*Ryan Thally, Ifalade Kenneth, Dalton Walz*

*East Tennessee State University*

*Final ProjecT Documentation*

ENTC 4517 – Fall 2018

# Overview

In this project, we are attempting to create an automated process such that two robot arms play each other in a simplified version of connect 4. Our process with use a divided control scheme, subdividing responsibilities to their respective devices. A finite state machine will be used in leu of direct control to designate the process each component is to perform. This implementation will be fully automated, requiring no human intervention for game play and resetting of the game pieces.

# Lab materials, Circuit(s), Functions, Programs

## Supplies

To complete this project, we needed to implement the following supplies:

* 2 Denso 6 Axis Robot Arms
* Siemens S7-1200 PLC
* NI LabView with Vision Builder
* Conveyer Belt
* Optical Isolated Relay Board
* Optical Contactless Presence Sensor
* Connect 4 Game Board and Pieces
* Various length wire for communication

On top of these components, various 3D printed parts were designed. These components include the following:

* End effector tooling for each robot
* Panels and guides for conveyer belt

Examples of these components can be seen in Appendices X, X, X, X below.

## Work Cell

The workspace was arranged as shown in Appendix X below. Due to the size and location of the conveyor belt motor, the cages had to be offset to place the motor housing outside of the path of Robot 2. This offset makes it difficult for each robot to efficiently reach the extremities of the game board. To remedy this, we created an extension arm for robot 2. This arm provided an extra 6 inches of reach to the robot, but required extra coordination to prevent a collision with the cage. This configuration allows both robots to reach the gameboard as well as the conveyor belt without exceeding their operational limits.

The placement of the conveyer belt also provides robot 1 with easy access to the end of the belt, where the pieces are received for sorting. In this area, the camera used by the vision system is suspended above on an arm. In this position, the camera itself is outside the working range of the robot, but the robot will pass into the camera’s view momentarily while picking up a part. This momentary flaw is an unnecessary evil due to the placement of the camera being ideal for properly visualizing and tracking the parts.

In the process of sorting the parts, a series of 3D printed guides were created in order to help orient the game pieces to be properly picked up. The first set are a set of panels that aim to prevent stacking of the parts by creating a lot one part high above the belt. The conveyer then passes each part through a series of gates to align each part into the final slot that aligns the part for the pickup point.

## Host Game Controller Overview

In our project, the primary controller responsible for running game play as well as directing the plc/robots is a program running in Python on a separate computer. Python was chosen as the language for this project for its PLC library support as well as it being a great learning opportunity.

Our Connect4 program is made up of several classes in order to effective isolate and configure separate game instances. Our launch point for the program is the connect4Driver.py file which first prompts the user the enter the number of matches he/she would like played. Once the number of matches is known, the system will iterate though each specified match, creating a new game instance each time. When each game instance is initialized, the same two player objects are passed in, allowing for an overall win/loss percentage to be tracked.

When the program first begins, the user is prompted to enter the number of games he/she would like to play. Once this value is obtained, the robots will perform a homing routine to allow the game board to reset. Once complete, the robots will begin performing the game routine either with manually enter positions (human) or with calculated positions (robot).

In its most basic mode, the game takes column input from the user and the robots will perform the action. In this mode, each robot is simply an extension of the human operators. After each move is completed, the game controller will search the game board for a win. In the event a win occurs, the last moved player’s score is incremented and the game board is cleaned up.

In the game’s second mode, a basic AI will calculate each robot’s move using a reclusive minimax search algorithm. In this algorithm, the player will simulate a specified depth worth of moves by each player to determine the best possible option. Each player is isolated by the program, so each move performed by the opponent is unknown and must be calculated. This level of isolation forces each opponent to play moves with the same knowledge as the human player with the exception of calculated each possible forward move.

After the completion of a game, the host controller will trigger an automated cleanup routine. In this routine, the robots will dispense and sort the game pieces into their respective colors. In order to orient the part to be picked up, the conveyor will alternate directions, forcing stacked parts to be laid flat.

