ELE4307

Real Time Systems

Assignment 1

Control of Pick and Place Machine for SMT Assembly

Student Name: Kate Bowater

Student Number: U1019160

Contents

[Introduction 3](#_Toc168344806)

[System Design 3](#_Toc168344807)

[Part A 3](#_Toc168344808)

[Part B 3](#_Toc168344809)

[Testing Results 6](#_Toc168344810)

[Part A 6](#_Toc168344811)

[Part B 7](#_Toc168344812)

[Appendix 8](#_Toc168344813)

# Introduction

This report details the design of a program written in C to control a pick and place machine for assembly of Surface Mount Technology (SMT) based Printed Circuit Boards (PCBs). The machine is designed to read a centroid file for component details and placement information. It is designed for both manual and automatic modes and is fully tested for functionality using a simulator by a POSIX compliant program. The program is deemed to be fully functional for both manual and autonomous modes although some synchronization issues occurred during testing that were unrepeatable and not attributed to the written code.

# System Design

## Part A

Figure 1 is a Mealy state-based diagram for manual operation. It is assumed that the order of placing components is required to be done in the order specified within the centroid file. It is also assumed that the user is aware of the steps to pick and place the components. No error or blocking of the program will occur for incorrect key presses; the program simply will not do anything unless a valid key is pressed.

If the user mistakenly enters the wrong feeder number, the program will not prevent this occurring but will display a warning that it is not the same feeder as specified in the centroid file. In the WAIT\_1 state, the user has the chance to change the feeder, move to home, or to continue with picking and placing the part in which case the program will continue to pull details from the centroid file for the rotation and placement coordinates. No implementation has been made to prevent the user from any action outside of what is expected for the pick and place machine to function per the requirements.

If the user initiates correction to nozzle rotation or gantry alignment prior to taking the look-up and look-down photos, the movement will simply be zero until the actual error is obtained. A notification will display to the user via the controller to recommend corrective action once errors are calculated, however the program will not prevent the user from placing the parts on the PCB in any instance.

## Part B

Figure 2 is the Mealy state-based diagram for autonomous operation. All three nozzles are used in this mode to reduce the time to pick and place parts. A design choice implemented is that the program will reorder the components in the centroid file ascending by feeder number. It was designed this way in correlation to the nozzle positions to shorten the time of the gantry movement from home position: first it moves to the closest feeder number to pick up the part with the left nozzle, then the next feeder with the centre nozzle, and the part in the furthest feeder will be picked up with the right nozzle. The decision was made against attempting to place the parts based on distance of the x and y coordinates on the PCB as the implementation was deemed to be too complicated.

**Part A State Diagram**

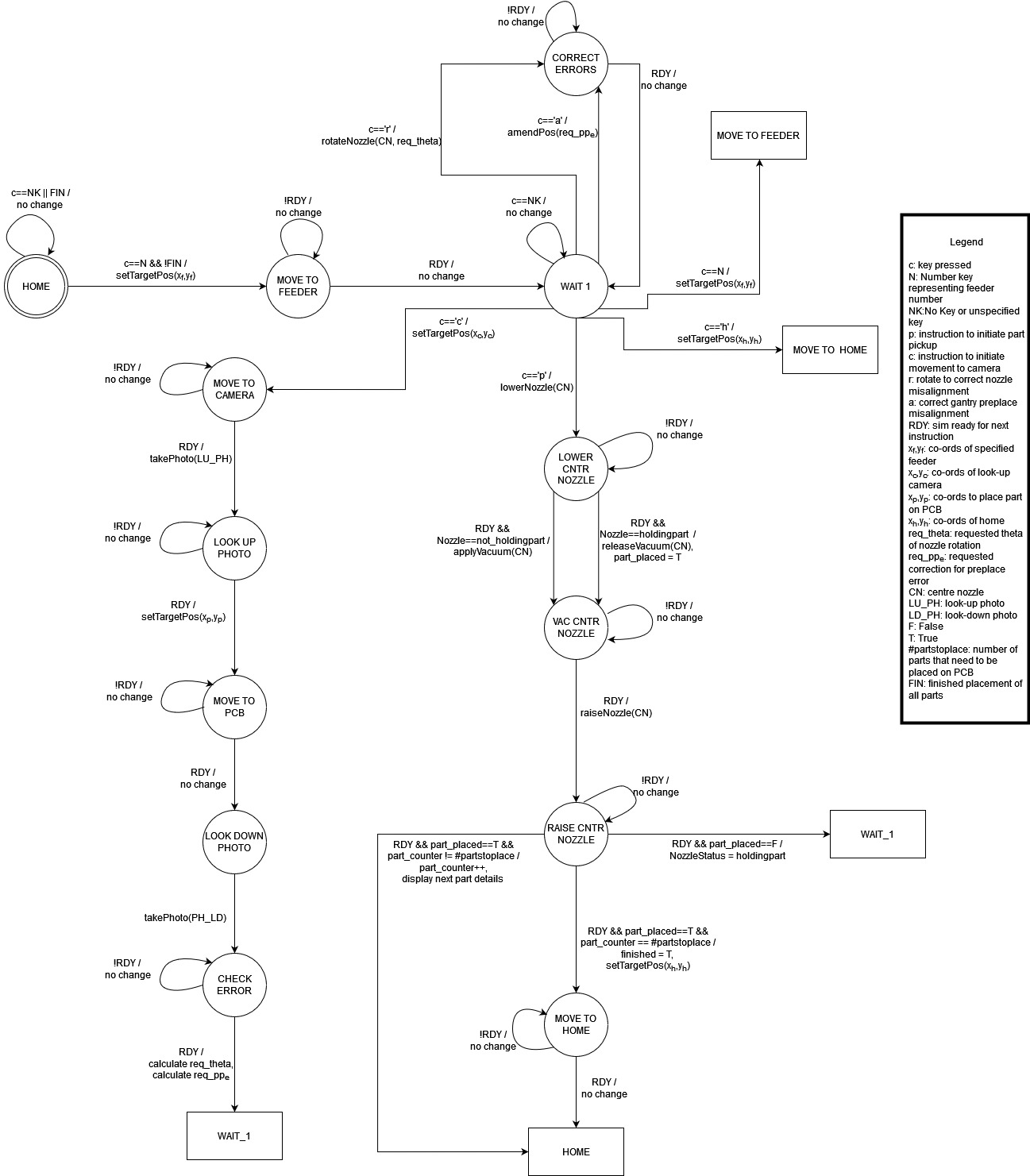


Figure 1: Mealy state-based diagram for manual operation

**Part B State Diagram**

A diagram of a company

Description automatically generated

Figure 2: Mealy state-based diagram for autonomous operation

# Testing Results

## Part A

Debugging was performed throughout development of the program. Once it was determined that the program could run through a centroid file without errors, formal testing was achieved by checking against Table 1 and rerunning the program after correction of any issues. The sequence of steps followed was as though the user would be using the pick and place machine per requirements.

