch loose articles (clothing, hair, etc.) reducing the overall safety of the design. However, in direct impact, the robot is not able to cause serious damage, partially due to its structure, and partially due to its limited strength. A production model of the robot would likely use a more advanced soft-shell design (ideas include rubber-coated metal/plastic, rubber body). When looking at the requirement of LED feedback, the Programmable Safety Arm satisfies such a requirement to some degree. The LED subsystem responds to some tasks accurately, conveying adequate functioning information to the user. The activity light conveys optimal functioning when the robot is running. The warning light conveys the potential for danger to the robot (in theory, however not in practice)., the position save light indicates to the user that the command has been processed (entirely possible, but not implemented in software correctly), and the stop light indicates that the robot has gone into safety stop mode (however it is triggered by false-positives from the PIR). In terms of manual control, the potentiometer and servo subsystem manages to effectively satisfy this requirement by allowing manual movement of each servo when the corresponding potentiometer is turned. Success is hindered to some degree, however, by the precision of manual control. The values of the potentiometer do not transfer in a one-to-one ratio of movement, producing some degree of instability in manual control; taking away from overall functionality.

Although with some minor issues, the Programmable Safety Arm is able to fulfil the minimums of all of its requirements, as well as fully satisfying some. This means that LED Feedback, three degrees of freedom, manual control, soft-shell design, motion sensor technology, and programmability, are all present in the device through its corresponding subsystem.

2. Remaining Issues:

The Programmable Safety Arm is a success, in theory. However, there remains much to refine and redo in its overall design to provide a better experience. The three main issues that exist with the robot revolve around its safety control, as well as the potential for expansion on power and interfacing.

The safety control system on the Programmable Safety Arm, while satisfactory, has the potential for much revisioning. Foremost, a singular sensor alone is only able to provide a small degree of coverage, that too to a simple depth due to the simplicity of the sensor. The safety design could be drastically enhanced if using a larger quantity of smarter sensors. The PIR used runs on its own closed loop system and thus only provides the PSA's systems with information one whether or not it is triggered. This means that information on exact numbers of triggers, direction of movement, distance of trigger, or intensity of heat detected are not available through a lower-end hobby PIR. With a smarter, more expensive PIR (or similar sensor) the robot is significantly smarter; being able to choose which direction to move in (to prevent injury), and most of all, enhancing its ability to distinguish actual organisms from false positives. Another major problem with the safety system is that of workflow. In its current state, the PIR system, when triggered, results in the stoppage of all work. Position movement does not resume when the PIR has been reset, resulting in workflow being hindered; requiring manual intervention to

continue. This issue could be solved with more intelligent servo motors. The one's in use of the prototype are not able to provide a true angular position in real-time. When the servo's position is read, it is only the last written position being recalled. This means that there is no way to know where the servo is at a given time (precisely), and this no way to efficiently resume work when the PIR is no longer triggered.

The next system of analysis revolves around the expansion of power in the system. In its current state, the PSA is not a very strong robot, due mostly to the parts it is made of. For a full, production line model, stronger materials, and most importantly, motors/servos must be used. The current servos are relatively weak and are thus unable to lift heavy loads; making the prototype robot unconventional in real tasks. Thus, the total torque output of the robot must be enhanced. The best way to do this would be through the use of larger or higher quality servo motors; able to provide more torque. Additional options revolve around the use of doubled-up servos, where every singular servo connection point is given another servo attached to it. This expansion of the design, however, would require some degree of remodeling in the robot's structure.

Possibly the biggest enhancement that could be added to the Programmable Safety Arm is that of an improved interface. The current interface revolves around color and position coded lights to convey crucial information; however, this could be expanded with the use of more complex parts and packaging. By adding a display, more information could be displayed in a clearer manner. This would also make the interface more software reliant, and thus, clean up the wiring. The second possibility of expansion revolves around a wireless, packaged, controller. The current controller is very simple and relies on multiple wire-links to the main robot. This

means that any users must be in a close vicinity to the device. Using a receiver and transmitter system, the controller could be separated from the robot, allowing it to be used remotely. This would drastically enhance the user experience by making to robot's functioning more convenient, as well as reducing the risk of close-proximity-induced injury.

3. Reflection

The design and assembly of the Programmable Safety Arm was most definitely a learning experience. To complete the project, I had to do many things which I had never done before, or to the same degree as this. Foremost, the PSA was my first real usage of an Arduino in connection to conventional circuitry. As such, through the interfacing required between the various parts of my circuit and the Arduino, I learned more about how to organize my circuitry, as well as the procedure when connecting it to an Arduino. Due to the large quantity of wires I used, I also learned about how to manage my wires better. To better make use of my time, I had to think of a way to organize my circuitry in a manner as to allow me to connect and disconnect it at a moment's notice. I discovered that this can be easily accomplished through grouping my wires by their function, labelling of important wires, as well as color-coding. Overall, this helped me improve my organizational skills. Perhaps the most important thing that I learned about in creating the Programmable Safety Arm is exactly what I set out to do in the robotics course: learn how to write software that can be used to interface with the physical world. Through the requirements of writing pseudocode, as well as actual working code (including the calibration and adjustments required to it), I learned how to better structure my programs to collaborate with physical components as well as how to write code that actually performs tasks using physical components. For example, in my use of micro-servos I learnt how to use read from and write to

servos for movement. This required the use of software to convert values from other inputs into values that servos understood (such as converting resistance values into degree correspondents for the potentiometer-servo pairs). I also learned the basics of using sensors and managing the outputs that they give off with their outputs; something that I had not previously done.

The final design project, and the PSA, allowed me to learn many things. Some of these things will surely be beneficial assets in my future career. As someone pursuing a career in software, learning to interface with physical systems will likely stand as a crucial skill. Through my understanding of not just the digital world of software, but the physical world, I have the upper hand in solving problems using my software, by being able to implement hardware in collaboration with software; where software alone is not sufficient. Furthermore, I also learned how to structure my code better through the pseudocode design which was required in the assignment. Although I had written software before, I had never gone through a pseudocode design process. By learning how to write pseudocode, I gained a skill that will surely benefit me in writing complex programs, by being able to provide me with an outline of its various functions.

Overall, the robotics final project, and my attempt at creating the Programmable Safety Arm, were excellent experiences to my education, and provided me with some skills that are not only practical and enjoyable, but some that are likely to prove beneficial later down the road in my career.

Handout

Follow this link for a pdf of the handout:

https://drive.google.com/file/d/1t2FDPZM1Y_kL0EgaHK1Chq3azjQ8_Q9G/view?usp=sharing

References

DIY Arduino Robotic Arm. (n.d.). Retrieved from https://circuitdigest.com/microcontroller-projects/diy-arduino-robotic-arm-tutorial

Language Reference. (n.d.). Retrieved from https://www.arduino.cc/reference/en/

PIR Motion Sensor. (n.d.). Retrieved from https://learn.adafruit.com/pir-passive-infrared-proximity-motion-sensor/using-a-pir-w-arduino

SICK Safety Relays(2018, May 18). Retrieved December 24, 2018, from https://www.youtube.com/watch?v=-KDpbLT8dM8

Safety relays. (n.d.). Retrieved from https://www.sick.com/ca/en/senscontrol-safe-control-solutions/safety-relays/c/g186153

Staff, R. (2018, January 26). What is Lidar and How Does it Help Robots See? Retrieved from https://www.roboticsbusinessreview.com/rbr/what_is_lidar_and_how_does_it_help_robots_see/

Velodyne Lidar. (n.d.). Retrieved from https://velodynelidar.com/hdl-64e.html