After a preset number of direction changes, the conveyer will run in the forward direction until a part is detected by the vision system. Occasionally, the conveyer will alternate directions in order to prevent any parts from clumping together. Once robot 1 is ready to receive a part, it will signal to the PLC using the lockout line to continue running the belt in the forwarded direction until the piece has been retrieved. Once retrieved, the belt will perform a direction change to shuffle the parts around. This process will continue until every part is removed, or the error timeout is triggered.

Finally, the program will set the runMode to the transfer routine, reversing the belt. The robots will coordinate with each other to transfer each gray piece, which is discussed in further detail below. Once the transfer is complete, the conveyer will stop and a new game object will be created if there are remaining games.

The program files used for the game host controller are referenced in the Appendix below.

## Robot Role Overview

To play the game, the two robots will perform 3 general tasks – sorting, transferring, and playing. To complete each of these procedures, we equipped each robot with custom designed end effector tooling to properly hold a single game piece. These tools were designed in Fusion 360 and then 3D printed. After a few iterations, we developed tooling that was not only effective at picking up each piece, but also at dropping it into a slot on the game board.

When the program first starts, an initialization procedure must first be completed that sorts the pieces into two respective stacks – red and grey. This is completed using the same sorting procedure used at the end of gameplay. Robot 2 will first complete a procedure that dispenses the game pieces from the board and onto the conveyer. The conveyer will then complete the assistance routine mentioned above that aims to lay each piece in a single layer. The pieces will then be forced into the vison detection area, where Robot 1 will them pick up the next piece off the belt and place it into its respective stack. This process will continue until all pieces are accounted for or an error timeout occurs, in which the operator will be required to resolve the error before continuing.

With the pieces stacked, the robots will then move into position to transfer all the grey pieces across the belt so robot 2 can obtain its game pieces. In this transfer procedure, robot 1 will obtain the next grey piece and wait until robot 2 signifies it is ready to accept the piece. This signaling procedure uses a series of inter-robot communication wires that will be discussed in further detail in the communications section below. Once robot 1 receives the ‘go’ signal, it will drop the piece onto the belt and send a confirmation signal to robot 2. Robot 2 will then wait until the optical presence sensor detects the part, then closing the gripper after a hardcoded delay of roughly 800ms has passed. This procedure is also protected against error by using timeout conditions. In the event a timeout condition is reached, the operator will need to resolve the error before gameplay can continue.

In the final task, the robot is able to play the game by transferring a piece off the stack and into one of four columns in the game board. Each robot’s turn is determined by the host game program discussed below. The robots will continue to remove pieces off the stack until the stack is empty or the host game controller calls for a different run state.

In each of these tasks, an operator override was implemented in the even a non-recoverable fault occurred. In these events, the process has been interrupted in such a way that the system cannot decide on how to proceed. When the operator error is triggered, the robots will return to the home position and await further commands. These events will be noted on the teach pendent of the robot at which the operator can manually repair the issue that caused the event and manually override any common issues. In the unlikely situation and manual repair will not remedy the situation, the operator can reload all game pieces into the board and perform a manual sort routine to attempt to restore the operation to a playable state. To prevent any unlikely motion, the robots set their lockout lines to prevent any change in run states.

In addition to the program used to control the robot’s motion for each run state, we created a custom panel in order to provide quick access to common IO and global variables. This panel provides values for the current read runMode bits, detected color, and game pieces remaining. The panel also provides status and control over the lockout, robotXState, and gripper IO ports. This operator panel is especially useful in debugging when values are unexpectedly changed and will alter the program flow.

The program and panel files used for each robot are referenced in the Appendix below

## PLC Role Overview

In our implementation of the Connect 4 game, the current hardware prevents the host game controller from directly talking with the robot controllers. In order to remedy this, we used the S7-1200 PLC to act as a protocol converter. The specifics to which this operates will be discussed in the communications section below. The PLC is configured with logic in order to latch each robot’s state outputs until the individual robot signals that it is ready for a new state by setting its lockout line low. When the line goes low, the plc will copy the memory bits to the set output ports.

On top of handing the protocol conversion for the robots, the plc also takes commands form the host game controller to drive the conveyer belt. In this scenario, the PLC does not use any ladder logic to control the outputs, instead relying on the host game controller to set/reset the appropriate memory bits. This provides the developer the ability to easily update the runtime code using simple python functions.

