Five Stage Pick and Place Robot

Controls Does Micro Final Project Report

Ross Alexander | Rebecca Schwartz | Jeremy Hamlin

May 15th, 2020

Background and Motivation

With recent advances in pressure sensing technology for robotic manipulation of fragile objects, our team originally intended to build a robot arm for dipping strawberries in chocolate. We altered the scope of our project to focus only on the pick and place action of a robotic arm, keeping in mind that pressure sensing could eventually be integrated. Our team is composed of Mechanical MEng students focusing on the Control of Robotic and Autonomous Systems, so completing a project within robotics made sense.

The pick and place robot arm is an EEZYbotARM MK2, which is an open source 3D printable robot. This robot was chosen because it has four degrees of freedom (DOF), uses off the shelf motors, is relatively easy to construct, and has simplified gripper dynamics. The mechanical design of this robot prevents the gripper from rotating about the y-axis. This simplifies the grippers motion and control (Figure 1, 2).

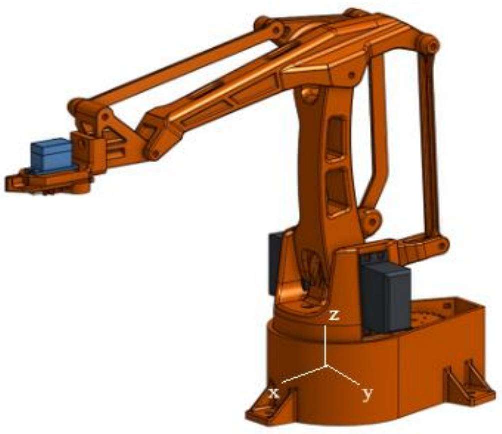


Figure 1: 3D Model with Cartesian Coordinate System Reference

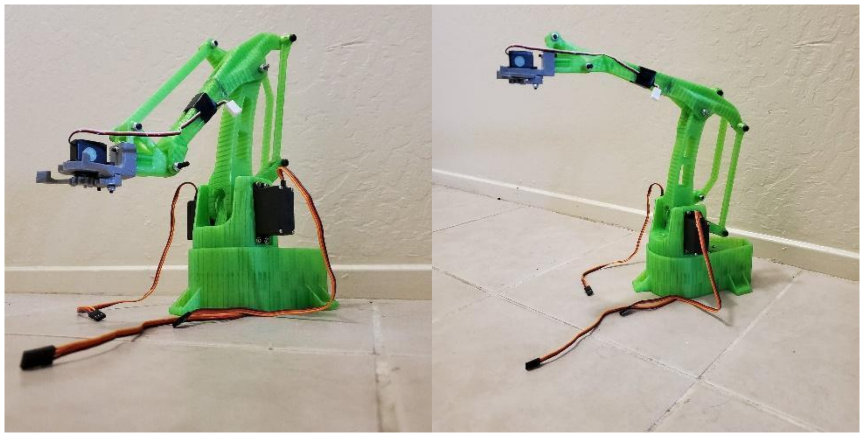


Figure 2: Gripper Independent of Y-axis Rotation

The pick and place robot is controlled by one Cypress PSOC 5LP microcontroller, which receives serial commands from a LabVIEW user interface. Using the Programming Mode, the user can create a custom pick and place trajectory from five save positions: pre-pick, pick, home (central position), pre-place, and place (Figure 3). Note that at the pick and place positions, the gripper goes to user defined close and open angles, respectively. Once those positions are saved, the user can execute pick and place motion by either: directing the arm to a specified save point, or starting an automation sequence which continuously directs the robot from Home Position → Pre-Pick Position → Pick Position → Home Position → Pre-Place Position → Place Position and then back to Home Position.

