ELE4307

Real-Time Systems

Assignment 2

Student name: Kate Bowater

Student number: U1019160

Contents

[Introduction 3](#_Toc172649189)

[System Design 3](#_Toc172649190)

[Process Spawning 3](#_Toc172649191)

[Inter-process Communication 3](#_Toc172649192)

[Synchronization 4](#_Toc172649193)

[Testing Results 4](#_Toc172649194)

[Autonomous Mode 4](#_Toc172649195)

[Manual Mode 6](#_Toc172649196)

[Testing Notes 6](#_Toc172649197)

[Appendix 7](#_Toc172649198)

[Startup and Monitor 7](#_Toc172649199)

[Display 13](#_Toc172649200)

[Simulator 25](#_Toc172649201)

[Controller 44](#_Toc172649202)

# Introduction

This report documents the design of a C program which simulates a pick and place machine for assembly of Surface Mount Technology (SMT) based Printed Circuits Boards (PCBs). The program initiates with a Startup and Monitor process that spawns three child processes: Display, Simulator and Controller. All processes are designed to send messages to the Display via pipes which the Display then prints to the terminal. The Simulator and Controller run the actions of the pick and place machine for both manual and autonomous mode. The Startup and Monitor process waits for all three child processes to terminate before concluding the program.

# System Design

## Process Spawning

The Startup and Monitor process is designed to spawn three child processes in the following order:

1. Display
2. Simulator
3. Controller

The Display process was chosen to be first so that the other process can send their messages to it for printing as per the specification which requires all messages to be printed by the Display. The Simulator is second because it initiates the pick and place program, and once the Controller is spawned and overlayed the picking and placing begins.

## Inter-process Communication

Inter-process communication is accomplished through pipes which are set up in the Startup and Monitor process. These are anonymous pipes, and each process uses a separate pipe to send messages to the Display process. Because the parent (Startup and Monitor) opens all pipes before fork(), each child process must then close the unneeded ends of the pipes before overlaying.

The specification stated that, as much as possible, messages were required to be sent to the Display via these pipes to be printed. This could be achieved for the Controller and Simulator (which then close their respective pipes upon terminating) but the Startup and Monitor process is required to print directly to the terminal in one instance: after the Display process has terminated. This is because the Display process cannot terminate until the pipe from the Startup and Monitor is closed. Startup and Monitor will use printf() to print directly to the terminal in this instance.

The specification requires that the child process PIDs are printed upon creation by the Startup and Monitor process, however this is only available with getpid() within the child process itself. Therefore, a shared memory is created for each child to store its PID so the Startup and Monitor can access it and then send it to the Display for printing.

A decision was made to make the pipe from the Startup and Monitor process to the Display a non-blocking pipe to prevent the read() function in Display from blocking other processes sending messages. Due to the lack of messages coming from Startup and Monitor while the pick and place simulation was in action the Display could not print and the terminal would only print everything once the Startup and Monitor pipe had been closed. However, another issue presented in that the Display would then no longer wait for Startup and Monitor messages and terminate too early. To correct this, the Display is designed to enter an infinite ‘while’ loop which will check the buffer of the pipe from Startup and Monitor each time around the loop. A conditional if statement is implemented to allow the Display process to exit once the pipe is closed.

## Synchronization

In terms of synchronization, the Startup and Monitor process uses semaphores to control access to the shared memory file containing the PIDs of the children. This is to ensure that it does not access the memory file before the child processes have written their PIDs to it.

A semaphore was used to ensure the Simulator and the Controller would wait for the Startup and Monitor process to finish spawning before beginning the pick and place program. This is to ensure the Display has a chance to print the messages from Startup and Monitor before the simulation begins due to the non-blocking nature of the pipe from Startup and Monitor to Display.

A semaphore was used in the Controller process just before finishing the pick and place program and terminating. The reason this was needed was due to the unloading of the PCB which takes 1.5 seconds for the Simulator to perform. In that time, the Controller would sometimes terminate before the Simulator finished the task. Implementing a semaphore to force it to wait resolved the issue.

In the event the program is ended early by pressing ‘q’, a race condition was discovered where the Simulator and Controller would terminate out of order, preventing the Display and Startup and Monitor processes from terminating, thus causing the program to hang. A semaphore was put in place to ensure the Controller terminates before the Simulator which appeared to resolve the issue.

Consideration was made on placing a semaphore in the Controller function isSimulatorReadyForNextInstruction() due to access to shared memory and potential for a race condition when both the Simulator and the Controller are attempting to access the value in pnp -> ready\_for\_next\_instruction. Although no issues had been experienced in regards to accessing the shared memory, it is anticipated that if a problem were to occur with desynchronization between Controller and Simulator that a semaphore located in isSimulatorReadyForNextInstruction() may resolve the issue. An attempt was made to code in the semaphore; however the program hangs if it is implemented and so it is commented out.

# Testing Results

## Autonomous Mode

The first part of testing was to run the program in autonomous mode. It was allowed to run with varying numbers of components to pick and place in the range from 1 to 8 to ensure there were no issues when continuing to use all three nozzles. It was also to determine any consistency with issues that may be presented, i.e. if the issue is related to a particular state, function, or restricted to a singular run wherein the issue is unrepeatable. Table 1 outlines the expected results and actual results of testing in autonomous mode. Notes from testing are provided in a separate section below.

Table 1: Test cases in autonomous mode

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Expected Result** | **Actual Result** |
| Startup and Monitor successfully spawns all three child processes | Once done, Startup message prints to the Display “Process Spawning complete” | Once done, Startup message prints to the Display “Process Spawning complete” |
| Once overlayed, Display receives messages through pipes from all processes for printing | Display prints initial messages from Startup containing Simulator and Controller PIDs.  Display prints initial message from Simulator: “Pick and place machine simulation started successfully!”  Display prints initial message from Controller: “Pick and place controller started successfully!” | Display prints initial messages from Startup containing Simulator and Controller PIDs.  Display prints initial message from Simulator: “Pick and place machine simulation started successfully!”  Display prints initial message from Controller: “Pick and place controller started successfully!” |
| Display sorts and prints messages received in order of simulation time | All messages are printed in order of simulation time | Most messages are printed in order of simulation time (See note 1) |
| PCB required to be loaded onto machine before placing parts, and unloaded at the end | Controller instructs Simulator to load and unload PCB, which takes 1.5 seconds | Controller instructs Simulator to load and unload PCB, which takes 1.5 seconds (See note 2) |
| Pick and place program runs automatically to successfully place all parts without intervention | Autonomous picking and placing successfully places all components | Autonomous picking and placing successfully places all components |
| Controller and Simulator processes terminate once picking and placing is complete or upon ‘q’ press | Pick and Place program completes and automatically terminates without error | Pick and Place program completes and automatically terminates without error |
| Startup and Monitor waits for children to terminate before exiting program | Startup and Monitor prints the PIDs of each child that terminates, then exits without error | Startup and Monitor prints the PIDs of each child that terminates, then exits without error |
| Processes are synchronized and produces no issues in performance | Race conditions are not present | Race conditions are not present (See note 3) |

## Manual Mode

The second part of testing involved manual operation of the pick and place simulation for 1 to 3 components. Table 2 outlines the results of the test cases. Notes from testing are provided in a separate section below.

