|  |  |
| --- | --- |
| \\cetus.ece.missouri.edu\users\fischerjd\Desktop\Current Semester\common\logo-clear.png | University of Missouri – Columbia  Department of Electrical and Computer Engineering  *“A Proud Tradition, Engineering the Future!”* |

Simon Says Cross Network

ECE 4220: Real time Embedded computer

May 12, 2015

— Engineer—

Adam Speichinger, Electrical and Computer Engineering

— Course Instructor —

Luis Rivera

## Abstract

The concept of this project is to create a multiplayer game where the players of the game will interact with each other over a Local Area Network (LAN). The game is a version of the classic Simon Says, where a player will input a string of valid characters and then their opponent will need to respond with the exact same string. The game is aimed at challenging the user’s memory skills by flashing the player’s submitted string across the opponent’s screen rapidly, and then forcing the opponent to relay the exact string. This game has been played for centuries, but is well known as the popular electronic toys of the 80s. Ralph H. Baer, the created of this 80s game, was inspired by the age old game and created the harmonic musical tones of a bugle and a little-known Atari arcade machine ‘Touch-Me’ that worked along similar lines [1]. With this version of Simon Says, it was set out to be an open source scalable multiplayer game where many different players could play against each other or simply just play against a computer simulation in order to improve their memory skills without worrying about their opponents skill level.

**Index Terms: UDP/TCP Socket Connection, Multiplayer Games, USB Serial Communication, Master-Slave Relation**

**.**

Table of Contents

[Abstract 2](#_Toc419429638)

[Chapter 1.  Introduction 6](#_Toc419429639)

[Chapter 2.  Background 7](#_Toc419429640)

[Chapter 3.  Implementation 8](#_Toc419429641)

[Proposed Implementation 8](#_Toc419429642)

[Actual Implementation 9](#_Toc419429643)

[Chapter 4.  13](#_Toc419429644)

[Chapter 5.  Experiment and Results 14](#_Toc419429645)

[Chapter 6.  Discussion and Conclusions 15](#_Toc419429646)

[Discussion 15](#_Toc419429647)

[Conclusion 15](#_Toc419429648)

[Chapter 7.  Appendices 16](#_Toc419429649)

[Purpose 16](#_Toc419429650)

[Bibliography 17](#_Toc419429651)

Table of Figures

[Figure 1: Proposed Implementation 8](#_Toc419422912)

[Figure 2: Actual Implementation 10](#_Toc419422913)

1. Introduction

The objective of this project is to be able to run a multiplayer game that will attempt to improve a user’s memory skills unconsciously as they play along with friends or against a computer simulation. The game will create a competitive environment between friends to challenge each other’s minds and push each other’s memory skills as the game goes on. The focus is to continue increasing the size of the array that the players will use in order to improve on the number of items they can remember in a short time span. Studies have shown that a typical adult’s memory span is approximately between 5 and 9 unrelated items, with a ‘magic number’ of 7 items [2]. With this in mind the game will start off easy for the players so they can gain experience with the game, and then increase with each correct opponent input. Of course since it would be unfair if only one player was able to input characters to the array, the game will switch masters after every even round to keep all players involved equally.

1. Background

As mentioned previously, Simon Says has been around for quite some time and there is always a new version coming out every year. A big difference between this version and the average previous version of Simon Says is that this was intended on being a scalable multiplayer game, that wasn’t necessarily restricted to only two players. When set out to complete this project, the goal was to allow several players to play against each other and see who could be the most successful. The game is mainly used as an entertainment tool by using visual effects to draw users into buying/playing the game. Unlike those versions, this game was meant to help users improve their short-term memory skills as well as indulge in some friendly competition. There aren’t any special visual effects to the game currently, but there is always a possibility that in future instances of the game there could be a Graphic User Interface (GUI) implemented in order to provide those visual effects.

1. Implementation

## Proposed Implementation

The original design implementation was to use a Raspberry Pi to host the game and connect to the TS-7250 board using a UDP connectionless protocol across the LAN. The Raspberry Pi would use a gaming controller in order to play the game, while the TS-7250 board was to use a keyboard. The reason for this was the cost of purchasing multiple Raspberry Pis and multiple gaming controllers. The assumption was that once it was working on the first Raspberry Pi that it wouldn’t be challenging to implement the UDP server using the same format as the client program. The Raspberry Pi would interact with the controller using USB serial communication where a button press would be read into the program and then broadcasted to the server program where it would convert the button push to corresponding keyboard buttons as seen in **Figure** **1**.