The program file used for the PLC is referenced in the Appendix below

## Vision Overview

In the sorting phase of the process, each piece must be identified in order for the robot to properly divide them for each player. To do this, we assembled a camera above the conveyer, watching for the color of the piece to appear at the end of the belt. Using the Vision Builder plugin for LabView, we trained a Piece Color Identificatior machine learning solver with sample images of each of three categories – Empty, Red, and Grey. The empty category simply contains samples of the conveyer belt and surrounding features. This training set was created in order for the system to predict when no pieces are present. The second and third training sets, red and grey, contain sample images of each game piece. In order to increase the accuracy of our color identifier, we provided each classification set with at least 5 sample images.

Once the Vision Builder identified a color, the output is the name of the classifier category in which the sample is best associated with. In order to convert this to an integer (0, 1, and 2), we created an array contains each of the three categories at their respective indices. A lookup is performed in the array to find the index at which the category string appears. This index value is then sent over the network to a global integer variable on the robot. The robot will then perform its task associated with the value of the variable.

The LabView file used for the part identification is referenced in the Appendix below.

## Communication Overview

When designing the primary method of control for our project, we found that compartmentalizing each function to its appropriate location to be best. In other words, we only wanted robots to worry about motion control, the plc conveyer and state processing, and the host controller the game. This level of compartmentalization makes it much harder to break any working function once it is verified working. For example, whenever the robot is called to perform a pick and place routing, it will always perform that same routine regardless of what code changes on the host controller. This tiered approach

In the above sections, we saw how the PLC acts as a protocol converter to determine which run state the robot will be set to perform. This process begins in the host game controller, a python program which oversees all actions appropriate to gameplay. To do this, we first needed to install the Step7 library, which provides commands that allow us to directly communicate to the PLC using the S7 protocol. Since the Step7 library is just a generic set of platform specific instructions, we also utilized the Snap7 python library and S71200 helper code. The Snap7 library allows us to utilize the python language to send commands using the platform specific machine code in the Step7 package. The S71200 helper code is just a set of functions provided by Simply Automationized to simplify the read/write instructions to match the Siemens syntax.

With these libraries installed, we were able to develop functions for controlling the conveyer belt. These functions were just simple writes to memory addresses mapped to the output ports. For example, the code plc.writeMem(‘QX0.0’, 1) would be used to set the output of port 0.0 to 1. Functions moveBelt(direction) and stopBelt() were created using code similar to the example syntax to control the two output pins needed to set the direction of the motor.

The PLC also served a second function in our procedure, controlling the run state of the two robots. To accomplish this, we connected 3 control wires from the PLC to each robot controller via optically isolated relays. In additions, we also connected a lockout wire from each robot controller to the PLC inputs. In my state control protocol, the host game controller set the memory byte MB8 with the values for each of the output pins. This memory byte is split into two, 4 bit sections, one for each robot’s control states. These pins are set to a value of 0 through 7 providing each robot with 8 run states. These states can be updated on the fly by the host controller and remain cached in the PLC memory until the robot lowers its lockout line. Once the lockout line low, the PLC performs a series of Boolean logic to set the ready robot(s) output run state pins. Once the output pins are updated, it is then up to the robot to read the value of these pins and perform the designated action. The is one instance in the program, during the sorting task, in which robot 1 will use the lockout wire to signify to the host controller to prevent the conveyer form reversing. In this situation, the robot has been specifically programed to ignore the value determined by the run state lines.

In addition to these run state lines, the robot controllers themselves have a pair of wires to allow each robot to send a Boolean ready signal to each other. This feature is most notably seen during the transfer task, in which the robots need to coordinate the dropping and pickup of a game piece on the conveyer belt. In this procedure, the robot 1 will wait until robot 2 signifies it is ready to receive a part by setting its line high. Robot 1 will then drop the part and set its line high confirming the request. Once this occurs, robot 2 will wait a designated time for the part to traverse the conveyer belt and arrive in the gripper and thus pick the part and place it on the stack. This very simple signal juggle provides a very quick way of properly timing what would otherwise be a very tricky maneuver.

Pin diagrams for each robot, PLC, and conveyer are located in the Appendix below.