Table 2: Test cases for manual mode of pick and place program

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Expected Result** | **Actual Result** |
| Startup and Monitor successfully spawns all three child processes | Once done, Startup message prints to the Display “Process Spawning complete” | Once done, Startup message prints to the Display “Process Spawning complete” |
| Once overlayed, Display receives messages through pipes from all processes for printing | Display prints initial messages from Startup containing Simulator and Controller PIDs.  Display prints initial message from Simulator: “Pick and place machine simulation started successfully!”  Display prints initial message from Controller: “Pick and place controller started successfully!” | Display prints initial messages from Startup containing Simulator and Controller PIDs.  Display prints initial message from Simulator: “Pick and place machine simulation started successfully!”  Display prints initial message from Controller: “Pick and place controller started successfully!” (See note 4) |
| Display sorts and prints messages received in order of simulation time | All messages are printed in order of simulation time | Most messages are printed in order of simulation (See note 1) |
| Pick and place program runs successfully to place all parts upon instruction | Manual picking and placing of all components can be done successfully through appropriate key presses | Manual picking and placing of all components can be done successfully through appropriate key presses |
| Controller and Simulator processes terminate once picking and placing is complete or upon ‘q’ press | Pick and Place program completes, and processes terminate without error | Pick and Place program completes, and processes terminate without error |
| Startup and Monitor waits for children to terminate before terminating program | Startup and Monitor prints the PIDs of each child that terminates, then exits without error | Startup and Monitor prints the PIDs of each child that terminates, then exits without error |
| Processes are synchronized and produces no issues in performance | Race conditions are not present | Race conditions are not present (See note 3) |

## Testing Notes

Note 1: Upon review of the order of messages printed, occasionally there are instances where the Display will print a message with an earlier simulation time after one with a later simulation time. The cause has been pinpointed to the string comparisons performed in the Display process. Issue remains unresolved.

Note 2: The instruction to load and unload the PCB is successful and the time it takes to do so is 1.5 seconds, however this action causes the program to halt while the Simulator is doing it. The messages from the Simulator to indicate that it is loading/unloading are not printed by the Display until after the Simulator has completed and moved on. This is attributed to the blocking nature of the read() function for the Controller read buffer.

Note 3: At the time of testing, it is unknown if there are any further race conditions or desynchronized behaviour that have not been resolved.

Note 4: After initialisation, the program appears to stop after printing message from Simulator: “Pick and place machine simulation started successfully!” and will only resume printing after pressing a number key to instruct the Simulator to move the gantry to a feeder. The Display will then print all at once everything that was sent through the pipe including component details from the Controller. This is attributed to a fault in the design of the Display program due to the blocking operation of read() for the Simulator and Controller pipe. The issue resolves if the Simulator pipe is set to non-blocking however this caused other issues with the processing of messages by Display, and so it is not implemented.

# Appendix

## Startup and Monitor

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Pick and Place Program

\* Startup and Monitoring main file

\* By: Kate Bowater

\* Student #: U1019160

\*

\* This file creates forks and pipes for communication

\* between the Display, Simulator, and Controller

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <string.h>

#include <semaphore.h>

#include <fcntl.h>

#include <sys/mman.h>

#define NUMBER\_OF\_CHILDREN 3

#define CHILD 0

#define FORK\_FAILED -1

#define READ 0

#define WRITE 1

#define MEMORY\_MAPPED\_FILE "pid\_shared\_file"

int pipe\_Startup\_to\_Display[2]; //[0] for read, [1] for write

int pipe\_Simulator\_to\_Display[2];

int pipe\_Controller\_to\_Display[2];

typedef struct

{

int display\_pid;

int sim\_pid;

int contrl\_pid;

} PID\_store;

int main()

{

PID\_store \*pid\_store;

/\* initialize file for memory mapping \*/

int PID\_memoryfile = open(MEMORY\_MAPPED\_FILE, (O\_CREAT | O\_RDWR), 0666);

if (PID\_memoryfile < 0)

{

perror("creation/opening of PID\_memoryfile failed");

exit(1);

}

ftruncate(PID\_memoryfile, sizeof(PID\_store));

/\* map the file to memory \*/

pid\_store = (PID\_store \*)mmap(0, sizeof(PID\_store), (PROT\_READ | PROT\_WRITE), MAP\_SHARED, PID\_memoryfile, (off\_t)0);

if (pid\_store == MAP\_FAILED)

{

perror("memory mapping of PID\_memoryfile failed");

close(PID\_memoryfile);

exit(2);

}

//named semaphore creation

sem\_t \*sem\_Startup = sem\_open("/sem\_Startup", O\_CREAT, 0666, 1);

sem\_t \*sem\_Sim = sem\_open("/sem\_Sim", O\_CREAT, 0666, 0);

sem\_t \*sem\_Contrl = sem\_open("/sem\_Contrl", O\_CREAT, 0666, 0);

sem\_t \*sem\_Display = sem\_open("/sem\_Display", O\_CREAT, 0666, 0);

if (sem\_Startup == SEM\_FAILED || sem\_Sim == SEM\_FAILED || sem\_Contrl == SEM\_FAILED)

{

perror("Semaphore creation failed");

exit(6);

}

char \*strFromStartup;

char Startup\_str\_array[100];

char pipeStartupToDisplayReadFdStr[10]; //used to allow display end of pipe to read from startup process

char pipeSimToDisplayReadFdStr[10]; //used to allow display end of pipe to read from simulator process

char pipeContrlToDisplayReadFdStr[10]; //used to allow display end of pipe to read from controller process

char pipeSimToDisplayWriteFdStr[10]; // used to allow simulator to write to display pipe

char pipeContrlToDisplayWriteFdStr[10]; // used to allow controller to write to display pipe

int Status; //for parent to monitor the status of child

//set up pipes before fork

if (pipe(pipe\_Startup\_to\_Display) < 0 || pipe(pipe\_Simulator\_to\_Display) < 0 || pipe(pipe\_Controller\_to\_Display) < 0)

{

perror("Pipe creation failed");

exit(5);

}

//setting up pipe to be non-blocking so display can continue reading from other pipes

if (fcntl(pipe\_Startup\_to\_Display[READ], F\_SETFL, O\_NONBLOCK) < 0) {

perror("Pipe non-blocking failed");

exit(5);