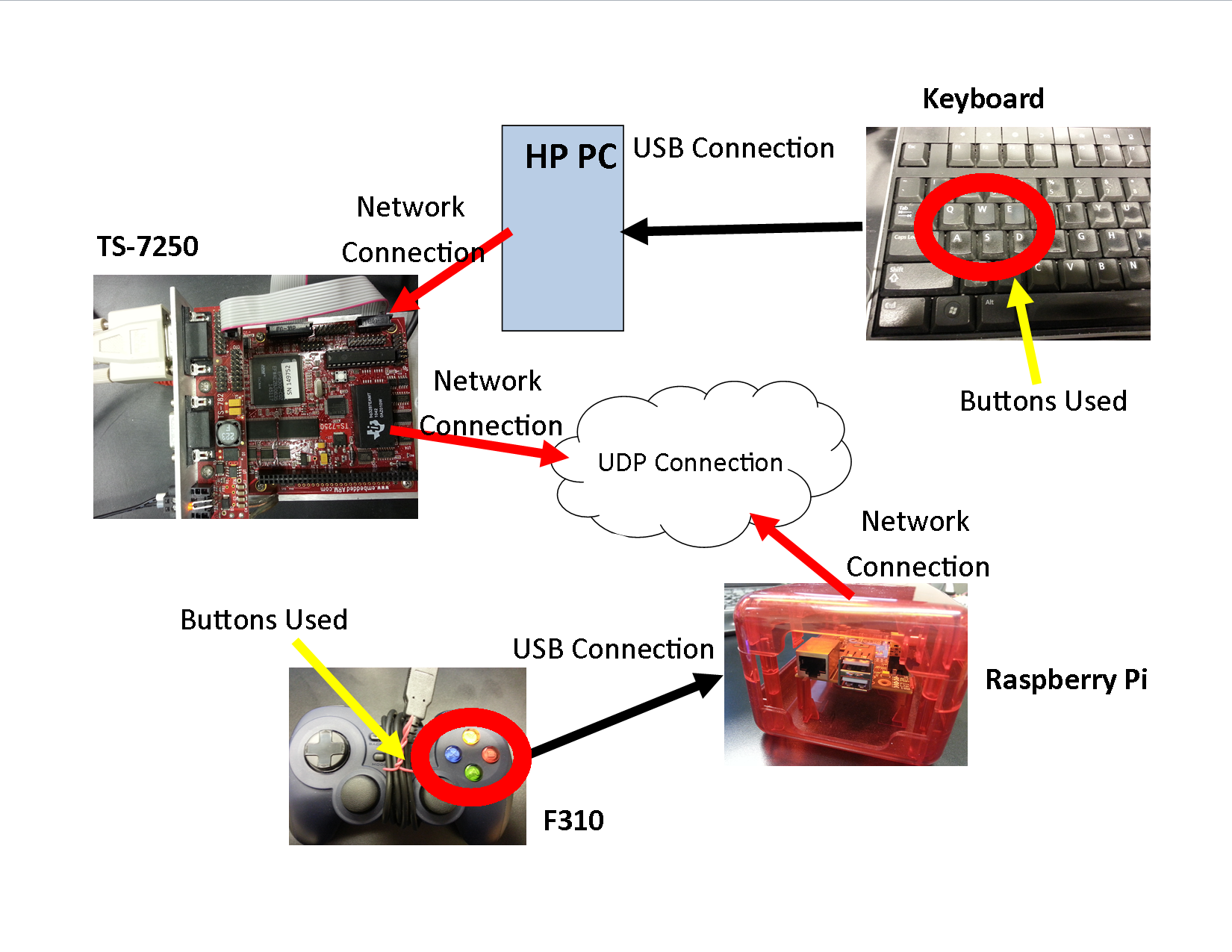


Figure : Proposed Implementation

As for the actual game itself, the very first thing to implement was the menu to run the game. This menu was what was going to define the rest of the game since the options provided to the user would need to be implemented. The goal was to create two setting for the game; the first was where the player could play a simulated version of the game against a computer, and the second was where the player would play against other users. Of course there are some things that were overlooked in the proposal that didn’t get noticed till actual implementation, but that will be discussed later.

## Actual Implementation

Unfortunately, not all the proposed implementation was implemented into the project due to unforeseen restrictions and limited time. The actual implementation was only able to use two TS-7250 boards using a keyboard inputs. The F310 controller was originally meant to be a Microsoft Windows product, but there were kernel modules out for the Raspberry Pi that would still read in the buttons using the same USB protocol. After much attempt at implementing these kernels, none seemed to work properly for the controller and the F310 controller wasn’t able to be used. Another issue that I ran into was that the NFS1 network does not allow new machines to just simply be plugged in. There is a firewall that stops any machine from just plugging into the network and obtaining an IP address, so the Raspberry Pi was not able to be used on the NFS1 server. These unforeseen problems are what led to having to use two TS-7250 boards and keyboards rather than the proposed implementation (see **Figure 2**).