Table 1: Testing checklist for manual mode

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case #** | **Description** | **Pass/Fail** | **Notes** |
| 1 | On numbered key press, gantry moves to correct feeder coordinate | Pass | No issues |
| 2 | Displays warning if incorrect feeder number is pressed | Pass | No issues |
| 3 | On ‘p’ key press, centre nozzle lowers, applies vacuum and raises | Pass | All sub-actions run consecutively without issues |
| 4 | On ‘c’ key press, gantry moves to camera and takes look-up photo. Then moves to PCB and takes look-down photo | Pass | All sub-actions run consecutively without issues. Alignment errors are displayed to the user |
| 5 | Error correction is calculated correctly | Pass | No issues |
| 6 | On ‘r’ key press, the nozzle rotation misalignment is corrected to the required θ as per centroid file | Pass | No issues |
| 7 | On ‘a’ key press, the nozzle rotation misalignment is corrected to the required x, y coordinates as per centroid file | Pass | No issues |
| 8 | On ‘p’ key press, centre nozzle lowers, releases vacuum to place part, then raises | Pass | No issues |
| 9 | After a part is placed, the next component details are displayed | Pass | No issues |
| 10 | On ‘h’ key press, the gantry moves to the home position | Pass | No issues |
| 11 | On state change, simulation time, state name and description are printed to the controller window | Pass | No issues |
| 12 | Notifies the user when all parts have been placed | Pass | No issues |
| 13 | Program will pick and place any number of parts in a centroid file | Pass | Tested from 1-10 components to place |

## Part B

Part B was tested in two separate phases, functionality and performance. Regarding functionality, it was noted during testing that occasionally the simulator would not perform certain actions, such as amending the misalignment despite the errors having been detected, or placing parts on the PCB. The errors seem inconsistent; after closing the program and running it again the simulator will perform the required actions correctly.

Due to the inconsistencies mentioned in performance, formal testing entailed running the program at least three times through one centroid file to determine whether any problems were due to the written code or due to asychronisation of the controller to the simulator. Errors of this nature are noted in the below if they were present.

Table 2: Functional testing checklist for autonomous mode

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case #** | **Description** | **Pass/Fail** | **Notes** |
| 1 | Components are reordered ascending by feeder number | Pass | No issues |
| 2 | All part details are displayed at the initiation of the program | Pass | Part numbers are out of order but correspond to the part number in the centroid file |
| 3 | The program moves to the correct feeder aligned with the left nozzle | Pass | No issues |
| 4 | The part is successfully picked up by the left nozzle | Pass | No issues |
| 5 | The program moves to the next feeder aligned with the centre nozzle | Pass | No issues |
| 6 | The part is successfully picked up by the centre nozzle | Pass | No issues |
| 7 | The program moves to the next feeder aligned with the right nozzle | Pass | No issues |
| 8 | The part is successfully picked up by the right nozzle | Pass | Nozzle failed to raise. Did not reoccur after restarting program |
| 9 | Once all nozzles have acquired parts, the gantry moves to the look-up camera and takes a look-up photo | Pass | No issues |
| 10 | Gantry moves to the correct PCB coordinates and takes a lookdown photo | Pass | No issues |
| 11 | Nozzle rotation and preplace misalignment are corrected to the required values in the centroid file | Pass | Issue noted with preplace misalignment error not corrected by simulator. Did not reoccur after restarting the program |
| 12 | The program places all parts in their correct x, y coordinates | Pass | Issue noted with nozzles not placing part on PCB. Did not reoccur after restarting program |
| 13 | Program detects the next set of parts for picking and placing | Pass | No issues |
| 14 | If there aren’t enough parts for all the nozzles, then the program moves on to the next step | Pass | No issues |
| 15 | Once all parts are placed, the user is notified and the gantry moves to home position | Pass | No issues |
| 16 | Program will pick and place parts for any odd number of components in the centroid file | Pass | No issues |
| 17 | Program will pick and place parts for any even number of components in the centroid file | Fail | Program fails to change state when 20 parts are in the centroid file and attempts to pick and place the 20th part. No other number of parts has issues. The cause is unknown. |

Regarding performance, three methods of part sorting were tested to determine the quickest time. The component list could be sorted by feeder number, x-coordinate, or y-coordinate. To measure the optimisation of the machine, picking and placing times were recorded in Table 3 by comparing the unsorted pick and place time to the reordered list pick and place time. Each test was run with the same list of components for consistency.

Variances in the instances is attributed to the differences in part coordinates on the PCB based on the centroid file. Sorting by feeder number produced the most consistent results and so is implemented into the final program for submission. Ideally, the part list would initially be sorted by feeder number for picking up parts, then sorted by closest x- and y-coordinates just before placing. This solution could not be implemented before final submission but is recommended for improving optimisation of the machine.

Table 3: Time in seconds to pick and place parts

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **# of parts** | **Unsorted list** | **Sorted by feeder #** | **Sorted by x-coord** | **Sorted by y-coord** |
| 5 | 13.02 | 13.62 |  |  |
| 10 | 24.67 | 23.74 |  |  |
| 15 | 36.43 | 34.76 |  |  |
| 17 | 43.25 |  |  |  |
| 18 | 45.49 |  |  |  |
| 19 | 47.67 |  |  |  |
| 20 |  |  |  |  |
| 30 |  | 71.20 |  |  |

# Appendix

/\*

\*

\* pnpControl.c - the controller for the pick and place machine in manual and autonomous mode

\*

\* Platform: Any POSIX compliant platform

\* Intended for and tested on: Cygwin 64 bit

\*

\* Edited by: Kate Bowater

\* Student Number: U1019160

\* Last edit: 3 Jun 2024

\*

\*/

#include "pnpControl.h"