}

// string creation for read end of pipe, to allow overlayed process to read or write

sprintf(pipeStartupToDisplayReadFdStr, "%d", pipe\_Startup\_to\_Display[READ]);

sprintf(pipeSimToDisplayReadFdStr, "%d", pipe\_Simulator\_to\_Display[READ]);

sprintf(pipeContrlToDisplayReadFdStr, "%d", pipe\_Controller\_to\_Display[READ]);

sprintf(pipeSimToDisplayWriteFdStr, "%d", pipe\_Simulator\_to\_Display[WRITE]);

sprintf(pipeContrlToDisplayWriteFdStr, "%d", pipe\_Controller\_to\_Display[WRITE]);

for (int count = 0; count < NUMBER\_OF\_CHILDREN; count++)

{

pid\_t return\_pid = fork();

//using switch to organise the child and parent processes

switch (return\_pid)

{

case CHILD:

if (count == 0) //first child is the display process

{

pid\_store -> display\_pid = getpid(); // save the child pid to the shared memory

close(pipe\_Startup\_to\_Display[WRITE]); //display will only be reading through the pipes

close(pipe\_Simulator\_to\_Display[WRITE]);

close(pipe\_Controller\_to\_Display[WRITE]);

sem\_post(sem\_Display); // allow parent process to continue and read the pid from shared memory

execl("..\\Assgn2\_2024\_Display\\bin\\Release\\Assgn2\_2024\_Display", "Assgn2\_2024\_Display", pipeStartupToDisplayReadFdStr,

pipeSimToDisplayReadFdStr, pipeContrlToDisplayReadFdStr, (char \*) NULL);

perror("Display overlay failed");

exit(5);

}

if(count == 1) // second child is the simulator

{

pid\_store -> sim\_pid = getpid(); //store pid to shared memory

close(pipe\_Startup\_to\_Display[WRITE]); //Simulator overlayed does not need this pipe

close(pipe\_Simulator\_to\_Display[READ]); //Simulator will only write to pipe

//close(pipe\_Startup\_to\_Display[READ]); // this pipe is already closed by the parent before spawning this child

close(pipe\_Controller\_to\_Display[READ]); //does not need access to the controller pipe

close(pipe\_Controller\_to\_Display[WRITE]);

sem\_post(sem\_Sim); //allow parent process to continue so it can access pid in shared memory

execl("..\\Assgn2\_2024\_Simulator\\bin\\Release\\Assgn2\_2024\_Simulator", "Assgn2\_2024\_Simulator", pipeSimToDisplayWriteFdStr, (char \*) NULL);

perror("Simulator overlay failed");

exit(5);

}

if(count == 2) // final child is the Controller

{

pid\_store -> contrl\_pid = getpid(); // store the controller pid

close(pipe\_Startup\_to\_Display[WRITE]); //Controller process overlayed does not need this pipe

close(pipe\_Controller\_to\_Display[READ]); //Controller will only write to pipe

//close(pipe\_Simulator\_to\_Display[READ]); //these pipes were already closed by the parent before spawning

//close(pipe\_Simulator\_to\_Display[WRITE]);

sem\_post(sem\_Contrl); //allow startup process to continue

execl("..\\Assgn2\_2024\_Controller\\bin\\Release\\Assgn2\_2024\_Controller", "Assgn2\_2024\_Controller", pipeContrlToDisplayWriteFdStr, (char \*) NULL);

perror("Controller overlay failed");

exit(5);

}

case FORK\_FAILED:

perror("Fork failed: ");

exit(5);

// the Startup process continues here as the parent

default:

if(count == 0) // after spawning display, startup and monitor comes here

{

sem\_wait(sem\_Display); // wait until display process has stored the pid to shared memory

int DisplayPID = pid\_store -> display\_pid;

sprintf(Startup\_str\_array, "Display process created with PID %d\n", DisplayPID);

write(pipe\_Startup\_to\_Display[WRITE], Startup\_str\_array, strlen(Startup\_str\_array));

close(pipe\_Startup\_to\_Display[READ]); //parent will only write to pipe, so close read end

}

if(count == 1) //after spawning simulator

{

sem\_wait(sem\_Sim); // wait until the simulator has stored its pid to shared memory

int SimPID = pid\_store -> sim\_pid;

sprintf(Startup\_str\_array, "Simulator process created with PID %d\n", SimPID);

write(pipe\_Startup\_to\_Display[WRITE], Startup\_str\_array, strlen(Startup\_str\_array));

close(pipe\_Simulator\_to\_Display[READ]); //does not need access to the simulator pipe

close(pipe\_Simulator\_to\_Display[WRITE]);

}

if(count == 2) // after spawning controller

{

sem\_wait(sem\_Contrl); // wait until controller has stored its pid to shared memory

int ContrlPID = pid\_store -> contrl\_pid;

sprintf(Startup\_str\_array, "Controller process created with PID %d\n", ContrlPID);

write(pipe\_Startup\_to\_Display[WRITE], Startup\_str\_array, strlen(Startup\_str\_array));

close(pipe\_Controller\_to\_Display[READ]); //does not need access to the controller pipe

close(pipe\_Controller\_to\_Display[WRITE]);

}

} // end switch

}//end for loop

strFromStartup = "Process spawning complete\n";

write(pipe\_Startup\_to\_Display[WRITE], strFromStartup, strlen(strFromStartup));

strFromStartup = "Waiting for children to terminate\n";

write(pipe\_Startup\_to\_Display[WRITE], strFromStartup, strlen(strFromStartup));

sem\_post(sem\_Startup); // Controller and Simulator wait for this to prevent race conditions

munmap(pid\_store, sizeof(PID\_store)); // memory mapped file no longer needed

close(PID\_memoryfile);

// waiting for children to terminate. Prints the PID and exit status of each.