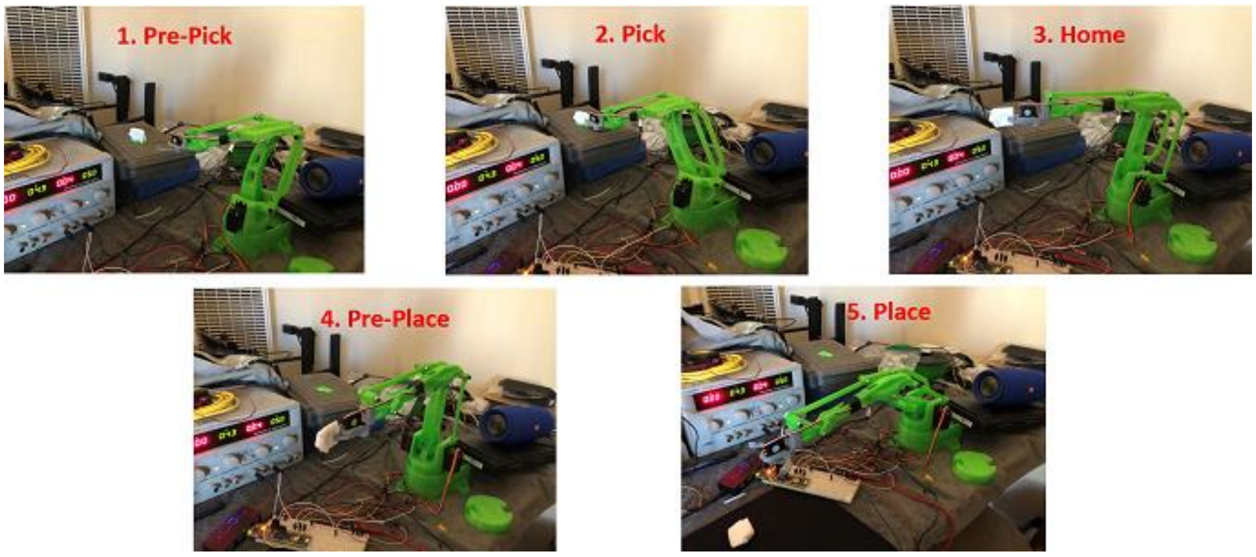


Figure 3: Example of the 5 Major Positions in the Pick-and-Place Sequence

Expected Technical Challenges

Software

The EEZYbotARM MK2 arm is designed to work with servo motors, meaning there is no encoder for position feedback. We expected that one of our biggest challenges would be accurately knowing the position of the robot arm in real time. With properly calibrated and functional servo motors, we can be confident of the position of the arm in steady state, however we have no feedback during arm motion. We brainstormed two methods for tackling this challenge. The first was to measure the current draw of the motors to determine when they arrived at the specified position. The second was incrementing the PWM by small changes so we could be confident that the current PWM value gave a reasonable representation of the arms current position. We would go on to use PWM incrementation in the form of speed control, as the arm would overshoot its position target when the PWM values were changed directly. This is explained further in the Speed Control section.

Hardware

We expected we would have trouble gripping and picking up objects, due to the limited torque of the motor and the mechanical design of the gripper. We also expected to have some difficulty with 3D printing tolerances and the overall strength of the 3D printed parts. We also anticipated having some difficulties drilling, hammering, and attaching the printed parts since they were weak plastic filament.

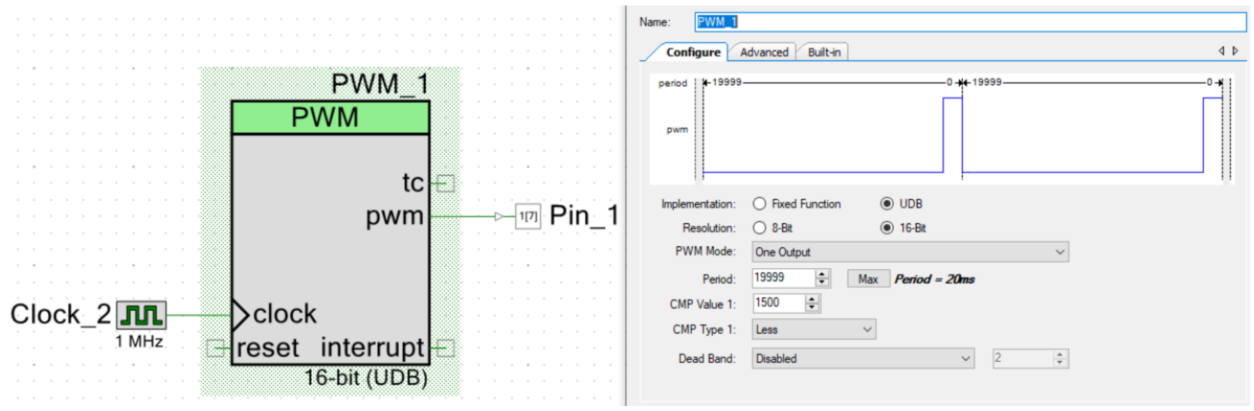
After the quarantine started, we also realized getting hardware would be difficult since a lot of stores closed. While it did not change our plans for printing, it definitely impacted our plans for acquiring screws, filament, nuts and bolts.

Actual Technical Challenges

This section describes our technical solution, focusing on the PSOC code, LabVIEW code, communications protocol, and hardware assembly.

Motor Control via PWM

The first challenge to setting up a pick and place robot is controlling the servo motors using the PSOC. The robotic arm has four servo motors: three to control the arm's motion, and one to control the gripper. Each of these motors is controlled via a PWM signal with a 20 ms period and a pulse width between ~0.5 – 2.5 ms. If the motors operate nominally, the motor position target is represented by the pulse width of the PWM signal. The PWM signals are generated by the PSOC and the pulse widths are controlled by changing a uint16 compare value that represents the pulse width in microseconds.