// state names and numbers

#define HOME 0

#define MOVE\_TO\_FEEDER 1

#define WAIT\_1 2

#define LOWER\_CNTR\_NOZZLE 3 //lowering the centre nozzle

#define VAC\_CNTR\_NOZZLE 4 //applying the vacuum for the centre nozzle

#define RAISE\_CNTR\_NOZZLE 5 //raising the centre nozzle

#define MOVE\_TO\_CAMERA 6

#define LOOK\_UP\_PHOTO 7

#define MOVE\_TO\_PCB 8

#define LOOK\_DOWN\_PHOTO 9

#define CHECK\_ERROR 10

#define CORRECT\_ERRORS 11

#define MOVE\_TO\_HOME 12

#define FIX\_NOZZLE\_ERROR 13

#define FIX\_PREPLACE\_ERROR 14

#define LOWER\_LEFT\_NOZZLE 15

#define VAC\_LEFT\_NOZZLE 16

#define RAISE\_LEFT\_NOZZLE 17

#define LOWER\_RIGHT\_NOZZLE 18

#define VAC\_RIGHT\_NOZZLE 19

#define RAISE\_RIGHT\_NOZZLE 20

#define holdingpart 1

#define not\_holdingpart 0

/\* state\_names of up to 19 characters (the 20th character is a null terminator), only required for display purposes \*/

const char state\_name[21][20] = {"HOME ",

"MOVE TO FEEDER ",

"WAIT 1 ",

"LOWER CNTR NOZZLE ",

"VAC CNTR NOZZLE ",

"RAISE CNTR NOZZLE ",

"MOVE TO CAMERA ",

"LOOK UP PHOTO ",

"MOVE TO PCB ",

"LOOK DOWN PHOTO ",

"CHECK ERROR ",

"CORRECT ERRORS ",

"MOVE TO HOME ",

"FIX NOZZLE ERROR ",

"FIX PREPLACE ERROR ",

"LOWER LEFT NOZZLE ",

"VAC LEFT NOZZLE ",

"RAISE LEFT NOZZLE ",

"LOWER RIGHT NOZZLE ",

"VAC RIGHT NOZZLE ",

"RAISE RIGHT NOZZLE "};

const double TAPE\_FEEDER\_X[NUMBER\_OF\_FEEDERS] = {FDR\_0\_X, FDR\_1\_X, FDR\_2\_X, FDR\_3\_X, FDR\_4\_X, FDR\_5\_X, FDR\_6\_X, FDR\_7\_X, FDR\_8\_X, FDR\_9\_X};

const double TAPE\_FEEDER\_Y[NUMBER\_OF\_FEEDERS] = {FDR\_0\_Y, FDR\_1\_Y, FDR\_2\_Y, FDR\_3\_Y, FDR\_4\_Y, FDR\_5\_Y, FDR\_6\_Y, FDR\_7\_Y, FDR\_8\_Y, FDR\_9\_Y};

const char nozzle\_name[3][10] = {"left", "centre", "right"};

int main()

{

pnpOpen();

int operation\_mode, number\_of\_components\_to\_place, res;

PlacementInfo pi[MAX\_NUMBER\_OF\_COMPONENTS\_TO\_PLACE];

/\*

\* read the centroid file to obtain the operation mode, number of components to place

\* and the placement information for those components

\*/

res = getCentroidFileContents(&operation\_mode, &number\_of\_components\_to\_place, pi);

if (res != CENTROID\_FILE\_PRESENT\_AND\_READ)

{ //throw an error if the centroid file is unreadable or not present

printf("Problem with centroid file, error code %d, press any key to continue\n", res);

getchar();

exit(res);

}

/\* state machine code for manual control mode \*/

if (operation\_mode == MANUAL\_CONTROL)

{

/\* initialization of variables and controller window \*/

int state = HOME, finished = FALSE, part\_counter = 0;

char c, part\_placed, NozzleStatus = not\_holdingpart;

double requested\_theta = 0; //the required angle theta of the nozzle position

double preplace\_diff\_x = 0, preplace\_diff\_y = 0; //difference in required gantry position and actual gantry position for preplacement

printf("Time: %7.2f Initial state: %.15s Operating in manual control mode, there are %d parts to place\n\n", getSimulationTime(), state\_name[HOME], number\_of\_components\_to\_place);

/\* print details of part 0 \*/

printf("Part 0 details:\nDesignation: %s\nFootprint: %s\nValue: %.2f\nx: %.2f\ny: %.2f\ntheta: %.2f\nFeeder: %d\n\n",

pi[0].component\_designation, pi[0].component\_footprint, pi[0].component\_value, pi[0].x\_target, pi[0].y\_target, pi[0].theta\_target, pi[0].feeder);

/\* loop until user quits \*/

while(!isPnPSimulationQuitFlagOn())

{

c = getKey(); //saves the value of the key pressed by the user

switch (state)

{

case HOME:

//gantry in home position, waiting for input by user to initiate movement to feeder

if (finished == FALSE && (c == '0' || c == '1' || c == '2' || c == '3' || c == '4' || c == '5' || c == '6' || c == '7' || c == '8' || c == '9'))

{

//check if user inputs a feeder number that is not next in the centroid file

if ((c - '0') != pi[part\_counter].feeder)

{ /\* the expression (c - '0') obtains the integer value of the number key pressed \*/

printf("Time: %7.2f WARNING The next part is in feeder %d.\n", getSimulationTime(), pi[part\_counter].feeder);

}

setTargetPos(TAPE\_FEEDER\_X[c - '0'], TAPE\_FEEDER\_Y[c - '0']);

state = MOVE\_TO\_FEEDER;

printf("Time: %7.2f New state: %.20s Issued instruction to move to tape feeder %c\n", getSimulationTime(), state\_name[state], c);

}

break;

case MOVE\_TO\_FEEDER:

//waiting for the simulator to complete movement of the gantry

if (isSimulatorReadyForNextInstruction())

{

state = WAIT\_1;

printf("Time: %7.2f New state: %.20s Arrived at feeder, waiting for next instruction\n", getSimulationTime(), state\_name[state]);

}

break;

case WAIT\_1: //waiting for next key press

//'p' for pickup

if((c == 'p') && (NozzleStatus == not\_holdingpart)) //checking if the nozzle is empty

{

lowerNozzle(CENTRE\_NOZZLE);

state = LOWER\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Issued instruction to pick up part. Lowering centre nozzle\n", getSimulationTime(), state\_name[state]);

}

//'p' to place the part that the nozzle is currently holding

else if((c == 'p') && (NozzleStatus == holdingpart))

{

lowerNozzle(CENTRE\_NOZZLE);

state = LOWER\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Issued instruction to place part on PCB. Lowering nozzle\n", getSimulationTime(), state\_name[state]);

}

//'c' for camera, should only go to the camera if the nozzle is holding a part

else if(c == 'c')

{

setTargetPos(LOOKUP\_CAMERA\_X,LOOKUP\_CAMERA\_Y); //the gantry will move to the position above the camera

state = MOVE\_TO\_CAMERA; //after the nozzle picked up a part, send the gantry to the lookup camera

printf("Time: %7.2f New state: %.20s Issued instruction to move to look-up camera\n", getSimulationTime(), state\_name[state]);