// The controller always terminates first followed by Simulator

pid\_t Contrl\_pid = wait(&Status);

sprintf(Startup\_str\_array, "Controller with PID %d terminated with status code %d\n", Contrl\_pid, Status>>8);

write(pipe\_Startup\_to\_Display[WRITE], Startup\_str\_array, strlen(Startup\_str\_array));

pid\_t Sim\_pid = wait(&Status);

sprintf(Startup\_str\_array, "Simulator with PID %d terminated with status code %d\n", Sim\_pid, Status>>8);

write(pipe\_Startup\_to\_Display[WRITE], Startup\_str\_array, strlen(Startup\_str\_array));

// Display pipe needs to be closed to allow Display process to terminate

close(pipe\_Startup\_to\_Display[WRITE]);

pid\_t Display\_pid = wait(&Status);

printf("STARTUP\nDisplay with PID %d terminated with status code %d\n", Display\_pid, Status>>8);

printf("STARTUP\nProgram has ended. Press any key to exit.\n");

sem\_close(sem\_Startup);

sem\_unlink("/sem\_Startup");

sem\_close(sem\_Sim);

sem\_unlink("/sem\_Sim");

sem\_close(sem\_Contrl);

sem\_unlink("/sem\_Contrl");

sem\_close(sem\_Display);

sem\_unlink("/sem\_Display");

exit(0);

}//end main

## Display

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <semaphore.h>

#define READ\_BLOCK\_SIZE 50

int main(int argc, char \*argv[])

{

char readBufferStartup[READ\_BLOCK\_SIZE+1], readBufferSim[READ\_BLOCK\_SIZE+1], readBufferContrl[READ\_BLOCK\_SIZE+1];

ssize\_t bytesReadStartup, bytesReadSim, bytesReadContrl;

// set up the file descriptors for each of the other processes to communicate

int readStartupFd = atoi(argv[1]);

int readSimFd = atoi(argv[2]);

int readContrlFd = atoi(argv[3]);

printf("DISPLAY\nNow reading and printing from pipes\n");

//display process will stop here until there is a message to be read

bytesReadSim = read(readSimFd, readBufferSim, READ\_BLOCK\_SIZE);

bytesReadContrl = read(readContrlFd, readBufferContrl, READ\_BLOCK\_SIZE);

while(1)

{ // check if there are bytes in the Startup buffer

bytesReadStartup = read(readStartupFd, readBufferStartup, READ\_BLOCK\_SIZE);

if (bytesReadStartup > 0) //if there are bytes to read

{

printf("STARTUP\n");

while(readBufferStartup[bytesReadStartup-1]!='\n') //keep printing until new line

{ // continue printing until the new line is specified

readBufferStartup[bytesReadStartup] = '\0';

printf("%s", readBufferStartup);

bytesReadStartup = read(readStartupFd, readBufferStartup,READ\_BLOCK\_SIZE);

}

//once a new line detected, print the remaining part of the message

readBufferStartup[bytesReadStartup] = '\0';

printf("%s", readBufferStartup);

bytesReadStartup = read(readStartupFd, readBufferStartup, READ\_BLOCK\_SIZE);

}

// otherwise if all pipes are closed, terminate the Display process

else if (bytesReadStartup == 0 && bytesReadSim == 0 && bytesReadContrl == 0)

{

printf("DISPLAY\nFinished reading from pipes\nTerminating...\n");

close(readStartupFd); //close all the pipes and terminate

close(readSimFd);

close(readContrlFd);

exit(10);

}

// if there are no bytes from Startup, continue with program

//i.e. simulator has earlier simulation time

if(strncmp("Time", readBufferSim, 3) == 0 && strncmp("Time", readBufferContrl, 3) == 0)

{ //Do comparison between simulation time to determine which came first

//strncmp will return <0 if readBufferSim is lower in value to readBufferContrl for the first 20 chars

if(strncmp(readBufferSim, readBufferContrl, 20) < 0)

{

printf("SIMULATOR\n");

while (readBufferSim[bytesReadSim-1] != '\n')

{

readBufferSim[bytesReadSim] = '\0';

printf("%s", readBufferSim);

bytesReadSim = read(readSimFd, readBufferSim, READ\_BLOCK\_SIZE);

}

readBufferSim[bytesReadSim] = '\0';

printf("%s", readBufferSim);

bytesReadSim = read(readSimFd, readBufferSim, READ\_BLOCK\_SIZE);

}

//might need to add another byte read here to compare next message sim time

// strncmp will return >0 if readBufferSim is higher in value to readBufferContrl for the first 20 chars

// i.e. the controller will have earlier simulation time

if (strncmp(readBufferSim, readBufferContrl, 20) > 0)

{

printf("CONTROLLER\n");

while (readBufferContrl[bytesReadContrl-1] != '\n')

{

readBufferContrl[bytesReadContrl] = '\0';

printf("%s", readBufferContrl);

bytesReadContrl = read(readContrlFd, readBufferContrl, READ\_BLOCK\_SIZE);

}

readBufferContrl[bytesReadContrl] = '\0';

printf("%s", readBufferContrl);

bytesReadContrl = read(readContrlFd, readBufferContrl, READ\_BLOCK\_SIZE);

}

}

else

{ // if the message from simulator or controller does not contain 'Time' or simulation time, print as usual

if (bytesReadSim > 0)

{

printf("SIMULATOR\n");

while (readBufferSim[bytesReadSim-1] != '\n')

{

readBufferSim[bytesReadSim] = '\0';

printf("%s", readBufferSim);

bytesReadSim = read(readSimFd, readBufferSim, READ\_BLOCK\_SIZE);

}

readBufferSim[bytesReadSim] = '\0';

printf("%s", readBufferSim);

bytesReadSim = read(readSimFd, readBufferSim, READ\_BLOCK\_SIZE);

}

if (bytesReadContrl > 0)

{

printf("CONTROLLER\n");

while (readBufferContrl[bytesReadContrl-1] != '\n')

{

readBufferContrl[bytesReadContrl] = '\0';

printf("%s", readBufferContrl);

bytesReadContrl = read(readContrlFd, readBufferContrl, READ\_BLOCK\_SIZE);

}

readBufferContrl[bytesReadContrl] = '\0';

printf("%s", readBufferContrl);

bytesReadContrl = read(readContrlFd, readBufferContrl, READ\_BLOCK\_SIZE);

}

}

}//end while loop

} // end main

## Simulator

/\*

\*

\* pnpSim.c - simulates the pick and place machine operation

\*

\* This program creates a shared memory segment with the controller via a memory mapped file

\*

\* Platform: Any POSIX compliant platform

\* Intended for and tested on: Cygwin 64 bit

\*

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <semaphore.h>

#include "pnpSim.h"

int main(int argc, char \*argv[])

{

char Sim\_str\_array[145];

char \*strFromSim;

int writeSimToDisplayFd = atoi(argv[1]); // the file descriptor to write from Simulator to Display

sem\_t \*sem\_Sim = sem\_open("/sem\_Sim", 0);

sem\_t \*sem\_Startup = sem\_open("/sem\_Startup", 0);

sem\_t \*sem\_Contrl = sem\_open("/sem\_Contrl", 0);

PnP \*pnp;