Name: PWM_1

Implementation:	Fixed Function	[✓ UDB]
Resolution:	8-Bit	[✓ 16-Bit]
PWM Mode:	One Output	
Period:	19999	*Period = 20ms*
CMP Value 1:	1500	
CMP Type 1:	Less	
Dead Band:	Disabled	2

Figure 4: PWM Signal Generation in PSOC Creator

To control the motors, a uint16 PWM value needs to be sent to the PSOC via a serial connection. Since we have four motors, the PSOC needs to be able to receive four distinct PWM commands via serial to control the robot.

Speed Control

Through experimentation we discovered the disadvantages of changing the PWM values of the motors abruptly. The motors would move very quickly towards their intended target and often overshoot due to the inertia generated during motion. To overcome this challenge, we developed a speed control system that split the PWM into a “current” value and a “target” value. When the robot's position is changed, a new target value is sent to the PSOC, which increments the current PWM value towards this target value. This allows us to control the speed of the motors in real time and mitigate position overshoot. Speed control is implemented using a timer interrupt, shown in the figure below.

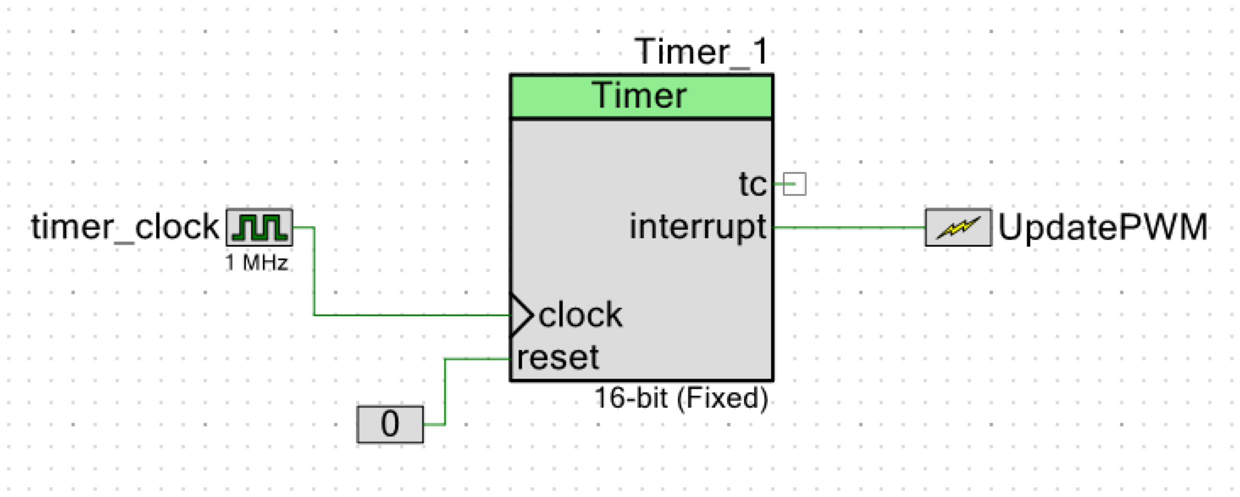


Figure 5: Timer and Update PWM Interrupt for Speed Control

The timer is used to execute the UpdatePWM interrupt at 1 ms intervals. This interrupt increments the current PWM values towards the target PWM values by ±1, corresponding to ±1 additional microsecond of pulse width. This enforces a constant PWM “speed” of 1 microsecond per millisecond which corresponds to a motor speed of approximately 180 degrees per second. Speed control was not used for the gripper motor because the inertia of the gripper was low enough to prevent overshoot and the action was limited to “open fully” and “close fully” commands.

PSOC Software Framework and Communications Protocol

To meet our project objectives, the PSOC needed to be able to receive PWM target values through serial communication. To do this, a case structure was used to receive 11 different serial commands that would affect 19 uint16 variables used to control the servo motors. The variables and commands are summarized in the tables on the next page.

Table 1: PWM uint16 Variables used for motor control

Variable Name	Description (All uint16)	Variable Name	Description (All uint16)
PWM_1_Target;	Target PWM value for motor 1	PWM_1_PrePick;	Saved PWM values for motor 1 Pre-Pick Position
PWM_2_Target;	Target PWM value for motor 2	PWM_2_PrePick;	Saved PWM values for motor 2 Pre-Pick Position
PWM_3_Target;	Target PWM value for motor 3	PWM_3_PrePick;	Saved PWM values for motor 3 Pre-Pick Position
PWM_4_Target;	Target PWM value for gripper motor. This value is written directly to the gripper PWM.	PWM_1_Place;	Saved PWM values for motor 1 Place Position
PWM_1_Current;	Current PWM value sent to motor 1 every loop iteration.	PWM_2_Place;	Saved PWM values for motor 2 Place Position
PWM_2_Current;	Current PWM value sent to motor 2 every loop iteration.	PWM_3_Place;	Saved PWM values for motor 3 Place Position
PWM_3_Current;	Current PWM value sent to motor 3 every loop iteration.	PWM_1_PrePlace;	Saved PWM values for motor 1 Pre-Place Position
PWM_1_Pick;	Saved PWM values for motor 1 Pick position	PWM_2_PrePlace;	Saved PWM values for motor 2 Pre-Place Position
PWM_2_Pick;	Saved PWM values for motor 2 Pick position	PWM_3_PrePlace;	Saved PWM values for motor 3 Pre-Place Position
PWM_3_Pick;	Saved PWM values for motor 3 Pick position		