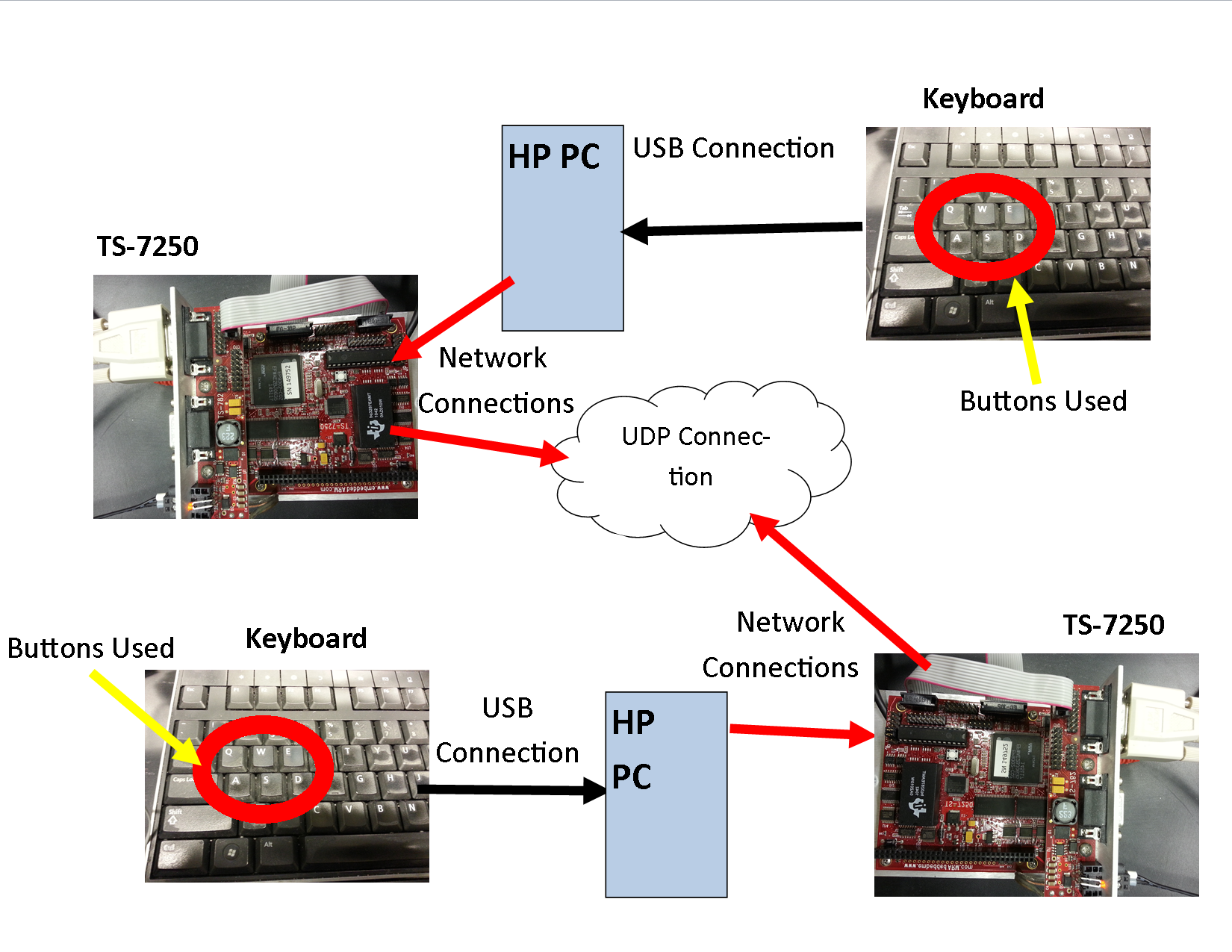


Figure : Actual Implementation

The software on the other hand was able to be implemented as proposed. I still had some complications with fully implementing everything I wanted to. The reason I had issues though was because I had scope creeps. Meaning that once I started implementing the game I wanted to add more and more features to the game and by adding more features, the more issues I ran into. The way that I went about implementing the first function of the game, playing against a computer, was to first set up how the user would interact if they were the master. First the user would be prompted to input a pattern with the correct amount of letters, and after checking that they used valid letters and had the appropriate length a simulated computer function would go about guessing the correct pattern. This function had a 70 percent probability rate of guessing the correct pattern. The reason for the 70 percent is since it was a computer the user was playing against it should be challenging to beat the computer but at the same time not impossible. If the computer was successful then the length of the pattern would be increased, once the length of the pattern had gone past 12 letters then the length would be reset to three and a new valid letter would be allowed. This is to continue until all six valid letters (A, S, D, W, Q, E) had been used. If all six letters were in the pattern and the length surpassed 12, the user would win the game. After every even round though the master should be switched up in order to provide a fair challenge.

This leads to the second half of the first feature. If the user is not the master then the computer would be required to simulate a random pattern, flash it on the screen briefly to the user, and then wait for the user to respond with what they saw. The random pattern generated by the computer was implemented by creating random probabilities for the valid letters each time the computer had to implement a pattern. The random numbers for the probabilities would not be higher than the length of the pattern though to give all valid letters an opportunity to be used. The first letter of the computer’s pattern would be the highest probable valid letter, after placing the letter into the pattern the probability for that letter would then be reduced by one. After reducing the probability it would look for the highest probability again and repeat the process until it had filled up the pattern. Once the computer generated it’s pattern it would prompt the user to input a response, check if the response was correct or not. If it was correct then the game would continue as described earlier where the length of the pattern would increase by one, after a 12 letter pattern a new valid letter would be introduced and the pattern length would be reset to 3. Also the same rule about after every even turn, the master would be switched, going back to the first half of the first feature. Again the game would continue like this until a winner was chosen.