PlacedPart placedPart[MAX\_NUMBER\_OF\_COMPONENTS\_TO\_PLACE];

double sim\_time = 0.0, instruction\_finish\_time = 0.0;

double x = HOME\_X, y = HOME\_Y, x\_target = 0.0, y\_target = 0.0, x\_preplace\_error = 0.0, y\_preplace\_error = 0.0, controller\_del\_x = 0.0, controller\_del\_y = 0.0;

double theta\_pick\_error[NUMBER\_OF\_NOZZLES] = {0.0, 0.0, 0.0}, controller\_theta = 0.0, theta\_actual[NUMBER\_OF\_NOZZLES] = {0.0, 0.0, 0.0};

int nozzle = CENTRE\_NOZZLE;

int nozzle\_down[NUMBER\_OF\_NOZZLES] = {FALSE, FALSE, FALSE};

int nozzle\_vacuum[NUMBER\_OF\_NOZZLES] = {FALSE, FALSE, FALSE};

int nozzle\_picked\_part[NUMBER\_OF\_NOZZLES] = {NO\_PICKED\_PART, NO\_PICKED\_PART, NO\_PICKED\_PART};

int instruction\_being\_executed = NO\_INSTRUCTION;

int number\_of\_placed\_parts = 0, number\_of\_dropped\_parts = 0;

int photo\_direction;

srand(time(0));

/\* initialize file for memory mapping \*/

int fd = open(MEMORY\_MAPPED\_FILE, (O\_CREAT | O\_RDWR), 0666);

if (fd < 0)

{

perror("creation/opening of memory mapped file failed");

exit(1);

}

ftruncate(fd, sizeof(PnP));

/\* map the file to memory \*/

pnp = (PnP \*)mmap(0, sizeof(PnP), (PROT\_READ | PROT\_WRITE), MAP\_SHARED, fd, (off\_t)0);

if (pnp == MAP\_FAILED)

{

perror("memory mapping of file failed");

close(fd);

exit(2);

}

/\* reset the pick and place machine\*/

resetPnP(pnp, sim\_time);

//wait for Startup to finish spawning other processes

sem\_wait(sem\_Startup);

sprintf(Sim\_str\_array, "Time: %7.2f Pick and place machine simulation started successfully!\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

const char nozzle\_name[3][10] = {"Left", "Centre", "Right"};

/\*

\* loop continuously until simulator is to quit

\* sleep for a short duration (dictated by POLL\_LOOP\_RATE)

\* on each loop

\*/

while (pnp -> quit == FALSE)

{

/\*

\* If there is no instruction currently being executed, this code checks whether there

\* is a new instruction pending from the controller, and if so, determines the instruction

\* finish time based upon the type of instruction and possibly the parameters of that instruction.

\*

\* It also signals that there is currently an instruction being executed back to the controller

\* so that the controller waits to issue any further instructions.

\*/

if (instruction\_being\_executed == NO\_INSTRUCTION)

{

int new\_instruction = pnp -> instruction\_to\_execute;

if (new\_instruction == LOAD\_PCB)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = LOAD\_PCB;

instruction\_finish\_time = sim\_time + PCB\_LOAD\_UNLOAD\_TIME;

sprintf(Sim\_str\_array, "Time: %7.2f PCB about to be loaded into pick and place machine\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

if (new\_instruction == UNLOAD\_PCB)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = UNLOAD\_PCB;

instruction\_finish\_time = sim\_time + PCB\_LOAD\_UNLOAD\_TIME;

sprintf(Sim\_str\_array, "Time: %7.2f PCB about to be unloaded\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

if (new\_instruction == MOVE\_HEAD)

{

x\_target = pnp -> instruction\_argument\_1;

y\_target = pnp -> instruction\_argument\_2;

if (nozzle\_down[LEFT\_NOZZLE] == FALSE && nozzle\_down[CENTRE\_NOZZLE] == FALSE && nozzle\_down[RIGHT\_NOZZLE] == FALSE)

{

if (x\_target >= MIN\_X && x\_target <= MAX\_X && y\_target >= MIN\_Y && y\_target <= MAX\_Y)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = MOVE\_HEAD;

instruction\_finish\_time = sim\_time + (double)sqrt(pow((x - x\_target), 2) + pow((y - y\_target), 2)) / HEAD\_FULL\_SPEED;

sprintf(Sim\_str\_array, "Time: %7.2f Head moving from (%.2f, %.2f) to (%.2f, %.2f)\n", sim\_time, x, y, x\_target, y\_target);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad MOVE\_HEAD command: destination out of range\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad MOVE\_HEAD command: one or more nozzles down\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else if (new\_instruction == ROTATE\_NOZZLE)

{

nozzle = pnp -> instruction\_argument\_3;

if (nozzle >= LEFT\_NOZZLE && nozzle <= RIGHT\_NOZZLE)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = ROTATE\_NOZZLE;

controller\_theta = pnp -> instruction\_argument\_1;

instruction\_finish\_time = sim\_time + (double)abs(controller\_theta) / NOZZLE\_ROTATE\_SPEED;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle being rotated by %.2f degrees\n", sim\_time, nozzle\_name[nozzle], controller\_theta);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad ROTATE\_NOZZLE command: nozzle out of range\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else if (new\_instruction == LOWER\_NOZZLE)

{

nozzle = pnp -> instruction\_argument\_3;

if (nozzle >= LEFT\_NOZZLE && nozzle <= RIGHT\_NOZZLE)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = LOWER\_NOZZLE;

instruction\_finish\_time = sim\_time + NOZZLE\_LOWER\_TIME;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle being lowered\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad LOWER\_NOZZLE command: nozzle out of range\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else if (new\_instruction == RAISE\_NOZZLE)

{

nozzle = pnp -> instruction\_argument\_3;

if (nozzle >= LEFT\_NOZZLE && nozzle <= RIGHT\_NOZZLE)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = RAISE\_NOZZLE;

instruction\_finish\_time = sim\_time + NOZZLE\_RAISE\_TIME;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle being raised\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad RAISE\_NOZZLE command: nozzle out of range\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else if (new\_instruction == APPLY\_VACUUM)

{

nozzle = pnp -> instruction\_argument\_3;

if (nozzle >= LEFT\_NOZZLE && nozzle <= RIGHT\_NOZZLE)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = APPLY\_VACUUM;

instruction\_finish\_time = sim\_time + VACUUM\_APPLY\_TIME;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle is about to apply vacuum\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad APPLY\_VACUUM command: nozzle out of range\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else if (new\_instruction == RELEASE\_VACUUM)

{

nozzle = pnp -> instruction\_argument\_3;

if (nozzle >= LEFT\_NOZZLE && nozzle <= RIGHT\_NOZZLE)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = RELEASE\_VACUUM;

instruction\_finish\_time = sim\_time + VACUUM\_RELEASE\_TIME;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle is about to release vacuum\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad RELEASE\_VACUUM command: nozzle out of range\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else if (new\_instruction == TAKE\_PHOTO)