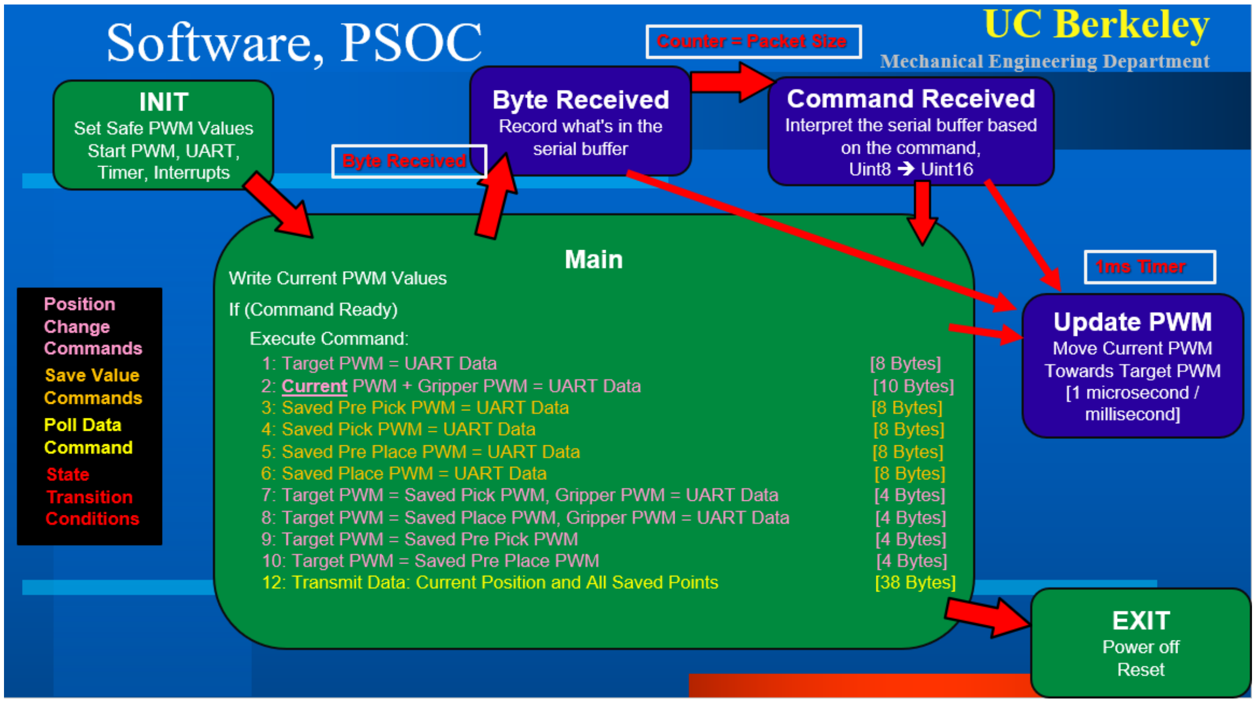
Table 2: PSOC Commands

Command uint8 trigger value	Description
1	Set PWM_(1-3)_Target variables to the uint16 values received via serial
2	Set PWM_(1-4)_Current variables to the uint16 values received via serial. This is a special case that bypasses the speed control and is used to directly control the motor PWMs via serial.
3	Set PWM_(1-3)_PrePick variables to the uint16 values received via serial
4	Set PWM_(1-3)_Pick variables to the uint16 values received via serial
5	Set PWM_(1-3)_PrePlace variables to the uint16 values received via serial
6	Set PWM_(1-3)_Place variables to the uint16 values received via serial
7	Set PWM_(1-3)_Target variables to PWM_(1-3)_Pick values, set gripper PWM [PWM_4_Target] to the uint16 value received via serial
8	Set PWM_(1-3)_Target variables to PWM_(1-3)_Place values, set gripper PWM [PWM_4_Target] to the uint16 value received via serial
9	Set PWM_(1-3)_Target variables to PWM_(1-3)_PrePick values
10	Set PWM_(1-3)_Target variables to PWM_(1-3)_PrePlace values
12	Transmit out all variables, except PWM_4_Target, out via serial in uint8 format

To summarize, the commands can be combined into the following categories:

- Commands [1,7,8,9,10] change the PWM target values
- Commands [3,4,5,6] save PWM values on the PSOC
- Command 2 directly sets the current PWM values of the motors
- Command 12 sends all PWM data out via serial

The overall framework of the PSOC code is presented in the state transition diagram shown below.