The second feature of the game is the multiplayer aspect. This implementation of the game is where the real purpose of the game came into play. By using UDP connectionless socket communication, the players are able to pass patterns back and forth to each other following the same functionality as the player versus computer implementation. The main reason the computer implementation came first was because the algorithm was originally designed to work on my Raspberry Pi which I was testing at my home and couldn’t set up a connection to another board easily so the computer simulated player aspect came about as an added idea to the game. So just like playing against the computer the users would try to input a pattern that would be flashed across the screen of the opponent and then they would have to mimic the pattern in order to continue the game. After every even turn the master that chooses the pattern was switched up from player to player, and after every correct guess at the opponents pattern then there would be an increase in the length of the pattern. I didn’t get to fully implement this multiplayer game since I had some complications originally and then didn’t have much time to implement this functionality properly.

1. Experiment and Results

Since my project was all software related, my experiments mainly involved going through the code and trying to break it in some way or looking at how to better implement functions. There really isn’t much to report as far as the experimentation part of the project goes. As for results though, the project was not as successful as I had hoped it would be by this point. The computer simulated aspect of the game was not able to compare the user generated pattern to the computer generated pattern well, so it would always end up showing false results and never moved on from that. This was kind of a big part of the computer versus player case and it wasn’t able to be implemented correctly. After multiple attempts at creating algorithms that would properly grasp each letter or even compare the whole string I was unsuccessful. Something that I thought was successful to the computer portion of the game was randomly generating a pattern based upon a random probability ratio of each letter. For the multiplayer aspect of the game, the resulting code doesn’t fully work. Sometimes it sends the signal across but sometimes it won’t. Also another issue is when reading in a string it doesn’t eliminate the stdin buffer at the moment. I know how to fix this now, but since I don’t have time to get the code and demonstration changed, it’s staying how it is for now. Overall, the code still needs some work, but since this I’ve wanted to create and maintain my own game for a while, I will still be working on this even after the course has passed.

1. Discussion and Conclusions

## Discussion

So first off, I had a really hard time implementing my client broadcast program. For some reason when I went to look at the example code about UDP algorithms and how to set it up, the udp\_client\_broadcast file was not in the folder or I glanced over it. Anyway after talking with Michael Butler about what some of my mistakes was the algorithm was fixed for the most part about sending and receiving data. I’m still not entirely sure how the code could send and receive fine one moment, but then have issues sending the next. There may be some weird bug in the code that I was overlooking when writing it up. The server program of the project wasn’t that bad, truthfully I should have just waited to implement the server after the client. I think somewhere along the late nights of coding that my logic switched up on me and I started going at it a different route than originally set out to be. Not that the current system won’t be able to be implemented correctly after a few adjustments, it’s just that it would have been easier if the logic was symmetrical from the beginning and didn’t cause issues with the algorithm.

As for the project itself, I think it’s a bit unfair to be asking the undergraduate students to do all 6 labs, this project, and the final. Most undergraduate students have more than 3 classes and some, such as me, have 5 classes to juggle. While the homework and labs are not that stressful to work on and finish, this class was not the only one that required projects. I had a group mini research project for my Intro to Nuclear Physics course and a group Web App for my Software Engineering (CS) course. This project might have been able to be implemented either by cutting some of the labs short or forcing them to be taught in an earlier course. Another option for the project which has already been implemented before is to make it an option instead of taking the final exam for undergraduates. That way those who have time to work on a project can work on the project, but those who have a busy semester would only have to worry about taking the final at the end of the semester rather than spending a whole month working on a project.

## Conclusion

Overall I like my project idea and will continue to work on the game myself. I believe I am close to fixing the issues that I am having with reading and comparing, maybe with more available time to work on the project I could have gotten it all to work. I am satisfied with where I got so far though since overall it was a lot of code to implement and make sure it all worked based off read in data and or assumed variables based on what the user input. One function that really frustrated me though was my printSimon( ) function that was meant to print out the pattern to your opponents screen. The plan was to have this function ‘flash’ a letter of the pattern then replace it with an X and continue this leaving the X’s on the screen. No matter how long I stalled it would either flash the letter so quickly it was unnoticeable or the X was just replaced with it and the ‘flashing’ of the letter was non-existent. The project was pretty fun though at times, it was just really frustrating with not being able to use the Pi like I wanted to. The controller I bought was not as easy to implement with the Pi like I would have liked. Also the fact that the Firwall on the NFS1 server prevents other devices from accessing the network via Ethernet connections became a major let down to my project and I had to resort to using only the TS-7250 boards.

1. Appendices

## Project.c (Server)

/\*

\* Project.c

\*

\* Created on: May 12, 2015

\* Author: ads6g7

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <sys/ioctl.h>

#include <netinet/in.h>

#include <net/if.h>

#include <time.h>

#define SIZE 40 //defines the max message size

#define LOCALIP localIP //defines the local IP address, just a variable

#define MAXLENGTH 12 //defines the max length of the game arrray

#define PORTNUM 5050 //defines the port number, makes it easier to change if in one location

//function that will take in a message, output the message and error then exit the program.