# Theory and Results

Overall, our implementation of the Connect 4 process was a success. In this implementation, the python game controller successfully connected to the PLC and set the control states. With proper tuning, the robots successfully read this value and performed the appropriate actions. With incremental tuning, the robots were able to drop the game piece into the designated column in the connect 4 game board. When the game was completed, the pieces were successfully dispensed onto the conveyer belt, and oriented for sorting. With the aid of a well-trained vision system, the colors were properly identified and sorted into their respective stacks. Finally, each grey piece was then successfully transferred to the other robot using inter-robot communication and tuned response times.

# Instructional Capture

The success of this project did not come without any learning experiences. This project pushed the envelope of the capabilities of the hardware provided in the lab. By far our most profound learning experience with the project is in how to adapt non-ideal hardware to complete the task. In an ideal world, we would be able to directly communicate via the network with the robot either using the game controller or the PLC. Limitations with our RC8 controller hardware prevented us from using this implementation, and is why we used the finite state machine to perform our actions. Direct controller communication would have allowed each robot to share global variables, improving the overall runtime of the process.

A second issue we encountered was the placement of the conveyer within the work cell. Our provided conveyer has the drive motor sticking up off of one end. This placement severely limits the access by the robot. To counteract this, we had to offset the robot cages by roughly 1.5 ft such that robot 2 could reach behind in order to move around the motor. This solution lead to another issue – the reachability of the connect 4 game board.

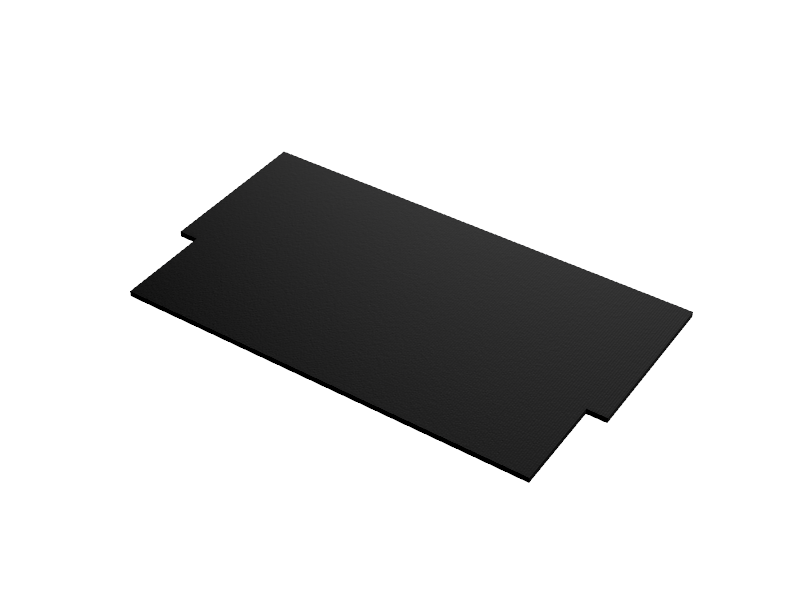
In our implementation, we wanted each robot to play its own turn, mimicking two human players. This meant that the game board had to sit in an area each robot could reach. Due to the offset in the robot cages, we ran into issues in which the robots could not reach the extremities of the game board, and thus not place a game piece in the column furthest away. To remedy this, we offset the game board toward robot 1 and created a 6-inch extension for robot 2. This extension resolved the reachability issue, but created a new movement issue.

With the extra length of a 6-inch extension, robot 2 was very prone to colliding with the cage when moving between the home position and the game positions. To remedy this, a simple passthrough position was created to coax the robot to interpolate its motion through this point when moving between home and conveyer positions. In the event the robot would not interpolate though this position, we implemented compliance parameters on the joints to trigger an error. These parameters successfully prevented damage on multiple occasions in which control states were changed abnormally. This comes with the side effect of slowing the robot motion, but it was not an issue as we do not want the robot moving at its maximum speed through these complex and risky motion path anyways.