Software, PSOC
UC Berkeley
Mechanical Engineering Department
Counter = Packet Size
INIT
Set Safe PWM Values
Start PWM, UART, Timer, Interrupts
Byte Received
Byte Received
Record what's in the serial buffer
Command Received
Interpret the serial buffer based on the command, Uint8 → Uint16
1ms Timer
Update PWM
Move Current PWM Towards Target PWM [1 microsecond / millisecond]
Position Change Commands
Save Value Commands
Poll Data Command
State Transition Conditions
EXIT
Power off
Reset
Main

Write Current PWM Values
If (Command Ready)
Execute Command:

1: Target PWM = UART Data	[8 Bytes]
2: Current PWM + Gripper PWM = UART Data	[10 Bytes]
3: Saved Pre Pick PWM = UART Data	[8 Bytes]
4: Saved Pick PWM = UART Data	[8 Bytes]
5: Saved Pre Place PWM = UART Data	[8 Bytes]
6: Saved Place PWM = UART Data	[8 Bytes]
7: Target PWM = Saved Pick PWM, Gripper PWM = UART Data	[4 Bytes]
8: Target PWM = Saved Place PWM, Gripper PWM = UART Data	[4 Bytes]
9: Target PWM = Saved Pre Pick PWM	[4 Bytes]
10: Target PWM = Saved Pre Place PWM	[4 Bytes]
12: Transmit Data: Current Position and All Saved Points	[38 Bytes]

Figure 6: PSOC State Transition Diagram

Each Block represents a state, with the green blocks representing the main loop while the dark blue blocks are interrupts. All the commands are handled in the main loop using a case structure, while serial communication and speed control are handled via interrupts. The program flows as follows:

1. INIT: Turn on all hardware, set PWMs to safe initial values.
2. Main: continuously run the main loop writing current PWM values to the motors, if we received a command, perform the specified action. Leave this state when an interrupt is triggered or the PSOC shuts down.
3. Byte Received: If a byte is at the serial buffer, interrupt out and read all bytes until the buffer is empty.
4. Command Received: When the number of bytes read by the Byte Received state is equal to the size of the expected message, this interrupt is triggered where the data is parsed based on the command value received.

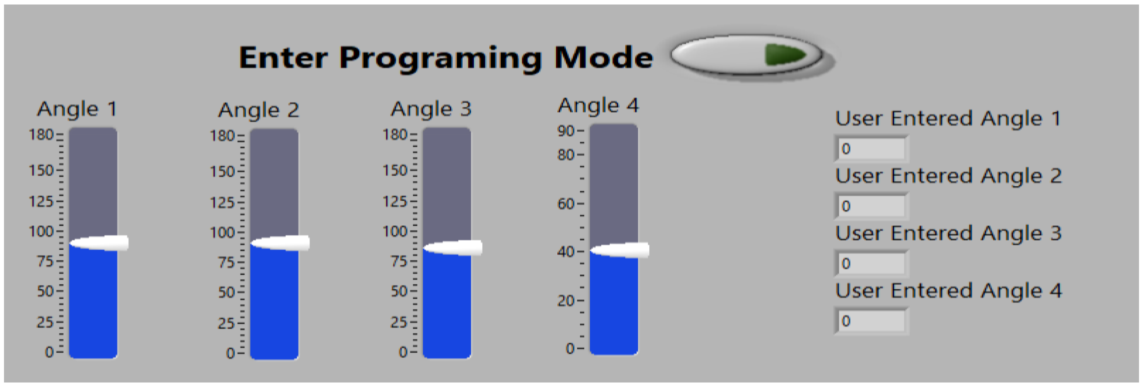
5. Update PWM: This is the highest priority interrupt which will trigger every 1 ms and increment / decrement the current PWM values towards the target values by 1 microsecond of pulse width.

Challenges

The most challenging part of the PSOC code was setting up constant speed control using the timer interrupt. Originally, we tried to include this in the main loop but we had issues because the loop execution time would vary drastically depending on if we had a command. We settled on using a timer interrupt so we could be sure to get accurate speed values in real time.