}

//'r' for rotate to fix the nozzle misalignment error

else if(c == 'r')

{

rotateNozzle(CENTRE\_NOZZLE, requested\_theta); //rotate the nozzle by the required calculated angle theta

state = CORRECT\_ERRORS;

printf("Time: %7.2f New state: %.20s Correcting part misalignment on nozzle\n", getSimulationTime(), state\_name[state]);

}

//'a' for adjusting the position of the gantry for preplace misalignment error

else if(c == 'a')

{

amendPos(preplace\_diff\_x, preplace\_diff\_y); //corrects the position by the calculated difference x and y

state = CORRECT\_ERRORS;

printf("Time: %7.2f New state: %.20s Correcting preplace misalignment of gantry\n", getSimulationTime(), state\_name[state]);

}

// 'h' for home. This will move the gantry back to its home position

else if(c == 'h')

{

setTargetPos(HOME\_X,HOME\_Y);

state = MOVE\_TO\_HOME;

printf("Time: %7.2f New state: %.20s Moving to home position\n", getSimulationTime(), state\_name[state]);

}

// in case the user pressed the wrong number key and needs to change the feeder

else if (c == '0' || c == '1' || c == '2' || c == '3' || c == '4' || c == '5' || c == '6' || c == '7' || c == '8' || c == '9')

{

//check if user inputs a feeder number that is not next in the centroid file

if ((c - '0') != pi[part\_counter].feeder)

{ /\* the expression (c - '0') obtains the integer value of the number key pressed \*/

printf("Time: %7.2f %19s WARNING The next part is in feeder %d.\n", getSimulationTime()," ", pi[part\_counter].feeder);

}

setTargetPos(TAPE\_FEEDER\_X[c - '0'], TAPE\_FEEDER\_Y[c - '0']);

state = MOVE\_TO\_FEEDER;

printf("Time: %7.2f New state: %.20s Issued instruction to move to tape feeder %c\n", getSimulationTime(), state\_name[state], c);

}

break;

case LOWER\_CNTR\_NOZZLE:

//Need to wait until simulator is ready before moving on to vacuum

if (isSimulatorReadyForNextInstruction())

{

if(NozzleStatus == not\_holdingpart)

{ //vacuum will apply when the nozzle is empty

applyVacuum(CENTRE\_NOZZLE);

state = VAC\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Applying vacuum\n", getSimulationTime(), state\_name[state]);

}

if(NozzleStatus == holdingpart)

{ //vacuum will release the part when the nozzle is holding something

releaseVacuum(CENTRE\_NOZZLE);

part\_placed = TRUE; //counter to indicate the part has been placed

state = VAC\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Releasing vacuum to place part\n", getSimulationTime(), state\_name[state]);

}

}

break;

case VAC\_CNTR\_NOZZLE:

//wait until the vacuum action is finished before raising the nozzle

if (isSimulatorReadyForNextInstruction())

{

raiseNozzle(CENTRE\_NOZZLE);

state = RAISE\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Raising nozzle\n", getSimulationTime(), state\_name[state]);

}

break;

case RAISE\_CNTR\_NOZZLE:

//once nozzle is raised, if a part hasn't just been placed, then it is determined that a part has just been picked up

if (isSimulatorReadyForNextInstruction())

{

if (part\_placed==FALSE)

{

NozzleStatus = holdingpart;

state = WAIT\_1;

printf("Time: %7.2f New state: %.20s Part acquired, ready for next instruction\n", getSimulationTime(), state\_name[state]);

}

//if the vacuum has just released a part, then the part has been placed and the nozzle is free again

if (part\_placed==TRUE)

{

NozzleStatus = not\_holdingpart;

part\_placed = FALSE; //variable to change state actions based on whether a part has just been placed or not

part\_counter++; //increment counter to keep track of the part number in the centroid file that has been placed

if (part\_counter != number\_of\_components\_to\_place)

{ //since there are still components to be placed, go back to Home to cycle again. Display the next set of part details

state = HOME;

printf("Time: %7.2f New state: %.20s Part %d placed on PCB successfully\n\n", getSimulationTime(), state\_name[state], (part\_counter-1));

printf("Part %d details:\nDesignation: %s\nFootprint: %s\nValue: %.2f\nx: %.2f\ny: %.2f\ntheta: %.2f\nFeeder: %d\n\n", part\_counter,

pi[part\_counter].component\_designation, pi[part\_counter].component\_footprint, pi[part\_counter].component\_value, pi[part\_counter].x\_target,

pi[part\_counter].y\_target, pi[part\_counter].theta\_target, pi[part\_counter].feeder);

}

else if(part\_counter == number\_of\_components\_to\_place)

{

finished = TRUE;

setTargetPos(HOME\_X,HOME\_Y);

state = MOVE\_TO\_HOME;

printf("Time: %7.2f New state: %.20s All parts have been placed! Moving to home\n", getSimulationTime(), state\_name[state]);

}

}

}

break;

case MOVE\_TO\_CAMERA:

//waiting for the gantry to move to the camera position before taking look-up photo

if (isSimulatorReadyForNextInstruction())

{

takePhoto(PHOTO\_LOOKUP);

state = LOOK\_UP\_PHOTO;

printf("Time: %7.2f New state: %.20s Arrived at camera. Taking look-up photo of part\n", getSimulationTime(), state\_name[state]);

}

break;

case LOOK\_UP\_PHOTO:

if (isSimulatorReadyForNextInstruction())

{ //once look-up photo is taken, move the gantry to the PCB for part placement

setTargetPos(pi[part\_counter].x\_target, pi[part\_counter].y\_target);

state = MOVE\_TO\_PCB;

printf("Time: %7.2f New state: %.20s Look-up photo acquired. Moving to PCB\n", getSimulationTime(), state\_name[state]);

}

break;

case MOVE\_TO\_PCB:

//once the gantry has finished moving to the PCB, then it is ready to take a look-down photo

if (isSimulatorReadyForNextInstruction())

{

state = LOOK\_DOWN\_PHOTO;

printf("Time: %7.2f New state: %.20s Now at PCB. Taking look-down photo\n", getSimulationTime(), state\_name[state]);

}

break;

case LOOK\_DOWN\_PHOTO:

//take the look-down photo, then move on to check for errors

takePhoto(PHOTO\_LOOKDOWN);

state = CHECK\_ERROR;

printf("Time: %7.2f New state: %.20s Look-down photo acquired. Checking for errors in alignment\n", getSimulationTime(), state\_name[state]);

break;

case CHECK\_ERROR:

//wait until the look-down photo is taken, then calculate errors

if (isSimulatorReadyForNextInstruction())

{

double errortheta = getPickErrorTheta(CENTRE\_NOZZLE); //acquire the part misalignment from the look-up photo

requested\_theta = pi[part\_counter].theta\_target - errortheta; //calculate misalignment of the part on the nozzle

preplace\_diff\_x = pi[part\_counter].x\_target - (pi[part\_counter].x\_target+getPreplaceErrorX()); //calculate the difference between the required x position and the actual x position of the gantry

preplace\_diff\_y = pi[part\_counter].y\_target - (pi[part\_counter].y\_target+getPreplaceErrorY()); //calculate the difference between the required y position and the actual y position of the gantry

state = WAIT\_1; //display the errors to the user so they are aware and then wait for instruction

printf("Time: %7.2f %19s Part misalignment error: %3.2f, preplace misalignment error: x=%3.2f y=%3.2f\n", getSimulationTime()," ", errortheta, getPreplaceErrorX(), getPreplaceErrorY());

printf("Time: %7.2f New state: %.20s Waiting for next instruction. Recommend error correction\n", getSimulationTime(),state\_name[state]);

}

break;

case CORRECT\_ERRORS:

if (isSimulatorReadyForNextInstruction())

{ //once the nozzle or gantry position has been corrected, go back to wait for next instruction

state = WAIT\_1;

printf("Time: %7.2f New state: %.20s Misalignment corrected, ready for next instruction\n", getSimulationTime(), state\_name[state]);

}

break;

case MOVE\_TO\_HOME:

if (isSimulatorReadyForNextInstruction())

{

state = HOME;

printf("Time: %7.2f New state: %.20s Gantry in Home position. Press q to quit.\n", getSimulationTime(), state\_name[state]);

}

break;

}

sleepMilliseconds((long) 1000 / POLL\_LOOP\_RATE);

}

}

/\* state machine code for autonomous control mode \*/

else

{

/\* initialization of variables and controller window \*/

int state = HOME, part\_counter = 0, nozzle\_errors\_to\_check = 0, left\_nozzle\_part\_num = 0,

centre\_nozzle\_part\_num = 0, right\_nozzle\_part\_num = 0, component\_num, req\_target = 0;

char part\_placed = FALSE, Centre\_NozzleStatus = not\_holdingpart, Left\_NozzleStatus = not\_holdingpart,

Right\_NozzleStatus = not\_holdingpart, lookup\_photo = FALSE, lookdown\_photo = FALSE, preplace\_errors\_fixed = 0;

double requested\_theta\_left = 0, requested\_theta\_centre = 0, requested\_theta\_right = 0; //the required angle theta of the nozzle position

double preplace\_diff\_x = 0, preplace\_diff\_y = 0; //difference in required gantry position and actual gantry position for preplacement

printf("Time: %7.2f Initial state: %.15s Operating in automatic mode, there are %d parts to place\n\n", getSimulationTime(), state\_name[HOME], number\_of\_components\_to\_place);

/\* reorder the centroid list by feeder in ascending order and print details \*/

int component\_list[number\_of\_components\_to\_place];

int number\_compare[number\_of\_components\_to\_place];

int i, j, hold\_value;

for (i = 0; i < number\_of\_components\_to\_place; i++)

{

number\_compare[i] = pi[i].feeder; //holds the feeder numbers

component\_list[i] = i; //holds the index of the feeder numbers

}

for (i = 0; i < number\_of\_components\_to\_place; i++)

{

for (j = i+1; j < number\_of\_components\_to\_place; j++)

{

if (number\_compare[i] > number\_compare[j])

{ //reorder the indexed numbers based on the feeder numbers

hold\_value = component\_list[i];

component\_list[i] = component\_list[j];

component\_list[j] = hold\_value;

hold\_value = number\_compare[i];

number\_compare[i] = number\_compare[j];

number\_compare[j] = hold\_value;

}

}

}

//display the new order of the part details

for (int i = 0; i < number\_of\_components\_to\_place; i++)

{

component\_num = component\_list[i];

printf("Part %d details:\nDesignation: %s Footprint: %s Value: %.2f x: %.2f y: %.2f theta: %.2f Feeder: %d\n\n", component\_num,

pi[component\_num].component\_designation, pi[component\_num].component\_footprint, pi[component\_num].component\_value,

pi[component\_num].x\_target, pi[component\_num].y\_target, pi[component\_num].theta\_target, pi[component\_num].feeder);

}

/\* loop until user quits \*/

while(!isPnPSimulationQuitFlagOn())

{

switch (state)

{

case HOME:

if(isSimulatorReadyForNextInstruction())

{

component\_num = component\_list[part\_counter]; //hold the value of the part to be placed

if(part\_counter == number\_of\_components\_to\_place)

{

//Do nothing. Program is complete, wait for user to quit program.

}

else

{

setTargetPos(TAPE\_FEEDER\_X[pi[component\_num].feeder]+20, TAPE\_FEEDER\_Y[pi[component\_num].feeder]); //go to the first feeder in the list, +20 for the left nozzle

state = MOVE\_TO\_FEEDER;

printf("Time: %7.2f New state: %.20s Moving to tape feeder %d\n", getSimulationTime(), state\_name[state], pi[component\_num].feeder);

}

}

break;

case MOVE\_TO\_FEEDER:

//waiting for the simulator to complete movement of the gantry

if (isSimulatorReadyForNextInstruction())

{

if (Left\_NozzleStatus == not\_holdingpart)

{ //left nozzle goes first due to order of the parts in ascending by feeder number

lowerNozzle(LEFT\_NOZZLE);

state = LOWER\_LEFT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Arrived at feeder, lowering left nozzle\n", getSimulationTime(), state\_name[state]);

}

else if (Centre\_NozzleStatus == not\_holdingpart)

{ // centre nozzle picks up part after left nozzle

lowerNozzle(CENTRE\_NOZZLE);

state = LOWER\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Arrived at feeder, lowering centre nozzle\n", getSimulationTime(), state\_name[state]);

}

else if (Right\_NozzleStatus == not\_holdingpart)

{ //right nozzle is last to pick up part as it is closest to the higher feeder number

lowerNozzle(RIGHT\_NOZZLE);

state = LOWER\_RIGHT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Arrived at feeder, lowering right nozzle\n", getSimulationTime(), state\_name[state]);

}

}

break;

case LOWER\_LEFT\_NOZZLE:

if (isSimulatorReadyForNextInstruction())

{

if(Left\_NozzleStatus == not\_holdingpart)