//Parameters:

// const char \*msg -> message that is to be printed out

//Returns:

// Void

void error(const char \*msg)

{

perror(msg);

exit(0);

}

//debug function that is used to read in a message and variable to print out for debugging purposes

//Parameters:

// const char \*msg -> this is the message that will always be printed out

// const char \*variable -> this won't be printed if NULL, will be printed otherwise

//Returns:

// Void

void debug(const char \*msg, const char \*variable){

// //if the variable is set to NULL then only print out the message

// if(variable==NULL){

// printf("DEBUG :: %s\n", msg);

// }

// //if the variable is not set to NULL then print out the message followed by the variable

// else{

// printf("DEBUG :: %s %s\n",msg,variable);

// }

}

//trimwhitespace is a function that will read in a sting and eliminate any spaces that are found.

//This was useful when trying to decifer some of the read in messages that didn't have any spaces.

//Parameters:

// char \*str -> the string that needs blank spaces eliminated

//Returns:

// char\* -> returns the newly created spaceles string

char \*trimwhitespace(char \*str)

{

char \*end;

// Trim leading space

while(isspace(\*str)) str++;

if(\*str == 0) // All spaces?

return str;

// Trim trailing space

end = str + strlen(str) - 1;

while(end > str && isspace(\*end)) end--;

// Write new null terminator

\*(end+1) = 0;

return str;

}

//This function is meant to print out the pattern to the opponent by flashing each letter,

//waiting a while and then backspacing and prints out an X to mask the pattern as it goes.

//Parameters:

// char \*array -> the array that is to be printed out

// int length -> the length of the array that needs printing

//Returns:

// Void

void printSays(char \*array, int length){

//NEEDS WORK: I've tried to do this a few different ways and even looked up several ways to do this,

// many have said to use the printf("\033[2D") method which would back track the cursor

// two positions then write over that. The second way is to just backspace over the letter

// and then rewrite the mask to terminal. When I try doing either method, it doesn't print out

// the pattern and will instead rapidly switch to the X. Couldn't get this fully working yet

int x,y;

for(x=0;x<length; x++){

printf("%c ",array[x]);

for(y=0;y<100000000;y++){ }

//printf("\033[%dD",1);

printf("\b\b");

for(y=0;y<100000;y++){ }

printf("X ");

}

}

//CheckAnswer tokenizes the opponents input and players input, then compares the two strings,

//if at any point the two don't match up then return 0. If they do match, then return 1.

//Parameters:

// char \*OppInput -> the opponents pattern

// char \*PlayerInput -> the players input pattern

// int length -> the length of both patterns

//Returns:

// integer -> this integer returns 1 if correct answer, 0 if wrong answer

int CheckAnswer(char \*OppInput, char \*PlayerInput,int length){

int x;

char \*O, \*P;

strtok(OppInput, " ");

for(x=0;x<length;x++){

O = strtok(OppInput," ");

P = strtok(PlayerInput, " ");

debug("DEBUG :: O = ",O);

debug("DEBUG :: P = ",P);

if(P != O){

debug("DEBUG :: P and O don't match", NULL);

return 0;

}

}

return 1;

}

//function that just prints out the instructions when prompted to

//Parameters:

// None

//Returns:

// Void

void printInstructions(){

printf("\n\n\n\*\*\*\*\*\*\*\*\*\*\*Welcome to Adam Says\*\*\*\*\*\*\*\*\*\*\*\n\n\n");

printf("This is a simple memory game that will test your memory skills and push them to \ntheir limits.\n\n");

printf("Once a game has been started you will use the following keys in order to play \n(CASE SENSITIVE): A, W, S, D. ");

printf("At the start of the game, you will only be \nallowed to enter 3 of the allowed letters and then ");

printf("if your opponent gets them \nright, you can enter 4 allowed letters. If your opponent gets those right,\n");

printf("then it will switch to the opponent inputting allowed letters as well as \nincrementing to 5 letters\n");

printf("\nThe pattern will continue this way switching back and forth after every even \nround (i.e. after 4, 6, 8, 10, etc.)");

printf(" until you or your opponent enters a \ncommand wrong.\n\n");

printf("If a wrong pattern is not input after using 12 letters, the game will reset \nthe pattern counter back to 3 ");

printf("and then add another valid letter, which \nwill only be seen once you've reached this point\n\nGood Luck!!\n");

