ELE4307
Real Time Systems

Assignment 1
Control of Pick and Place Machine for SMT Assembly

Student Name: Kate Bowater

Student Number: U1019160

Contents

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 component, which are as follows:

1. Press a number key in the range 0-9 to move the gantry to the tape feeder
2. Press ‘p’ to pick up a part using the centre nozzle
3. Press ‘c’ to move to the look-up camera. Machine will automatically take a photo, then move to the PCB and take a look-down photo
4. Press ‘r’ or ‘a’ in any order to correct nozzle and preplacement errors
5. Press ‘p’ to place the part on the PCB
6. Repeat until all parts have been placed. System will notify user when it is complete

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 A State Diagram

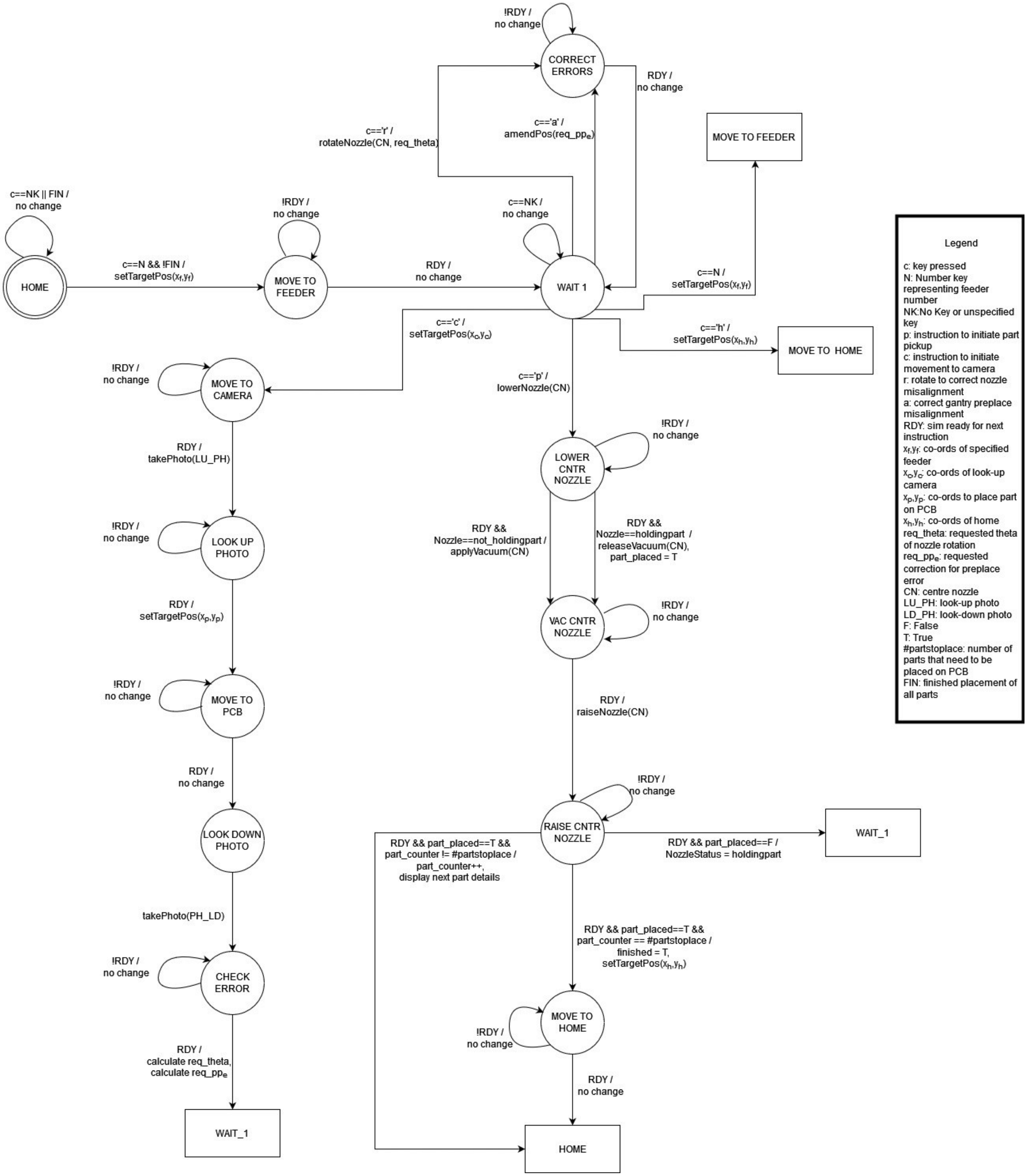

Figure 1: Mealy state-based diagram for manual operation

Part B

Figure 2 is the Mealy state-based diagram for autonomous operation. All three nozzles are used in this mode to maximise productivity and reduce the time to pick and place parts. A design choice that was implemented is to 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.