{ //vacuum will apply when the nozzle is empty

applyVacuum(LEFT\_NOZZLE);

state = VAC\_LEFT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Applying vacuum\n", getSimulationTime(), state\_name[state]);

}

else if(Left\_NozzleStatus == holdingpart)

{ //vacuum will release the part when the nozzle is holding something

releaseVacuum(LEFT\_NOZZLE);

part\_placed = TRUE; //counter to indicate the part has been placed

state = VAC\_LEFT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Releasing vacuum to place part\n", getSimulationTime(), state\_name[state]);

}

}

break;

case LOWER\_CNTR\_NOZZLE:

if (isSimulatorReadyForNextInstruction())

{

if(Centre\_NozzleStatus == not\_holdingpart)

{ //vacuum will apply when the nozzle is empty

applyVacuum(CENTRE\_NOZZLE);

state = VAC\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Applying vacuum\n", getSimulationTime(), state\_name[state]);

}

else if(Centre\_NozzleStatus == holdingpart)

{ //vacuum will release the part when the nozzle is holding something

releaseVacuum(CENTRE\_NOZZLE);

part\_placed = TRUE; //counter to indicate the part has been placed

state = VAC\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Releasing vacuum to place part\n", getSimulationTime(), state\_name[state]);

}

}

break;

case LOWER\_RIGHT\_NOZZLE:

if (isSimulatorReadyForNextInstruction())

{

if(Right\_NozzleStatus == not\_holdingpart)

{ //vacuum will apply when the nozzle is empty

applyVacuum(RIGHT\_NOZZLE);

state = VAC\_RIGHT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Applying vacuum\n", getSimulationTime(), state\_name[state]);

}

else if(Right\_NozzleStatus == holdingpart)

{ //vacuum will release the part when the nozzle is holding something

releaseVacuum(RIGHT\_NOZZLE);

part\_placed = TRUE; //counter to indicate the part has been placed

state = VAC\_RIGHT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Releasing vacuum to place part\n", getSimulationTime(), state\_name[state]);

}

}

break;

case VAC\_LEFT\_NOZZLE:

//wait until the vacuum action is finished before raising the nozzle

if (isSimulatorReadyForNextInstruction())

{

raiseNozzle(LEFT\_NOZZLE);

state = RAISE\_LEFT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Raising left nozzle\n", getSimulationTime(), state\_name[state]);

}

break;

case VAC\_CNTR\_NOZZLE:

//wait until the vacuum action is finished before raising the nozzle

if (isSimulatorReadyForNextInstruction())

{

raiseNozzle(CENTRE\_NOZZLE);

state = RAISE\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Raising centre nozzle\n", getSimulationTime(), state\_name[state]);

}

break;

case VAC\_RIGHT\_NOZZLE:

//wait until the vacuum action is finished before raising the nozzle

if (isSimulatorReadyForNextInstruction())

{

raiseNozzle(RIGHT\_NOZZLE);

state = RAISE\_RIGHT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Raising right nozzle\n", getSimulationTime(), state\_name[state]);

}

break;

case RAISE\_LEFT\_NOZZLE:

if (isSimulatorReadyForNextInstruction())

{

if (part\_placed==FALSE)

{

left\_nozzle\_part\_num = component\_num; //storing the index of the next part number

part\_counter++; //incrementing the number of parts that have been picked

component\_num = component\_list[part\_counter]; //hold the index value of the next component

Left\_NozzleStatus = holdingpart; //if a part hasn't just been placed then it is determined that a part has just been picked up

nozzle\_errors\_to\_check++; //the picked up part needs to be checked for alignment errors

if (pi[part\_counter].feeder >=0 && pi[part\_counter].feeder <= 9) //this checks if there is another component in the list

{ //if there is another feeder waiting, then go to the next feeder in the reordered list

setTargetPos(TAPE\_FEEDER\_X[pi[component\_num].feeder], TAPE\_FEEDER\_Y[pi[component\_num].feeder]);

state = MOVE\_TO\_FEEDER;

printf("Time: %7.2f New state: %.20s Moving to feeder %d\n", getSimulationTime(), state\_name[state], pi[component\_num].feeder);

}

else

{ //if there is no other feeder in the file, then go to the camera

setTargetPos(LOOKUP\_CAMERA\_X,LOOKUP\_CAMERA\_Y);

state = MOVE\_TO\_CAMERA;

printf("Time: %7.2f New state: %.20s Part acquired, moving to look-up camera\n", getSimulationTime(), state\_name[state]);

}

}

else if (part\_placed==TRUE)

{

Left\_NozzleStatus = not\_holdingpart; //if the vacuum has just released a part, then the part has been placed and the nozzle is free again

part\_placed = FALSE; //reset the variable

lookdown\_photo = FALSE; //reset the photo variable

printf("Time: %7.2f %19s Part %d placed on PCB successfully\n\n", getSimulationTime(), " ", left\_nozzle\_part\_num);

if (Centre\_NozzleStatus == holdingpart)

{ //if the centre nozzle has a part, then move to the required position on the PCB

req\_target = centre\_nozzle\_part\_num; // this is required to obtain the correct alignment errors

setTargetPos(pi[centre\_nozzle\_part\_num].x\_target, pi[centre\_nozzle\_part\_num].y\_target);

state = MOVE\_TO\_PCB;

printf("Time: %7.2f New state: %.20s Moving to next position x: %3.2f y: %3.2f\n", getSimulationTime(), state\_name[state],pi[centre\_nozzle\_part\_num].x\_target, pi[centre\_nozzle\_part\_num].y\_target);

}

else if(part\_counter == number\_of\_components\_to\_place)

{ //there are no more parts to place, so move gantry to home

setTargetPos(HOME\_X,HOME\_Y);

state = MOVE\_TO\_HOME;

printf("Time: %7.2f New state: %.20s All parts have been placed! Moving to home\n", getSimulationTime(), state\_name[state]);

}

}

}

break;

case RAISE\_CNTR\_NOZZLE:

if (isSimulatorReadyForNextInstruction())

{

if (part\_placed==FALSE)

{

centre\_nozzle\_part\_num = component\_num; //holding the indexed value of the component for the centre nozzle

part\_counter++; //increment the part counter to ensure number of components are accounted for

component\_num = component\_list[part\_counter]; //hold the next part number index

Centre\_NozzleStatus = holdingpart; //if a part hasn't just been placed, then it is determined that a part has just been picked up

nozzle\_errors\_to\_check++; //the part needs to be checked for alignment errors

if(pi[part\_counter].feeder >=0 && pi[part\_counter].feeder <= 9)