LabVIEW GUI and User Interface

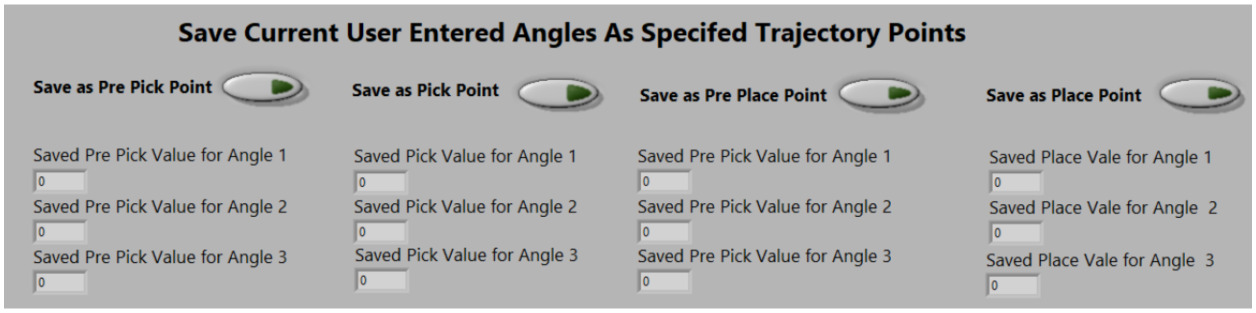
To test our PSOC code, a LabVIEW VI was used to send each of the commands. However, this VI was difficult to use and the primary purpose of our LabVIEW GUI is to present an intuitive interface to the user. To make this interface, a new VI was created that uses buttons and sliders to trigger specific commands on the PSOC. A description of the GUI and the button commands are shown below:



User Entered Angle 1
0
User Entered Angle 2
0
User Entered Angle 3
0
User Entered Angle 4
0

Figure 7: LabVIEW GUI, Programming Mode

The figure above allows the user to enter “Programing Mode” which corresponds to continuously sending command 2 to the PSOC which directly writes the current PWM values as the data received. With the four sliders, the user can directly control the motors and move the arm in real time. The purpose of this mode is to allow for the individual steps (pre-pick, pick, pre-place, place) to be saved on the PSOC.



Saved Pre Pick Value for Angle 1
0
Saved Pre Pick Value for Angle 2
0
Saved Pre Pick Value for Angle 3
0

Saved Pick Value for Angle 1
0
Saved Pick Value for Angle 2
0
Saved Pick Value for Angle 3
0

Saved Pre Pick Value for Angle 1
0
Saved Pre Pick Value for Angle 2
0
Saved Pre Pick Value for Angle 3
0

Saved Place Vale for Angle 1
0
Saved Place Vale for Angle 2
0
Saved Place Vale for Angle 3
0

Figure 8: LabVIEW GUI, Saving Specified Trajectory Points

The figure above shows the four buttons used to save the current slider values as the specified trajectory points. Once all the values have been saved, the robotic arm can be operated using the buttons shown in the figure below.

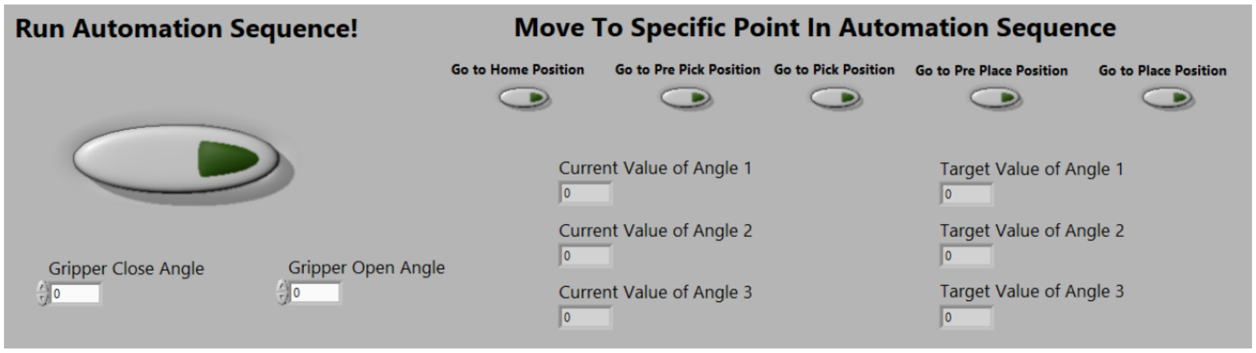


Figure 9: LabVIEW GUI, Automation Sequence

As seen in Figure 3, the automation sequence moves the robot from Home Position → Pre-Pick Position → Pick Position → Home Position → Pre-Place Position → Place Position and then back to home position. Alternatively, the user can use the five buttons on the right to move directly to one of the specified positions. From this screen the user can also specify the closed and open angle of the gripper during picking and placing.

LabVIEW Code Overview

The state transition diagram for the LabVIEW block diagram is shown below.