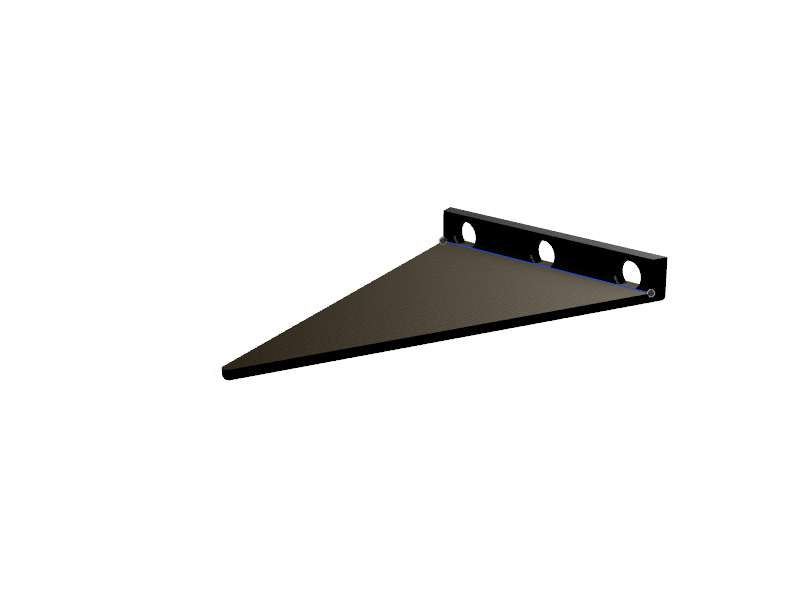
# Appendix:

### 3D Printed Conveyer Guides

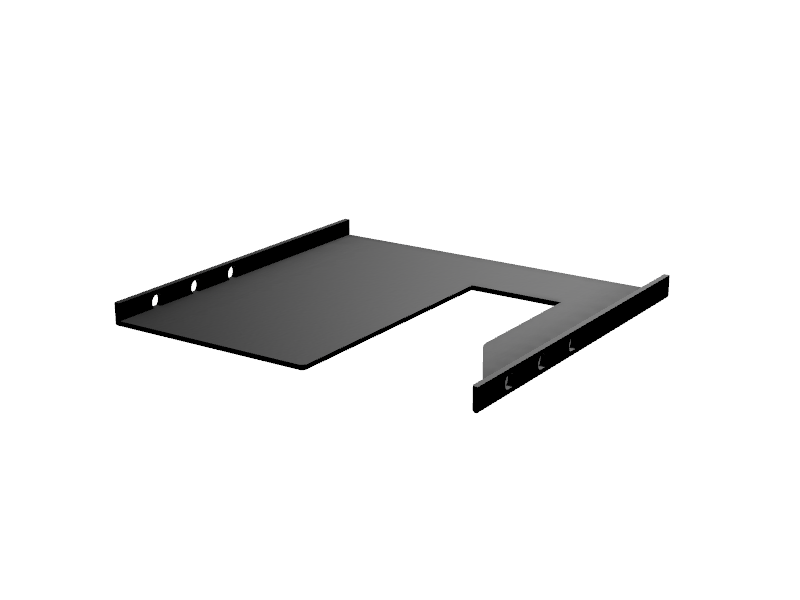
#### Drop Chute



#### Alignment Guide

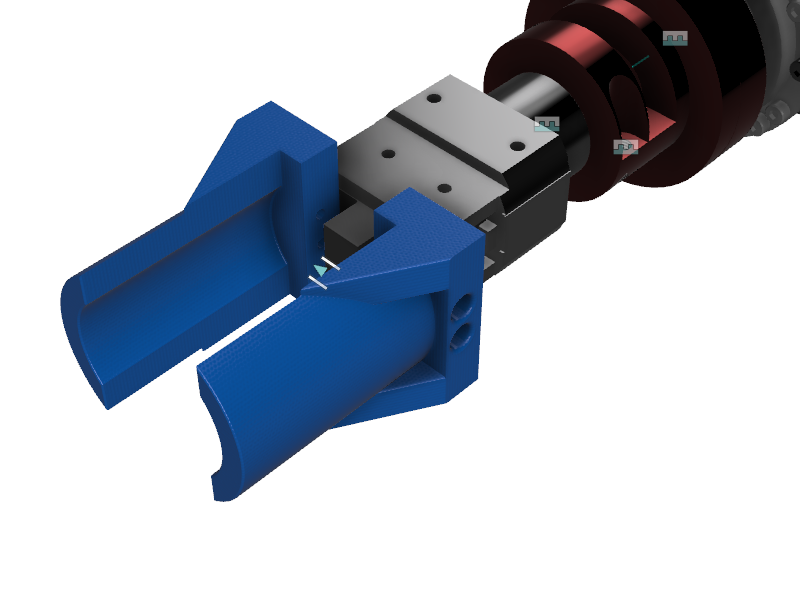


#### End Stop/Channel

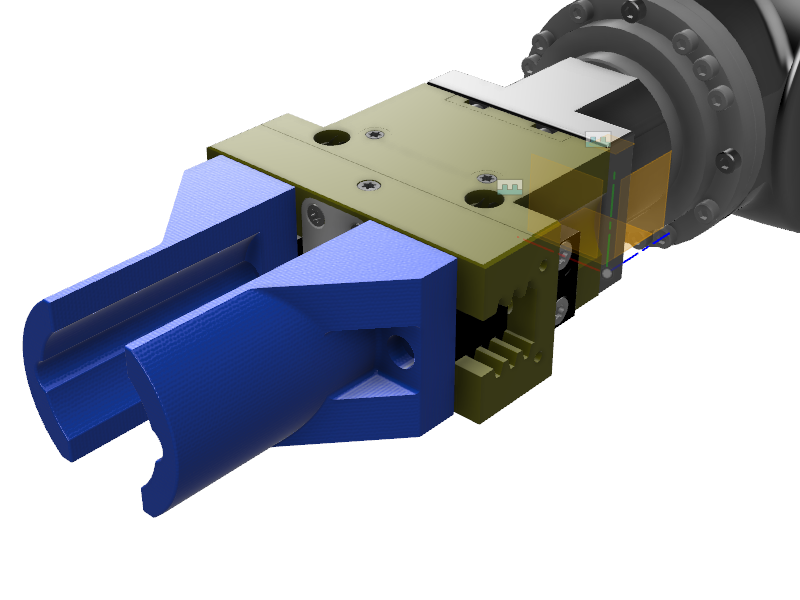


### Robot Tooling

#### Robot 1 End Effector

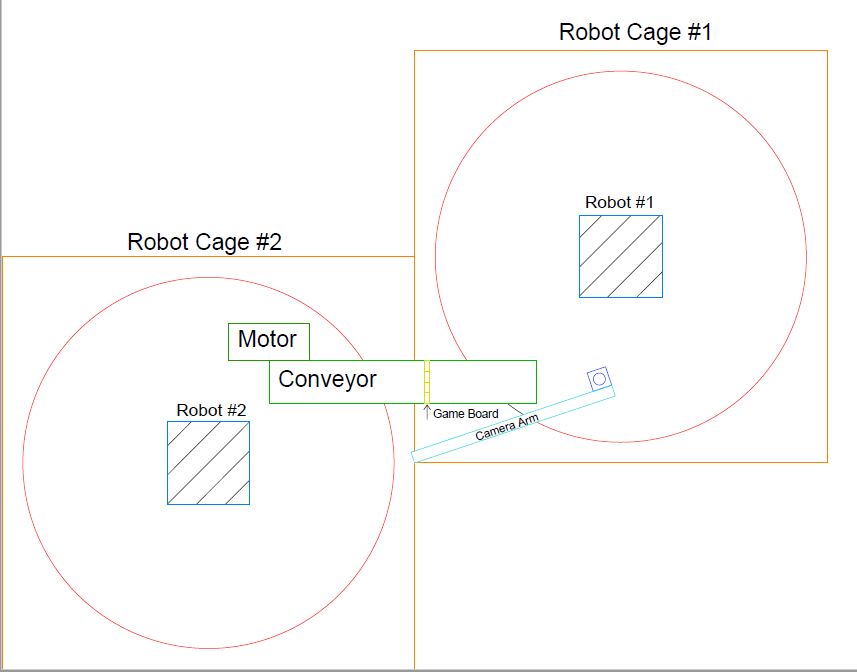


#### Robot 2 End Effector



#### Robot 2 Extension

### Work Cell Layout



### Host Game Controller Code

The Host Game Controller code can be found in the following files:

* connect4Driver.py
* robotCommander.py
* S71200.py
* game.py
* player.py

### Robot Code

The Robot Code can be found in the following files:

* Robot1GameStateController.pcs
* Robot2GameStateController.pcs
* Robot1Pnl.pns
* Robot2Pnl.pns

### PLC Code

The Project file for the PLC can be found in the FinalPLC.ap14 file

### Vision Code

The LabView file for acquiring the color of each part using the Vision Builder plugin can be found in the Connect4.vi file.