{

photo\_direction = pnp -> instruction\_argument\_3;

if (photo\_direction == PHOTO\_LOOKUP || photo\_direction == PHOTO\_LOOKDOWN)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = TAKE\_PHOTO;

instruction\_finish\_time = sim\_time + PHOTO\_TAKE\_TIME;

if (photo\_direction == PHOTO\_LOOKUP)

{

sprintf(Sim\_str\_array, "Time: %7.2f Photo about to be taken by lookup camera\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Photo about to be taken by lookdown camera\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad TAKE\_PHOTO command: specified camera is not Lookup or Lookdown\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else if (new\_instruction == AMEND\_HEAD\_POSITION)

{

controller\_del\_x = pnp -> instruction\_argument\_1;

controller\_del\_y = pnp -> instruction\_argument\_2;

if (nozzle\_down[LEFT\_NOZZLE] == FALSE && nozzle\_down[CENTRE\_NOZZLE] == FALSE && nozzle\_down[RIGHT\_NOZZLE] == FALSE)

{

if (x + controller\_del\_x >= MIN\_X && x + controller\_del\_x <= MAX\_X && y + controller\_del\_y >= MIN\_Y && y + controller\_del\_y <= MAX\_Y)

{

pnp -> ready\_for\_next\_instruction = FALSE;

pnp -> instruction\_to\_execute = NO\_INSTRUCTION;

instruction\_being\_executed = AMEND\_HEAD\_POSITION;

instruction\_finish\_time = sim\_time + (double)sqrt(pow((controller\_del\_x), 2) + pow((controller\_del\_y), 2)) / HEAD\_FULL\_SPEED;

sprintf(Sim\_str\_array, "Time: %7.2f Head moving from (%.2f, %.2f) to (%.2f, %.2f)\n", sim\_time, x, y, x + controller\_del\_x, y + controller\_del\_y);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad AMEND\_HEAD\_POSITION command: destination out of range\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

else

{

sprintf(Sim\_str\_array, "Time: %7.2f Bad AMEND\_HEAD\_POSITION command: one or more nozzles down\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

}

}

/\*

\* If there is an instruction currently being executed, this code checks whether the

\* instruction has finished based upon the previously calculated instruction finish time.

\* If so, variables are updated based upon the type of the instruction that was executed

\* (e.g. x and y for MOVE\_HEAD).

\*

\* It then signals that there is currently no instruction being executed back to the controller

\* so that the controller can issue its next instruction if required.

\*/

else if (sim\_time >= instruction\_finish\_time)

{

int feeder;

switch(instruction\_being\_executed)

{

case LOAD\_PCB:

sprintf(Sim\_str\_array, "Time: %7.2f PCB has been loaded\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

break;

case UNLOAD\_PCB:

sprintf(Sim\_str\_array, "Time: %7.2f PCB has been unloaded\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

sem\_post(sem\_Sim); // the controller waits for the simulator to finish this task before terminating

break;

case MOVE\_HEAD:

x = x\_target;

y = y\_target;

sprintf(Sim\_str\_array, "Time: %7.2f Head arrived at nominal location (%.2f, %.2f)\n", sim\_time, x, y);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

break;

case ROTATE\_NOZZLE:

theta\_actual[nozzle] = theta\_actual[nozzle] + controller\_theta;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle finished rotating by %.2f degrees, effective rotation including misalignment theta\_error=%.2f degrees is %.2f degrees\n",

sim\_time, nozzle\_name[nozzle], controller\_theta, theta\_pick\_error[nozzle], theta\_actual[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

break;

case LOWER\_NOZZLE:

nozzle\_down[nozzle] = TRUE;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle lowered\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

/\* code for when part is being picked up from tape feeder \*/

feeder = getTapeFeederNumberAtLocation(x + (nozzle - CENTRE\_NOZZLE) \* NOZZLE\_X\_SEPARATION,y);

if (nozzle\_vacuum[nozzle] == TRUE

&& nozzle\_picked\_part[nozzle] == NO\_PICKED\_PART

&& feeder != NO\_TAPE\_FEEDER\_AT\_THIS\_LOCATION)

{

nozzle\_picked\_part[nozzle] = feeder;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle has picked up part from feeder %d\n", sim\_time, nozzle\_name[nozzle], feeder);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else if (nozzle\_vacuum[nozzle] == TRUE

&& nozzle\_picked\_part[nozzle] == NO\_PICKED\_PART)

{

sprintf(Sim\_str\_array, "Time: %7.2f No tape feeder underneath nozzle %s when vacuum applied so no part picked up\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

break;

case RAISE\_NOZZLE:

nozzle\_down[nozzle] = FALSE;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle raised\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

break;

case APPLY\_VACUUM:

nozzle\_vacuum[nozzle] = TRUE;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle now has vacuum applied\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

/\* code for when part is being picked up from tape feeder \*/

feeder = getTapeFeederNumberAtLocation(x + (nozzle - CENTRE\_NOZZLE) \* NOZZLE\_X\_SEPARATION,y);

if (nozzle\_down[nozzle] == TRUE

&& nozzle\_picked\_part[nozzle] == NO\_PICKED\_PART

&& feeder != NO\_TAPE\_FEEDER\_AT\_THIS\_LOCATION)

{

nozzle\_picked\_part[nozzle] = feeder;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle has picked up part from feeder %d\n", sim\_time, nozzle\_name[nozzle], feeder);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

else if (nozzle\_down[nozzle] == TRUE && nozzle\_picked\_part[nozzle] == NO\_PICKED\_PART)

{

sprintf(Sim\_str\_array, "Time: %7.2f No tape feeder underneath nozzle %s when vacuum applied so no part picked up\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

break;

case RELEASE\_VACUUM:

nozzle\_vacuum[nozzle] = FALSE;

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle now has vacuum released\n", sim\_time, nozzle\_name[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

/\* code for when part is being placed on PCB \*/

if (nozzle\_down[nozzle] == TRUE

&& nozzle\_picked\_part[nozzle] != NO\_PICKED\_PART

&& x >= 0.0 && y >= 0.0)

{

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle has placed part from feeder %d at (%.2f, %.2f) with rotation %.2f degrees\n",

sim\_time, nozzle\_name[nozzle], nozzle\_picked\_part[nozzle], x, y, theta\_actual[nozzle]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

placedPart[number\_of\_placed\_parts].x\_actual = x;

placedPart[number\_of\_placed\_parts].y\_actual = y;

placedPart[number\_of\_placed\_parts].theta\_actual = theta\_actual[nozzle];

placedPart[number\_of\_placed\_parts].feeder = nozzle\_picked\_part[nozzle];

number\_of\_placed\_parts++;

strFromSim = "\nSummary of placed parts so far:\n";

write(writeSimToDisplayFd, strFromSim, strlen(strFromSim));

for (int i = 0; i < number\_of\_placed\_parts; i++)