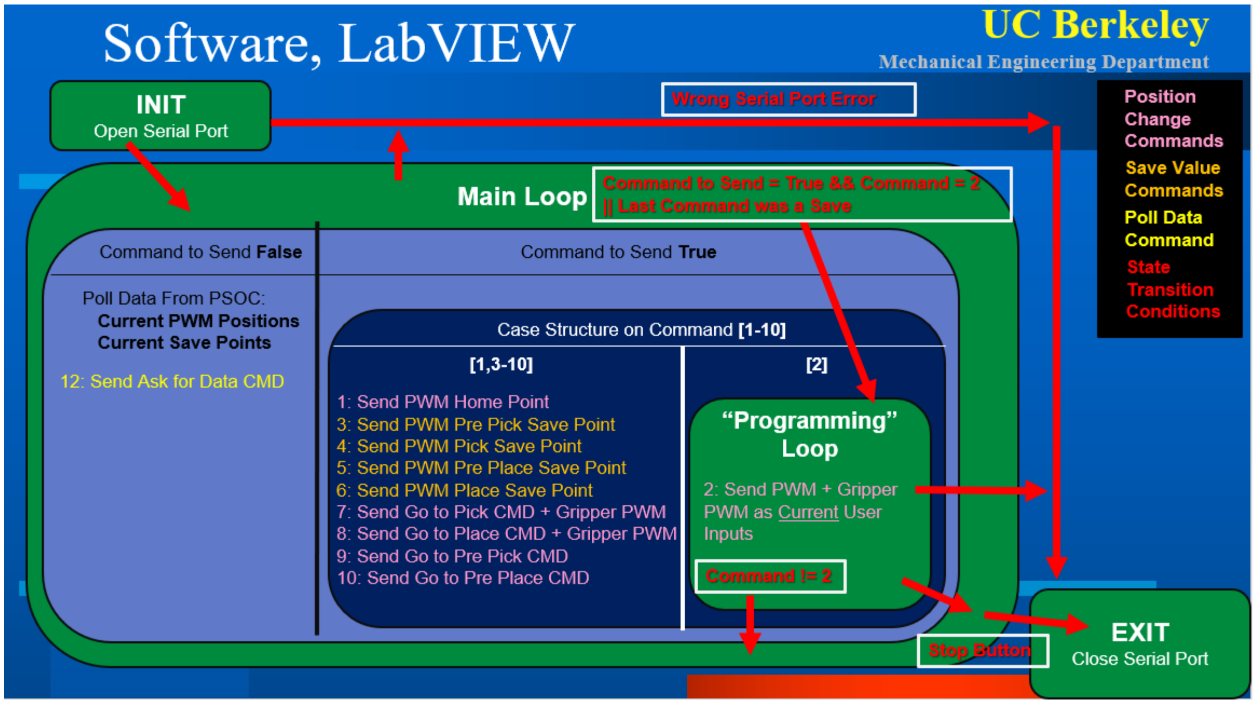

Software, LabVIEW
UC Berkeley
Mechanical Engineering Department
INIT
Open Serial Port
Wrong Serial Port Error
Main Loop
Command to Send = True && Command = 2
|| Last Command was a Save
Position Change Commands
Save Value Commands
Poll Data Command
State Transition Conditions
Command to Send False
Command to Send True
Poll Data From PSOC:
Current PWM Positions
Current Save Points
12: Send Ask for Data CMD
Case Structure on Command [1-10]
[1,3-10]
[2]
1: Send PWM Home Point
3: Send PWM Pre Pick Save Point
4: Send PWM Pick Save Point
5: Send PWM Pre Place Save Point
6: Send PWM Place Save Point
7: Send Go to Pick CMD + Gripper PWM
8: Send Go to Place CMD + Gripper PWM
9: Send Go to Pre Pick CMD
10: Send Go to Pre Place CMD
"Programming" Loop
2: Send PWM + Gripper PWM as Current User Inputs
Command != 2
Stop Button
EXIT
Close Serial Port

Figure 10: LabVIEW State Transition Diagram

There are four states represented by the green blocks, and two case structures represented by the blue blocks. The program flows as follows:

- INIT: Opens the serial port and sets default values.
- Main Loop: This state either sends out a command, or, if there is no command, polls the PSOC for data using command 12. The send command decisions are based on buttons on the GUI. For example, when "Go To Home Point" is pressed, the main loop executes command 1 and sends the PWM target values for the home point.
- Programming loop: This state is entered when the user presses the "Enter Programing Mode" button and holds the program in a timed loop. This loop continuously sends the Current PWM values to the PSOC over serial.
- EXIT: This state closes the serial port and shuts down the program if the stop button is pushed or if there is a serial error.

Challenges

The most difficult part was re-entering the programming mode after the user executes a save command. This required the program to exit the Programming Loop, run though the Main Loop once to send the save command, run though the main loop again to poll the PSOC for data to verify that the save value was received, then re-enter the programming loop. This was done by recording the last command and entering the programming loop if the last command was a save and the PSOC had been polled for data. Another challenge was building front panel buttons to work with our original "debug" LabVIEW VI. For all the GUI buttons, we used a case structure to reset the button, set the send command flag true, and change the command to the appropriate value. An example is shown below for the "Save as Place Point" button.