If any of the feeder numbers are the same the program will then sort by ascending y-coordinate so the gantry will place parts on the PCB by closest position to the feeders. Sorting by the y-coordinate was chosen over sorting by the x-coordinate as the test results (see Table 3) indicated that sorting by y-coordinate reduced the run time compared to the x-coordinate in most instances.

Part B State Diagram

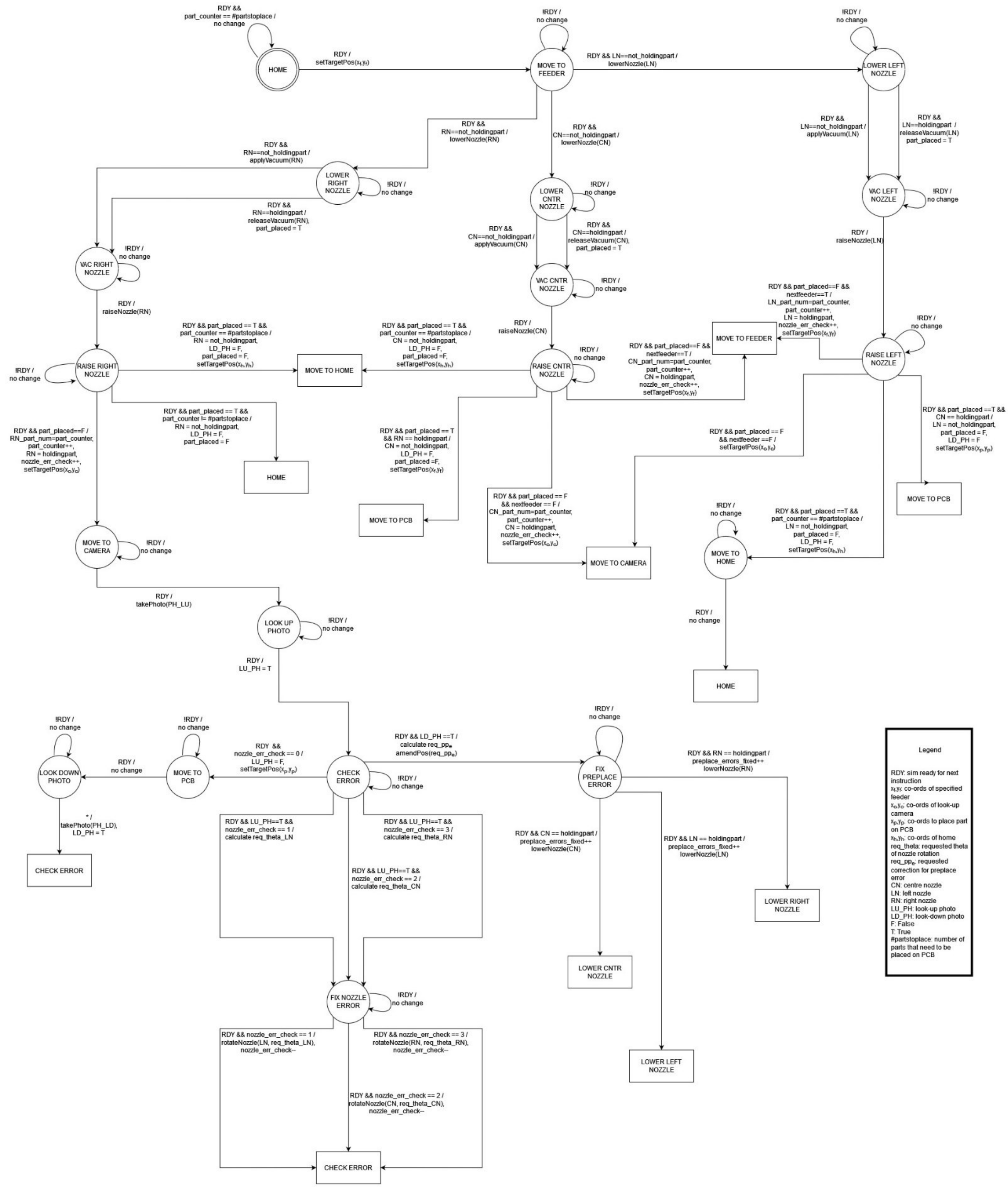

Figure 2: Mealy state-based diagram for autonomous operation

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.

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 phases, functionality and performance. Table 2 outlines the results of functional testing and Table 3 outlines the results of performance.

Regarding functionality, formal testing entailed running the program starting with one part in the centroid file and increasing by one up to 30 parts and checking the placement details each time against the contents of the centroid file to determine any inconsistencies. This was to ensure it could pick and place any number of components in the centroid file without fail. If any problems were discovered, the program was run again with the same file to determine if the issue was due to the written code or due to asynchronization of the controller to the simulator as 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 not placing parts on the PCB even though the vacuum was released. After closing the program and running it again the simulator would perform the required actions correctly.