}

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

time\_t t; //sets up a time\_t structure to use for random generator

srand((unsigned) time(&t)); //sets up random generator based upon current time of day

int sock, sendReceive; //integer values used to determine if some functions were successful

int flag = 1; //flag used in setting up sockopt

socklen\_t fromlen; //used to store the length of the socket

struct sockaddr\_in server; //structure used for sending data to the server

struct sockaddr\_in addr; //structure used for receiving data from the client

struct ifreq ifr; //used to dynamiclly get the IP address

char localIP[10]; //used to stored the local IP address

u\_char hostname[50]; //stores the hostname

int MasterFlag = 0; //used as a boolean value to determine if I am the master or not

int game = 1; //used as a boolean value to determine if the game needs to run or not

int turn = 3; //used to keep track of the number of letters in the pattern (length of pattern)

int verify; //used as a boolean value to verify if something was right or not

char \*cbuffer = malloc(sizeof(char)\*SIZE); //buffer pointer that will store any character data

char \*gameArray = malloc(sizeof(char)\*MAXLENGTH); //this pointer is used to store the game pattern

char \*garbage = malloc(sizeof(char\*)\*SIZE); //another buffer but takes in non useful data

//creates the socket for connectionless communication

sock = socket(AF\_INET, SOCK\_DGRAM, 0);

//checks if the socket was created

if(sock < 0){

error("Error with opening socket");

}

//gets the length of the sockaddr\_in structure for use later

fromlen = sizeof(struct sockaddr\_in);

//sets all bits of the server sockaddr\_in variable to zero

bzero(&server, sizeof(server));

//symbol constant for Internet domain

server.sin\_family = AF\_INET;

//this allows reading from any server that is broadcasting

server.sin\_addr.s\_addr = INADDR\_ANY;

//sets the port by converting it to a network byte order

server.sin\_port = htons(PORTNUM);

//binds the socket together with the client and checks that it was bound correctly

if(bind(sock,(struct sockaddr \*)&server, sizeof(server)) < 0){

error("Error with binding\n");

}

//sets up the socket option to be broadcast

if(setsockopt(sock, SOL\_SOCKET, SO\_BROADCAST, &flag, sizeof(flag)) < 0){

error("Error with setting up socket options\n");

}

//used to dynamically get the local IP address

ifr.ifr\_addr.sa\_family = AF\_INET;

snprintf(ifr.ifr\_name, IFNAMSIZ, "eth0");

ioctl(sock, SIOCGIFADDR, &ifr);

sprintf(localIP, "%s", inet\_ntoa(((struct sockaddr\_in \*)&ifr.ifr\_addr)->sin\_addr));

//gets the host IP address and checks that it worked

if(gethostname(hostname, sizeof(hostname))!=0){

error("gethostname fail\n");

}

//prints out the instructions for the user

printInstructions();

//sets the cbuffer values to zero

bzero(cbuffer, SIZE);

while(1){

//receives the message that was broadcasted

sendReceive = recvfrom(sock, cbuffer, SIZE, 0, (struct sockaddr \*)&addr, &fromlen);

if(sendReceive < 0){

error("Error with recvfrom\n");

}else{

debug("MESSAGE WAS -> ", cbuffer);

break;

}

}

int val = 1; //integer value that acts as a boolean value for the following loop

//loop that waits till the user has entered valid data

while(val){

printf("Do you still want to play (Y or N)? ");

char \*answer = malloc(sizeof(char\*)\*SIZE);

fgets(answer,sizeof(answer),stdin);

//if the user entered n or N then exit the program

if(answer[0] == 'N' || answer[0] == 'n'){

printf("I'm sorry you feel that way, goodbye.\n\n");

return 0;

//if the user entered y or Y then prompt them to send a signal back to the client to start

}else if(answer[0] == 'Y' || answer[0] == 'y'){

printf("Type '$' and hit enter when ready to play\n");

fgets(garbage, 3, stdin);

garbage = trimwhitespace(garbage);

debug("garbage = ", garbage);

//if the user entered '$' then send the message "$ localIP"

if(strncmp(garbage, "$", 1)==0){

bzero(cbuffer, SIZE);

sprintf(cbuffer, "$ %s", localIP);

cbuffer[SIZE - 1] = '\0';

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255");

debug("issue here : cbuffer = ", cbuffer);

//sends out a message that was setup previously by sprintf

sendReceive = sendto(sock, cbuffer, SIZE, 0,(struct sockaddr\*)&addr, fromlen);

if(sendReceive >= 0){

debug("Sent successful ", cbuffer);

val= 0;

break;

}else{

error("Invalid commands using sendndto\n");

}

}

//if the user did not enter valid date then have them retry

}else{

printf("Not a valid input! Try again!\n");

}

}

//start up the game, this loop runs until game is over

while(game){

//sets the cbuffer bits to zero everytime, essentially clearing it

bzero(cbuffer, SIZE);

//receives the message that was broadcasted, prints error message and exits if invalid message

sendReceive = recvfrom(sock, cbuffer, SIZE, 0, (struct sockaddr \*)&addr, &fromlen);

if(sendReceive < 0){

error("Error with recvfrom\n");

}

//prints out on the terminal what was received

debug("Received the message: ", cbuffer);

//if the user is the master then do the following

if(MasterFlag){

//if the

if(turn != 0){

//prompt the user to enter a pattern then read the pattern into gameArray

printf("Enter %d characters now: ",turn);

fgets(gameArray,SIZE,stdin);

debug("You entered: ", gameArray);