{ //if there is another feeder number waiting, then go to the next feeder

setTargetPos(TAPE\_FEEDER\_X[pi[component\_num].feeder]-20, TAPE\_FEEDER\_Y[pi[component\_num].feeder]); //move to the next feeder for the right nozzle

state = MOVE\_TO\_FEEDER;

printf("Time: %7.2f New state: %.20s Moving to feeder %d\n", getSimulationTime(), state\_name[state], pi[component\_num].feeder);

}

else

{ //if no other feeder and no other parts to pick up, then go to the camera

setTargetPos(LOOKUP\_CAMERA\_X,LOOKUP\_CAMERA\_Y); //the gantry will move to the position above the camera

state = MOVE\_TO\_CAMERA;

printf("Time: %7.2f New state: %.20s Part acquired, moving to look-up camera\n", getSimulationTime(), state\_name[state]);

}

}

else if (part\_placed==TRUE)

{

Centre\_NozzleStatus = not\_holdingpart; //if the vacuum has just released a part, then the part has been placed and the nozzle is free again

lookdown\_photo = FALSE; //reset the photo variabla

part\_placed = FALSE; //reset the variable

printf("Time: %7.2f %19s Part %d placed on PCB successfully\n\n", getSimulationTime(), state\_name[state], centre\_nozzle\_part\_num);

if (Right\_NozzleStatus == holdingpart)

{ //if the right nozzle has a part then, move to the required position on the PCB

req\_target = right\_nozzle\_part\_num; // this is required in order to calculate preplace errors

setTargetPos(pi[right\_nozzle\_part\_num].x\_target, pi[right\_nozzle\_part\_num].y\_target); //right nozzle holding part\_counter-1

state = MOVE\_TO\_PCB;

printf("Time: %7.2f New state: %.20s Moving to next position x: %3.2f y: %3.2f\n", getSimulationTime(), state\_name[state],pi[right\_nozzle\_part\_num].x\_target, pi[right\_nozzle\_part\_num].y\_target);

}

else if(part\_counter == number\_of\_components\_to\_place)

{ //if there are no more parts to place then go to home

setTargetPos(HOME\_X,HOME\_Y);

state = MOVE\_TO\_HOME;

printf("Time: %7.2f New state: %.20s All parts have been placed! Moving to home\n", getSimulationTime(), state\_name[state]);

}

}

}

break;

case RAISE\_RIGHT\_NOZZLE:

if (isSimulatorReadyForNextInstruction())

{

if (part\_placed==FALSE)

{

right\_nozzle\_part\_num = component\_num; //storing the indexed value of the component

part\_counter++; //keeping a counter on the number of parts that have been picked

component\_num = component\_list[part\_counter]; //storing the next part index

Right\_NozzleStatus = holdingpart;//once nozzle is raised, if a part hasn't just been placed, then it is determined that a part has just been picked up

nozzle\_errors\_to\_check++; //right nozzle needs to be checked for alignment errors

setTargetPos(LOOKUP\_CAMERA\_X,LOOKUP\_CAMERA\_Y); //the gantry will move to the position above the camera

state = MOVE\_TO\_CAMERA;

printf("Time: %7.2f New state: %.20s All parts acquired, moving to look-up camera\n", getSimulationTime(), state\_name[state]);

}

else if (part\_placed==TRUE)

{

Right\_NozzleStatus = not\_holdingpart; //if the vacuum has just released a part, then the part has been placed and the nozzle is free again

lookdown\_photo = FALSE; //reset the photo variable

part\_placed = FALSE; //reset the variable

printf("Time: %7.2f %19s Part %d placed on PCB successfully\n\n", getSimulationTime(), state\_name[state], right\_nozzle\_part\_num);

if(part\_counter == number\_of\_components\_to\_place)

{ //if there are no more parts to place, then go to home

setTargetPos(HOME\_X,HOME\_Y);

state = MOVE\_TO\_HOME;

printf("Time: %7.2f New state: %.20s Moving to home.\n", getSimulationTime(), state\_name[state]);

}

else

{ // once the part is placed, if there are more parts then go to home to obtain details for the next feeder

state = HOME;

printf("Time: %7.2f New state: %.20s Moving to next feeder\n\n", getSimulationTime(), state\_name[state]);

}

}

}

break;

case MOVE\_TO\_CAMERA:

//waiting for the gantry to move to the camera position before taking look-up photo

if (isSimulatorReadyForNextInstruction())

{

takePhoto(PHOTO\_LOOKUP);

state = LOOK\_UP\_PHOTO;

printf("Time: %7.2f New state: %.20s Arrived at camera. Taking look-up photo of part\n", getSimulationTime(), state\_name[state]);

}

break;

case LOOK\_UP\_PHOTO:

if (isSimulatorReadyForNextInstruction())

{ //once look-up photo is taken, move on to calculate errors

lookup\_photo = TRUE;

state = CHECK\_ERROR;

printf("Time: %7.2f New state: %.20s Look-up photo acquired. Checking errors and calculating corrections\n", getSimulationTime(), state\_name[state]);

}

break;

case MOVE\_TO\_PCB:

//once the gantry has finished moving to the PCB, then it is ready to take a look-down photo

if (isSimulatorReadyForNextInstruction())

{

state = LOOK\_DOWN\_PHOTO;

printf("Time: %7.2f New state: %.20s Now at PCB. Taking look-down photo\n", getSimulationTime(), state\_name[state]);

}

break;

case LOOK\_DOWN\_PHOTO:

//take the look-down photo, then move on to calculate errors

takePhoto(PHOTO\_LOOKDOWN);

lookdown\_photo = TRUE;

state = CHECK\_ERROR;

printf("Time: %7.2f New state: %.20s Look-down photo acquired. Checking for errors in gantry alignment\n", getSimulationTime(), state\_name[state]);

break;

case CHECK\_ERROR:

//wait until the photo is taken, then calculate errors

if (isSimulatorReadyForNextInstruction() && lookup\_photo == TRUE)

{ //for look-up photos, cycle through and correct errors one by one using nozzle\_errors\_to\_check as a counter

if (nozzle\_errors\_to\_check == 3)

{ //since the right nozzle is last to pick up a part, it is the first to be corrected

double errortheta = getPickErrorTheta(RIGHT\_NOZZLE); //acquire the part misalignment from the look-up photo

requested\_theta\_right = pi[right\_nozzle\_part\_num].theta\_target - errortheta; //calculate misalignment of the part on the nozzle

printf("Time: %7.2f %19s Right part misalignment error: %3.2f Correction required: %3.2f degrees\n", getSimulationTime()," ", errortheta, requested\_theta\_right);

state = FIX\_NOZZLE\_ERROR;

printf("Time: %7.2f New state: %.20s Correction made to right nozzle for part alignment\n", getSimulationTime(), state\_name[state]);