### Pin Tables and Wiring Diagrams

#### Robot 1 IO Robot 2 IO

|  |  |  |
| --- | --- | --- |
| Purpose | Robot IO | Mini IO Port |
| RunMode bit 0 | 10 | 21 |
| RunMode bit 1 | 11 | 22 |
| RunMode bit 2 | 12 | 23 |
| Gripper | 26 | 55 |
| Robot1Status | 27 | 56 |
| Robot2Status | 13 | 24 |

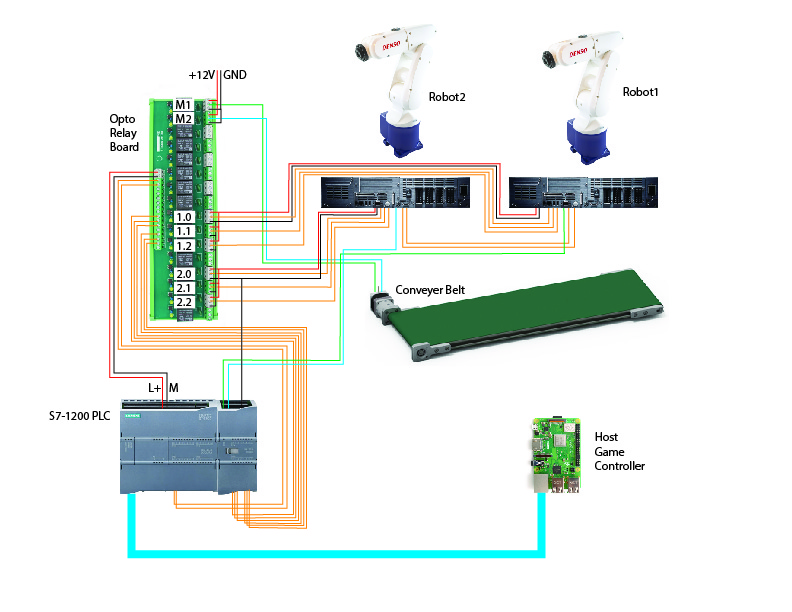
|  |  |  |
| --- | --- | --- |
| Purpose | Robot IO | Mini IO Port |
| RunMode bit 0 | 10 | 21 |
| RunMode bit 1 | 11 | 22 |
| RunMode bit 2 | 12 | 23 |
| Gripper | 26 | 55 |
| Robot1Status | 13 | 24 |
| Robot2Status | 27 | 56 |

#### PLC IO Optical Isolated Relay Board

|  |  |
| --- | --- |
| Purpose | PLC IO Port |
| Robot 1 RunMode bit 0 | QX8.0 |
| Robot 1 RunMode bit 1 | QX8.1 |
| Robot 1 RunMode bit 2 | QX8.2 |
| Robot 2 RunMode bit 0 | QX8.4 |
| Robot 2 RunMode bit 1 | QX8.5 |
| Robot 2 RunMode bit 2 | QX8.6 |
| Motor Rly 1 | QX0.0 |
| Motor Rly 2 | QX0.1 |
| Robot 1 Lockout | IX8.0 |
| Robot 2 Lockout | IX8.1 |

|  |  |
| --- | --- |
| Purpose | Relay IO Port |
| Robot 1 RunMode bit 0 | 9 |
| Robot 1 RunMode bit 1 | 10 |
| Robot 1 RunMode bit 2 | 11 |
| Robot 2 RunMode bit 0 | 13 |
| Robot 2 RunMode bit 1 | 14 |
| Robot 2 RunMode bit 2 | 15 |
| Motor Rly 1 | 1 |
| Motor Rly 2 | 2 |