Table 2: Functional testing checklist for autonomous mode

Test Case #	Description	Pass/Fail	Notes
1	Components are reordered ascending by feeder number and y-coordinate.	Pass	No issues
2	All part details are displayed at the initiation of the program	Pass	Part numbers still correspond to the part number in the centroid file, so appear out of order on the controller
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	Issues with nozzle and preplace misalignment errors 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 or skipping pick up of parts. Reset amends issue
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	Pass	No issues

To measure performance, testing entailed running the program without any sorting of the part list, recording the time, then running the program again with reordering of the parts, specifically by ascending feeder number and then ascending x- or y-coordinate for parts with the same feeder number to reduce the distance the gantry must travel. The number of components starts at 5 and increases in increments of 5 to a maximum of 40 parts. Several runs were completed using different component details and an average taken. This is because run times vary depending on x- and y-coordinates of the parts to be placed on the PCB.

The results were recorded in Table 3. The difference in run time is more noticeable at larger numbers of components. It is also worth noting that sorting by y-coordinate typically yielded a slightly faster run time than sorting by x-coordinate, hence it was implemented in the final program submission.

Table 3: Time in seconds to pick and place parts

# of parts	Unsorted list	Sorted by feeder # & y-coordinate	Sorted by feeder # & x-coordinate
5	12.91	12.85	13.50
10	24.45	23.43	23.72
15	36.44	33.69	34.37
20	49.32	45.26	46.24
25	62.5	59.40	58.66
30	74.87	67.59	70.03
35	90.48	81.20	81.51
40	102.24	93.95	92.94

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: 5 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);
    }
  } // end of manual mode



  /*
  ****************************************************
  *
  *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;
    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 feeder_num_compare[number_of_components_to_place];
int y_target_compare[number_of_components_to_place];
int i, j, hold_value;
for (i = 0; i < number_of_components_to_place; i++)
{
  feeder_num_compare[i] = pi[i].feeder;  //holds the feeder numbers in the centroid file
  y_target_compare[i] = pi[i].y_target;  // holds the y coord values in the centroid file
  component_list[i] = i;  //holds the indexes that correlate to the values
}
for (i = 0; i < number_of_components_to_place; i++)
{
  for (j = i+1; j < number_of_components_to_place; j++)
  {
    if (feeder_num_compare[i] > feeder_num_compare[j])
    {  //reorder the indexed numbers based on the feeder numbers. Swaps the y-coords so they
correlate
      hold_value = component_list[i];
      component_list[i] = component_list[j];
      component_list[j] = hold_value;
      hold_value = feeder_num_compare[i];
      feeder_num_compare[i] = feeder_num_compare[j];
      feeder_num_compare[j] = hold_value;
      hold_value = y_target_compare[i];
      y_target_compare[i] = y_target_compare[j];
      y_target_compare[j] = hold_value;
```

```
        }
        // sort by ascending y-coordinates if the feeder numbers are the same
        else if (feeder_num_compare[i] == feeder_num_compare[j])
        {
          if (y_target_compare[i] > y_target_compare[j])
          {
            hold_value = component_list[i];
            component_list[i] = component_list[j];
            component_list[j] = hold_value;
            hold_value = y_target_compare[i];
            y_target_compare[i] = y_target_compare[j];
            y_target_compare[j] = hold_value;
            hold_value = feeder_num_compare[i];
            feeder_num_compare[i] = feeder_num_compare[j];
            feeder_num_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:\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. The counter starts at zero
        if(part_counter == number_of_components_to_place)
        {
         //Do nothing. Program is complete, wait for user to quit program.
        }
        else
        { //go to the first feeder in the list, +20 for the left nozzle positioning
          setTargetPos(TAPE_FEEDER_X[pi[component_num].feeder]+20,
TAPE_FEEDER_Y[pi[component_num].feeder]);
          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) // if the nozzle is already holding a part, then skip to the next nozzle

          {  //left nozzle goes first due to order of the parts 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) // applies when the nozzle has not just placed a part
    {
      left_nozzle_part_num = component_num;  //storing the index of the part number from
the reordered list
      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 (part_counter == number_of_components_to_place)
      {  //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 there is another feeder waiting, then go to the next feeder in the reordered list, positioned for the centre nozzle
                        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 (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)  //applies if the nozzle has not just placed a part on the PCB

                {

                  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 (part_counter == number_of_components_to_place)
```

```
        {   //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 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 (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)  // applies if the nozzle hasn't just placed a part on the PCB
            {
              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 up
              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 right nozzle is the last to pick up a part, so the gantry will move to 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
          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())
          {
            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;
}
```