{

sprintf(Sim\_str\_array, "Part %d from feeder %d placed at (%.2f, %.2f) with rotation %.2f degrees\n", i,

placedPart[i].feeder, placedPart[i].x\_actual, placedPart[i].y\_actual, placedPart[i].theta\_actual);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

}

strFromSim = "\n";

write(writeSimToDisplayFd, strFromSim, strlen(strFromSim));

nozzle\_picked\_part[nozzle] = NO\_PICKED\_PART;

/\* reset pick and preplace alignment error values after part placed \*/

x\_preplace\_error = 0.0;

y\_preplace\_error = 0.0;

theta\_pick\_error[nozzle] = 0.0;

pnp -> x\_preplace\_error = x\_preplace\_error;

pnp -> y\_preplace\_error = y\_preplace\_error;

pnp -> theta\_pick\_error[nozzle] = theta\_pick\_error[nozzle];

theta\_actual[nozzle] = 0.0;

}

/\* code for when part is dropped from a height \*/

else if (nozzle\_down[nozzle] == FALSE

&& nozzle\_picked\_part[nozzle] != NO\_PICKED\_PART)

{

sprintf(Sim\_str\_array, "Time: %7.2f %s nozzle has DROPPED part from feeder %d at (%.2f, %.2f)\n", sim\_time, nozzle\_name[nozzle], nozzle\_picked\_part[nozzle], x, y);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

number\_of\_dropped\_parts++;

nozzle\_picked\_part[nozzle] = NO\_PICKED\_PART;

}

break;

case TAKE\_PHOTO:

/\* code for when lookup camera is used to take photos to discover pick misalignment \*/

if (photo\_direction == PHOTO\_LOOKUP && x == LOOKUP\_CAMERA\_X && y == LOOKUP\_CAMERA\_Y)

{

sprintf(Sim\_str\_array, "Time: %7.2f Photo taken by lookup camera\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

for (int i = 0; i < NUMBER\_OF\_NOZZLES; i++)

{

if (nozzle\_picked\_part[i] != NO\_PICKED\_PART)

{

theta\_pick\_error[i] = MAX\_THETA\_PICK\_MISALIGNMENT \* (double)rand()/RAND\_MAX - MAX\_THETA\_PICK\_MISALIGNMENT / 2;

theta\_actual[i] = theta\_pick\_error[i];

sprintf(Sim\_str\_array, "Time: %7.2f Picked part on %s nozzle has misalignment theta\_error=%.2f degrees\n", sim\_time, nozzle\_name[i], theta\_pick\_error[i]);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

pnp -> theta\_pick\_error[i] = theta\_pick\_error[i];

}

}

}

else if (photo\_direction == PHOTO\_LOOKDOWN && x >= 0.0 && y >= 0.0)

{

sprintf(Sim\_str\_array, "Time: %7.2f Photo taken by lookdown camera\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

x\_preplace\_error = MAX\_X\_PREPLACE\_MISALIGNMENT \* (double)rand()/RAND\_MAX - MAX\_X\_PREPLACE\_MISALIGNMENT / 2;

y\_preplace\_error = MAX\_Y\_PREPLACE\_MISALIGNMENT \* (double)rand()/RAND\_MAX - MAX\_Y\_PREPLACE\_MISALIGNMENT / 2;

sprintf(Sim\_str\_array, "Time: %7.2f Head has preplace misalignment x\_error=%.2f y\_error=%.2f\n", sim\_time, x\_preplace\_error, y\_preplace\_error);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

x = x + x\_preplace\_error;

y = y + y\_preplace\_error;

pnp -> x\_preplace\_error = x\_preplace\_error;

pnp -> y\_preplace\_error = y\_preplace\_error;

}

break;

case AMEND\_HEAD\_POSITION:

x = x + controller\_del\_x;

y = y + controller\_del\_y;

sprintf(Sim\_str\_array, "Time: %7.2f Head position amended to (%.2f, %.2f)\n", sim\_time, x, y);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

break;

}

/\* update shared memory for instruction related variables \*/

instruction\_being\_executed = NO\_INSTRUCTION;

pnp -> ready\_for\_next\_instruction = TRUE;

//sem\_post(sem\_Sim); // allowing the Controller to access the shared memory for next instruction

}

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

sim\_time += (double) 1 / POLL\_LOOP\_RATE;

/\* update shared memory for simulation time (since this must always be updated every poll cycle) \*/

pnp -> sim\_time = sim\_time;

}

// if program is terminated early, need to wait for controller to terminate first

sem\_wait(sem\_Contrl);

sprintf(Sim\_str\_array, "Time: %7.2f Terminating...\n", sim\_time);

write(writeSimToDisplayFd, Sim\_str\_array, strlen(Sim\_str\_array));

close(writeSimToDisplayFd);

/\* unmap memory and close file descriptor before exit \*/

sleep(1);

resetPnP(pnp, 0.0);

munmap(pnp, sizeof(PnP));

close(fd);

sem\_close(sem\_Startup);

sem\_close(sem\_Sim);

sem\_close(sem\_Contrl);

exit(20);

}

## Controller

/\*

\*

\* 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

\*

\*/

#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 PCB 21

#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[22][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 ",

"PCB "};

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(int argc, char \*argv[])

{

sleep(1); // give time for other processes to initialise

char Contrl\_str\_array[150];

int writeContrlToDisplayFd = atoi(argv[1]); // the file descriptor to write from controller to Display

sem\_t \*sem\_Startup = sem\_open("/sem\_Startup", 0); // open the named semaphores

sem\_t \*sem\_Sim = sem\_open("/sem\_Sim", 0);

sem\_t \*sem\_Contrl = sem\_open("/sem\_Contrl", 0);

pnpOpen(); // open the shared file with the simulator

sprintf(Contrl\_str\_array, "Time: %7.2f Pick and place controller started successfully!\n", getSimulationTime());

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

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);

}

// wait for startup to finish spawning processes and closing pipes

sem\_wait(sem\_Startup);

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

sprintf(Contrl\_str\_array, "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);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

/\* print details of part 0 \*/

sprintf(Contrl\_str\_array, "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);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

/\* 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 \*/

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

state = MOVE\_TO\_FEEDER;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

if(finished == TRUE)

{

sprintf(Contrl\_str\_array, "Time: %7.2f Terminating...\n", getSimulationTime());

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

close(writeContrlToDisplayFd);

pnpClose();

sem\_post(sem\_Contrl); // allow simulator to terminate

sem\_close(sem\_Sim);

sem\_close(sem\_Startup);

sem\_close(sem\_Contrl);

exit(30);

}

break;

case MOVE\_TO\_FEEDER:

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

if (isSimulatorReadyForNextInstruction())

{

state = WAIT\_1;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

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

{

lowerNozzle(CENTRE\_NOZZLE);

state = LOWER\_CNTR\_NOZZLE;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

//'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

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

//'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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

//'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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

// '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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

// 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 \*/

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s WARNING The next part is in feeder %d.\n", getSimulationTime(), state\_name[state], pi[part\_counter].feeder);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

state = MOVE\_TO\_FEEDER;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

//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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

sprintf(Contrl\_str\_array, "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);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

{

finished = TRUE;

setTargetPos(HOME\_X,HOME\_Y);

state = MOVE\_TO\_HOME;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

break;

case LOOK\_DOWN\_PHOTO:

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

takePhoto(PHOTO\_LOOKDOWN);

state = CHECK\_ERROR;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

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

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Part misalignment error: %3.2f, preplace misalignment error: x=%3.2f y=%3.2f\n", getSimulationTime(),state\_name[state], errortheta, getPreplaceErrorX(), getPreplaceErrorY());

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

break;

case MOVE\_TO\_HOME:

if (isSimulatorReadyForNextInstruction())

{

state = HOME;

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Gantry in Home position\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

break;

}

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

} //end while loop

} // 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, loaded = 1, PCB\_status = 0, unloaded = 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

sprintf(Contrl\_str\_array, "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);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

/\* 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];

sprintf(Contrl\_str\_array, "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);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

/\* 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)

{ // program is complete, terminate program

sem\_wait(sem\_Sim); // waiting for the simulator to finish unloading the PCB

sprintf(Contrl\_str\_array, "Time: %7.2f Terminating...\n", getSimulationTime());

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

close(writeContrlToDisplayFd);

pnpClose();

sem\_post(sem\_Contrl); // allow the simulator to terminate

sem\_close(sem\_Sim);

sem\_close(sem\_Startup);

sem\_close(sem\_Contrl);

exit(30);

}

else if(PCB\_status == loaded)

{//program has cycled back around, go to the next feeder, +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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

else if (PCB\_status == unloaded)

{ // if the PCB is not currently in the machine, then load

loadPCB();

state = PCB;

PCB\_status = loaded;

sprintf(Contrl\_str\_array, "Time: %7.2f New State: %.15s Loading PCB onto pick and place machine\n\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

break;

case PCB:

if(isSimulatorReadyForNextInstruction())

{

if(PCB\_status == loaded)

{//once PCB is loaded, 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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

else if(PCB\_status == unloaded)

{ // if the PCB has just been unloaded then program is complete, go to HOME to terminate

state = HOME;

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s PCB unloaded successfully\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

{ // centre nozzle picks up part after left nozzle

lowerNozzle(CENTRE\_NOZZLE);

state = LOWER\_CNTR\_NOZZLE;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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

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;

sprintf(Contrl\_str\_array, "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);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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

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;

sprintf(Contrl\_str\_array, "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);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

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

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

if (isSimulatorReadyForNextInstruction())

{

setTargetPos(HOME\_X,HOME\_Y);

state = MOVE\_TO\_HOME;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

else

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

state = HOME;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

break;

case LOOK\_UP\_PHOTO:

if (isSimulatorReadyForNextInstruction())

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

lookup\_photo = TRUE;

state = CHECK\_ERROR;

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Checking errors and calculating corrections\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

takePhoto(PHOTO\_LOOKDOWN);

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

break;

case LOOK\_DOWN\_PHOTO:

if (isSimulatorReadyForNextInstruction())

{

lookdown\_photo = TRUE;

state = CHECK\_ERROR;

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Look-down photo acquired. Calculating corrections\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Right part misalignment error: %3.2f Correction required: %3.2f degrees\n", getSimulationTime(), state\_name[state], errortheta, requested\_theta\_right);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

state = FIX\_NOZZLE\_ERROR;

rotateNozzle(RIGHT\_NOZZLE, requested\_theta\_right);

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Correcting right nozzle rotation...\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Centre part misalignment error: %3.2f Correction required: %3.2f degrees\n", getSimulationTime(),state\_name[state], errortheta, requested\_theta\_centre);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

state = FIX\_NOZZLE\_ERROR;

rotateNozzle(CENTRE\_NOZZLE, requested\_theta\_centre);

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Correcting centre nozzle rotation...\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Left part misalignment error: %3.2f Correction required: %3.2f degrees\n", getSimulationTime(),state\_name[state], errortheta, requested\_theta\_left);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

state = FIX\_NOZZLE\_ERROR;

rotateNozzle(LEFT\_NOZZLE, requested\_theta\_left);

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Correcting left nozzle rotation\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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;

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s No further errors. Moving to PCB\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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

//sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Preplace misalignment error: x=%3.2f y=%3.2f\n", getSimulationTime(), state\_name[state], getPreplaceErrorX(), getPreplaceErrorY());

//write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

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

state = FIX\_PREPLACE\_ERROR;

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Correcting gantry position...\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

break;

case FIX\_NOZZLE\_ERROR:

if (isSimulatorReadyForNextInstruction())

{

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

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

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

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

state = CHECK\_ERROR;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

{ //centre nozzle is second to be corrected

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

nozzle\_errors\_to\_check--;

state = CHECK\_ERROR;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

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

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

nozzle\_errors\_to\_check--;

state = CHECK\_ERROR;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

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;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

else if (Centre\_NozzleStatus == holdingpart)

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

lowerNozzle(CENTRE\_NOZZLE);

state = LOWER\_CNTR\_NOZZLE;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

else if (Right\_NozzleStatus == holdingpart)

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

lowerNozzle(RIGHT\_NOZZLE);

state = LOWER\_RIGHT\_NOZZLE;

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

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

}

break;

case MOVE\_TO\_HOME:

if (isSimulatorReadyForNextInstruction())

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

unloadPCB(); // then board can be unloaded from the machine

state = PCB;

PCB\_status = unloaded;

sprintf(Contrl\_str\_array, "Time: %7.2f New state: %.20s Gantry in Home position. Unloading PCB\n", getSimulationTime(), state\_name[state]);

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

}

break;

} //closing switch

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

}//closing while loop

}

// if program is quit early, the controller needs to terminate before simulator to prevent program hanging

sprintf(Contrl\_str\_array, "Time: %7.2f Terminating...\n", getSimulationTime());

write(writeContrlToDisplayFd, Contrl\_str\_array, strlen(Contrl\_str\_array));

close(writeContrlToDisplayFd);

pnpClose();

sem\_post(sem\_Contrl); // now allow the simulator to terminate

sem\_close(sem\_Startup);

sem\_close(sem\_Sim);

sem\_close(sem\_Contrl);

exit(30);

}