//setup the string to be sent where string should be "@ gameArray"

bzero(cbuffer,SIZE);

strcat(cbuffer, "@");

sprintf(cbuffer, " %s", gameArray);

cbuffer[SIZE-1] = '\0';

//This sends cbuffer out to all boards on port PORTNUM

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255");

sendReceive = sendto(sock, cbuffer, SIZE, 0, (struct sockaddr \*)&addr, fromlen);

if(sendReceive < 0){

error("Error with sendto1\n");

}else{

debug("Sent successfully: ", cbuffer);

}

}

//if the turn is divisible by 2 then switch up the master

else if(turn%2==0){

MasterFlag = 0;

printf("Switching up the master...\n");

//sets up cbuffer to send out "!!SWITCH!!" which is the signal to switch master

bzero(cbuffer,SIZE);

sprintf(cbuffer, " %s", "!!SWITCH!!");

cbuffer[SIZE-1] = '\0';

//This sends cbuffer to all on port PORTNUM

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255");

sendReceive = sendto(sock, cbuffer, SIZE, 0, (struct sockaddr \*)&addr, fromlen);

if(sendReceive < 0){

error("Error with sendto1\n");

}else{

debug("Sent successfully: ", cbuffer);

}

}else{

if(cbuffer[0] == '#'){

verify = CheckAnswer(cbuffer, gameArray, turn);

if(verify == 1){

printf("Your opponent responded correctly...Try again\n");

printf("Enter %d characters now: ",turn);

fgets(gameArray,SIZE,stdin);

debug("You entered: ", gameArray);

//setup cbuffer to be sent out with "@ gameArray"

bzero(cbuffer,SIZE);

strcat(cbuffer, "@");

sprintf(cbuffer, " %s", gameArray);

cbuffer[SIZE-1] = '\0';

//This sends cbuffer after the input has been made\*\*\*\*\*\*\*\*\*\*

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255");

sendReceive = sendto(sock, cbuffer, SIZE, 0, (struct sockaddr \*)&addr, fromlen);

if(sendReceive < 0){

error("Error with sendto1\n");

}else{

debug("Sent successfully: ", cbuffer);

}

}else if(verify==0){

printf("CONGRATULATIONS!!!\nYOU WON!!!\n\n");

printf("Returning to Menu\n\n");

game = 0;

break;

}

}else{

debug("Error with '#' statement. cbuffer = ", cbuffer);

}

}

//if users is not the master then look at what message is being read in

}else if(!MasterFlag){

//if message is recieved that begins with '@', do the following

if(cbuffer[0] == '@'){

//print out the pattern letter by letter while masking previous letters

printSays(cbuffer, turn);

//printSays doesn't exactly work, usually this print statement is a debug statent

printf("opponent entered -> %s", cbuffer);

//prompt the user for their response

printf("\nResponse: ");

fgets(gameArray, SIZE, stdin);

debug("you entered -> ", gameArray);

//set up your response in cbuffer

bzero(cbuffer,SIZE);

strcat(cbuffer, "#");

sprintf(cbuffer, " %s", gameArray);

cbuffer[SIZE-1] = '\0';

//This sends cbuffer to all on port PORTNUM

addr.sin\_addr.s\_addr = inet\_addr("10.3.52.255");

sendReceive = sendto(sock, cbuffer, SIZE, 0, (struct sockaddr \*)&addr, fromlen);

if(sendReceive < 0){

error("Error with sendto1\n");

}else{

debug("Sent successfully: ", cbuffer);

}

//if cbuffer is equal to "!!SWITCH!!" then reset master flag high(true)

}else if(strncmp(cbuffer,"!!SWITCH!!", 10)==0){

MasterFlag = 1;

}

}

//increment the turn(length of the pattern)

turn++;

}

return 0;

}

## Raspberry.c (Client)

/\*

\* Raspberry.c

\*

\* Created on: May 12, 2015

\* Author: ads6g7

\* Description: This is going to be the client UDP program that will be ran on the Raspberry Pi.

\* This program will also initiate the game, waiting for the opponent to respond before

\* starting the game when in multiplayer mode.

\*

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <sys/ioctl.h>

#include <netinet/in.h>

#include <net/if.h>

#include <time.h>

#define SIZE 50 //defines the max message size

#define MAXLENGTH 12 //defines the max length of the game array

#define LOCALIP localIP //defines the local IP address, just a variable

#define PORTNUM 5050 //defines the port number, easier to find than searching if needed to be changed

const char valInput[] = {'A','S','D','W', 'Q','E'}; //constant character array of the valid data

char voidBuff[SIZE]; //just a character array used as a buffer

//function that will take in a message, output the message and error then exit the program.