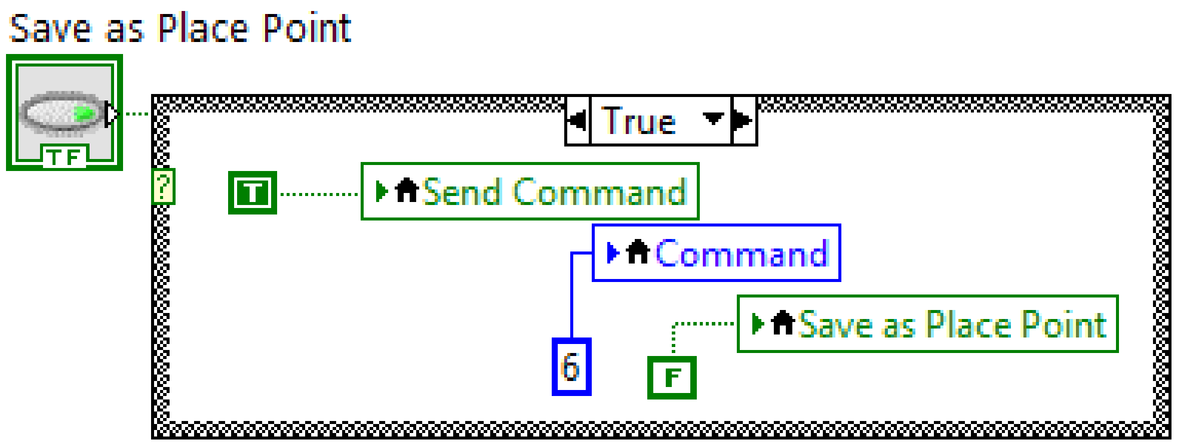


Figure 11: An Example of How Our VI Handles Button Presses

Hardware

As expected, we did have a bit of trouble with the gripper: it stalled out sometimes and its ability to hold an object was unreliable. For sake of demonstration, we had the gripper pick and place a tissue, which has an easy-to-hold rough surface with low weight and a relatable object for the current global pandemic. We also had issues with the quality of the 3D printed parts, but we were able to modify the parts after printing to satisfy our needs. With the tolerance of the 3D printer, some of the parts did not come out perfectly, resulting in ‘jerky’ motion that took a lot of mechanical tweaking to mitigate.

We also did not realize going into the process that some of the parts were too large for our simple 3D printer we have on-hand, so we had to redesign some of the parts so they could be printed on a smaller printing platform. We also ran into other snags with the printer which caused setbacks, like running out of filament after printing the gripper, and a clogged nozzle that needed replacing.

And of course, acquiring the right screws, nuts, wires, and bolts was quite a challenge given the current global pandemic with enormous wait lines and inventory shortages at every hardware store. Even though this was not a hardware project, we believed having an actual robotic arm was important to demonstrate our work, and we were lucky enough to have access to our own 3D printer, oscilloscope, power supply, and more.

Summary and Reflections: What We Would Do Differently

If we were to redo this project, we would implement pressure sensing in order to prevent the gripper from crushing fragile objects. We would also redesign the gripper to be bigger and use a gripper motor with higher holding torque.

We would also use a higher quality 3D printer with better tolerances and a larger platform to ensure we would not need to redesign any parts. This would cause the arm to run smoother and put less strain on the motors.

Regarding the PSOC and LabVIEW code, the majority of it was created as the project evolved. This resulted in a system with a lot of unnecessary commands. If given another opportunity we would simplify the relationship between the PSOC and LabVIEW. One option would be to simplify the PSOC to only accept 3 commands: change target PWM value, poll data, change current PWM value. We would then use LabVIEW to record the save positions and the PSOC solely for speed control. Another option would be to record all save positions on the PSOC and include the automation command. This would allow the PSOC to run the entire code with just a serial terminal or with LabVIEW using the GUI buttons as direct commands. Both of these options would reduce the interdependence between LabVIEW and PSOC, allowing the programming tasks to be developed independently, and making it easier to interface part of the code with another system.

If we were to continue this project, we would focus on improving the “resolution” of the automation trajectory by adding additional steps to our automation process. For example, we could use a program like MATLAB to generate an optimum trajectory which could be converted to a series of points. LabVIEW could then be used to send these points as PWM targets to the PSOC which could track the trajectory using speed control.

Appendix and Attachments

- Presentation: Controls Does Micro.pptx
  - Includes videos and diagrams embedded within
- User Interface: Controls_Does_Micro_VI.zip
  - The top VI is Controls-Does-Micro-Pick-And-Place.vi
- PSOC Code: Workspace_Controls_Does_Micro.zip
  - File location is /ME135_325/ME135_235.cywrk
  - The project to flash is CommandInterpreter_Copy_01