#### Wiring Diagram



### Position/Tool Data

#### Robot 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | X | Y | Z | RX | RY | RZ | FIGURE |
| P1 | 215.0776 | -0.18577 | 337.3923 | 180 | -1.58E-13 | -179.468 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P4 | 286.5817 | -490.61 | 404.2715 | -16.7056 | 82.25773 | -106.789 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P5 | 250.7015 | -490.467 | 405.662 | -16.7056 | 82.25773 | -106.789 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P6 | 214.8757 | -490.325 | 407.0504 | -16.7056 | 82.25773 | -106.789 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P7 | 178.071 | -490.179 | 408.4767 | -16.7056 | 82.25773 | -106.789 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P10 | 300 | 200 | 5.174495 | 179.182 | 0.887159 | 179.3367 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P11 | 300 | 100 | 5.174495 | 179.182 | 0.887159 | 179.3367 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P15 | 267.1936 | -168.84 | 118.3102 | 178.8075 | -1.55E-03 | 91.777 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P17 | 205.5083 | -326.852 | 120.964 | 180 | -1.74E-03 | -179.468 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
|  |  |  |  |  |  |  |  |
| Tool1 | 0 | 0 | 165 | 0 | 0 | 0 |  |

#### Robot 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | X | Y | Z | RX | RY | RZ | FIGURE |
| P1 | 208.6551 | -0.01 | 240.953 | 180 | 0.00E+00 | 0 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P4 | -152.109 | 599.9937 | 401.1378 | -27.0688 | -87.0705 | -62.1522 | 1 - Lefty | Above | Flip | J6Single | J4Single | J1Single |
| P5 | -187.879 | 598.5059 | 399.782 | -15.7699 | -85.0896 | -73.4843 | 1 - Lefty | Above | Flip | J6Single | J4Single | J1Single |
| P6 | -221.608 | 597.6423 | 402.6927 | -27.0688 | -87.0705 | -62.1522 | 1 - Lefty | Above | Flip | J6Single | J4Single | J1Single |
| P7 | -257.834 | 600.1624 | 399.9953 | -27.0314 | -87.0761 | -62.1877 | 1 - Lefty | Above | Flip | J6Single | J4Single | J1Single |
| P10 | 349.5469 | 1.29E-02 | 29.46387 | 179.9975 | 0.144748 | -0.66834 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P11 | -103.049 | 3.68E+02 | 365.8247 | -151.339 | -46.3233 | 83.14191 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
| P15 | -147.305 | 584.808 | 231.5075 | -179.832 | -5.92E+01 | 90.48125 | 1 - Lefty | Above | Flip | J6Single | J4Single | J1Single |
| P16 | -128.904 | 585.0089 | 231.4799 | -179.832 | -59.1661 | 90.48125 | 1 - Lefty | Above | Flip | J6Single | J4Single | J1Single |
| P17 | -159.124 | 322.0418 | 151.4186 | 179.166 | -6.47E-01 | 86.73625 | 5 - Lefty | Above | NonFlip | J6Single | J4Single | J1Single |
|  |  |  |  |  |  |  |  |
| Tool1 | 0 | 0 | 254 | 0 | 0 | 85 |  |

### ­­Local/Global Variables

#### Robot 1

|  |  |  |
| --- | --- | --- |
| I0 | Integer | Detected Color |
| I1 | Integer | Pieces Remaining |
| greyCount | Integer | Number of picked grey pieces |
| runMode | Byte | runMode set by IO bits |

#### Robot 2

|  |  |  |
| --- | --- | --- |
| I1 | Integer | Pieces Remaining |
| runMode | Byte | runMode set by IO bits |

#### PLC

|  |  |  |
| --- | --- | --- |
| MB8 | Byte | Memory byte for robot state |

### Compliance

#### Robot 1 Robot 2

|  |  |
| --- | --- |
| J1 | 31 |
| J2 | 50 |
| J3 | 45 |
| J4 | 30 |
| J5 | 45 |
| J6 | 0 |

|  |  |
| --- | --- |
| J1 | 31 |
| J2 | 50 |
| J3 | 45 |
| J4 | 30 |
| J5 | 45 |
| J6 | 0 |