}

else if (nozzle\_errors\_to\_check == 2)

{ //the centre nozzle is second to pick a part and is second to have the alignment corrected

double errortheta = getPickErrorTheta(CENTRE\_NOZZLE); //acquire the part misalignment from the look-up photo

requested\_theta\_centre = pi[centre\_nozzle\_part\_num].theta\_target - errortheta; //calculate misalignment of the part on the nozzle

printf("Time: %7.2f %19s Centre part misalignment error: %3.2f Correction required: %3.2f degrees\n", getSimulationTime()," ", errortheta, requested\_theta\_centre);

state = FIX\_NOZZLE\_ERROR;

printf("Time: %7.2f New state: %.20s Correction made to centre nozzle for part alignment\n", getSimulationTime(), state\_name[state]);

}

else if (nozzle\_errors\_to\_check == 1)

{ //the left nozzle was first to pick up a part, and if it is the only nozzle used then only one error to check

double errortheta = getPickErrorTheta(LEFT\_NOZZLE); //acquire the part misalignment from the look-up photo

requested\_theta\_left = pi[left\_nozzle\_part\_num].theta\_target - errortheta; //calculate misalignment of the part on the nozzle

printf("Time: %7.2f %19s Left part misalignment error: %3.2f Correction required: %3.2f degrees\n", getSimulationTime()," ", errortheta, requested\_theta\_left);

state = FIX\_NOZZLE\_ERROR;

printf("Time: %7.2f New state: %.20s Correction made to left nozzle for part alignment\n", getSimulationTime(), state\_name[state]);

}

else

{ //if no more nozzle errors to check, then reset the photo variable and go to the PCB to place parts

lookup\_photo = FALSE;

req\_target = left\_nozzle\_part\_num; //this is needed to obtain and calculate the relevant misalignment errors later

setTargetPos(pi[left\_nozzle\_part\_num].x\_target, pi[left\_nozzle\_part\_num].y\_target);

state = MOVE\_TO\_PCB;

printf("Time: %7.2f New state: %.20s No furthers errors. Moving to PCB\n", getSimulationTime(), state\_name[state]);

}

}

else if (isSimulatorReadyForNextInstruction() && lookdown\_photo == TRUE)

{ //calculate the difference between the required target and the error of the gantry over the PCB

preplace\_diff\_x = pi[req\_target].x\_target - (pi[req\_target].x\_target+getPreplaceErrorX()); //calculate the difference between the required x position and the actual x position of the gantry

preplace\_diff\_y = pi[req\_target].y\_target - (pi[req\_target].y\_target+getPreplaceErrorY()); //calculate the difference between the required y position and the actual y position of the gantry

printf("Time: %7.2f %19s Preplace misalignment error: x=%3.2f y=%3.2f\n", getSimulationTime(), " ", getPreplaceErrorX(), getPreplaceErrorY());

amendPos(preplace\_diff\_x, preplace\_diff\_y); //fix the gantry preplace position over the PCB

state = FIX\_PREPLACE\_ERROR;

printf("Time: %7.2f New state: %.20s Correction made to gantry position\n", getSimulationTime(), state\_name[state]);

}

break;

case FIX\_NOZZLE\_ERROR:

if (isSimulatorReadyForNextInstruction())

{ //apply correction to nozzle rotation for part alignment

if (nozzle\_errors\_to\_check == 3)

{ //using nozzle\_errors\_to\_check as a counter to ensure the correct nozzle is addressed

rotateNozzle(RIGHT\_NOZZLE, requested\_theta\_right); //rotate the nozzle by the required calculated angle theta

nozzle\_errors\_to\_check--; //decrement to track the errors needed for correction

state = CHECK\_ERROR;

printf("Time: %7.2f New state: %.20s Checking for errors...\n", getSimulationTime(),state\_name[state]);

}

else if (nozzle\_errors\_to\_check == 2)

{ //centre nozzle is second to be corrected

rotateNozzle(CENTRE\_NOZZLE, requested\_theta\_centre); //rotate the nozzle by the required calculated angle theta

nozzle\_errors\_to\_check--;

state = CHECK\_ERROR;

printf("Time: %7.2f New state: %.20s Checking for errors...\n", getSimulationTime(),state\_name[state]);

}

else if (nozzle\_errors\_to\_check == 1)

{ //since the left nozzle was first to pick up a part, it is last to be corrected. Applies if it is the only nozzle in use for a singular part

rotateNozzle(LEFT\_NOZZLE, requested\_theta\_left); //rotate the nozzle by the required calculated angle theta

nozzle\_errors\_to\_check--;

state = CHECK\_ERROR;

printf("Time: %7.2f New state: %.20s Checking for errors...\n", getSimulationTime(),state\_name[state]);

}

}

break;

case FIX\_PREPLACE\_ERROR:

if (isSimulatorReadyForNextInstruction())

{

preplace\_errors\_fixed++; //counter to track preplace errors that have been corrected

if (Left\_NozzleStatus == holdingpart)

{ //only need to apply correction if the nozzle is holding a part

lowerNozzle(LEFT\_NOZZLE);

state = LOWER\_LEFT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Now lowering left nozzle to place part on PCB\n", getSimulationTime(),state\_name[state]);

}

else if (Centre\_NozzleStatus == holdingpart)

{//only need to apply correction if the nozzle is holding a part

lowerNozzle(CENTRE\_NOZZLE);

state = LOWER\_CNTR\_NOZZLE;

printf("Time: %7.2f New state: %.20s Now lowering centre nozzle to place part on PCB\n", getSimulationTime(),state\_name[state]);

}

else if (Right\_NozzleStatus == holdingpart)

{//only need to apply correction if the nozzle is holding a part

lowerNozzle(RIGHT\_NOZZLE);

state = LOWER\_RIGHT\_NOZZLE;

printf("Time: %7.2f New state: %.20s Now lowering right nozzle to place part on PCB\n", getSimulationTime(),state\_name[state]);

}

}

break;

case MOVE\_TO\_HOME:

if (isSimulatorReadyForNextInstruction())

{ //moves the gantry to home position once placement of all components is complete

state = HOME;

printf("Time: %7.2f New state: %.20s Gantry in Home position. Placement complete. Press q to quit.\n", getSimulationTime(), state\_name[state]);

}

break;

} //closing switch

sleepMilliseconds((long) 1000 / POLL\_LOOP\_RATE);

}//closing while loop

}

pnpClose();

return 0;

}