//Parameters:

// const char \*msg -> message that is to be printed out

//Returns:

// Void

void error(const char \*msg){

perror(msg);

exit(0);

}

//debug function that is used to read in a message and variable to print out for debugging purposes

//Parameters:

// const char \*msg -> this is the message that will always be printed out

// const char \*variable -> this won't be printed if NULL, will be printed otherwise

//Returns:

// Void

void debug(const char \*msg, const char \*variable){

//if the variable is set to NULL then only print out the message

if(variable==NULL){

printf("DEBUG :: %s\n", msg);

}

//if the variable is not set to NULL then print out the message followed by the variable

else{

printf("DEBUG :: %s %s\n",msg,variable);

}

}

//function prototypes that are defined after main:

//checkAnswer is used to check if two strings match up or not

int CheckAnswer(char \*OppInput, char \*PlayerInput,int length);

//validateInput is used to verify that they have entered the correct letters for the game

int validateInput(char \*input, int length, int game);

//printInstructions is just used to easily print out the game's instructions

void printInstructions();

//trimWhiteSpace is a function that will eliminate any spaces in a string

char \*trimWhiteSpace(char \*str);

//simulateInput is used to create the "Computer's" input when playing one player

char \*simulateInput(int len, int game);

//computerPlay is used to guess the user's input

int computerPlay(char \*userInput, int length, int game);

//printSays is used to print out the array to the opponent, should flash the data then mask it

void printSays(char \*array, int length);

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

//defining variables:

time\_t t; //sets up a time\_t structure to use for random generator

srand((unsigned) time(&t)); //sets up random generator based upon current time of day

int sock, sendReceive; //integer values used to determine if some functions were successful

int flag = 1; //flag used in setting up sockopt

socklen\_t fromlen; //used to store the length of the socket

struct sockaddr\_in server; //structure used for the sending data to the server

struct sockaddr\_in from; //structure used for recieving the data from the server

char buffer[SIZE]; //just a buffer that will store any character data

struct ifreq ifr; //used to dynamically get the IP address

char localIP[10]; //stores the local IP address to this character array

int MasterFlag; //used as a boolean value to determine if I am the master or not

int gameOn = 0; //used as a boolean value to determine if the game needs to run or not

char \*buff = malloc(sizeof(char)\*SIZE); //buffer pointer that will store any character data

char \*debugger = malloc(sizeof(char)\*SIZE); //debugging pointer that is used for debugging strings

int option1= -1, game=1; //more variables used to run the game, option1 is the menu choice and game is the game boolean

char \*gameArray = malloc(sizeof(char)\*MAXLENGTH); //this pointer is used to locate the

//creates the socket for connectionless communication

sock = socket(AF\_INET, SOCK\_DGRAM, 0);

//checks if the socket was created

if(sock < 0){

error("ERROR :: Error with opening socket");

}

//gets the length of the sockaddr\_in structure

fromlen = sizeof(struct sockaddr\_in);

//sets all values to zeor. memset() could also be used

bzero(&server, sizeof(server));

//symbol constant for Internet domain

server.sin\_family = AF\_INET;

//this allows reading from any server that is broadcasting

server.sin\_addr.s\_addr = INADDR\_ANY;

//sets up the socket permissions to broadcast and checks for errors simultaneously

if(setsockopt(sock, SOL\_SOCKET, SO\_BROADCAST, &flag, sizeof(flag)) < 0){

error("DEBUG :: Error with setting up socket options\n");

}

//this chunk of code was used to get the IP address dynamically

ifr.ifr\_addr.sa\_family = AF\_INET;

snprintf(ifr.ifr\_name, IFNAMSIZ, "eth0");

ioctl(sock, SIOCGIFADDR, &ifr);

sprintf(localIP, "%s", inet\_ntoa(((struct sockaddr\_in \*)&ifr.ifr\_addr)->sin\_addr));

//sets the server to the correct port number and then sets the address to the nfs1 LAN allowing communication to all boards

server.sin\_port = htons(PORTNUM);

server.sin\_addr.s\_addr = inet\_addr("10.3.52.255");

//prints out the instructions using printInstructions function

printInstructions();

//runs the game until the user selects 4 or is kicked out of the loop

while(option1 != 4){

//shows user options for the game

printf("How would you like to play the game:\n");

printf("\t1)Play computer\n");

printf("\t2)Play Opponent\n");

printf("\t3)See Instructions again\n");

printf("\t4)Exit\n");

printf("Select: ");

//reads in the user option

scanf("%d",&option1);

//gets the null characters after the user option and discards them

fgets(buff, 2,stdin);

//switch statement that will control the user selected options.

switch(option1){

//case for when the user wants to play against a computer simulation

case 1:

///NEEDS WORK: the following is having issues with comparing the users data to the simulated

/// data if they are not the master. There are also some other misc issues that

/// can occur at different times. Didn't have enough time to fully debug this.

//lets the user know that the game is starting

printf("Initializing game against computer...\n");

printf("The computer will initially be the master. Good Luck!\n");

int length, num; //these define the length set by the user and the num is just a dummy variable

MasterFlag = 0; //sets master flag to low at start

int gameSize = 3; //starts the game size at 3, meaning there are only the first 3 valid characters allowed

//prompts the user for the length they want to initiate at, or starts at 3(DEFAULT)

//will continue to loop until they have entered valid information

while(1){

printf("What count (of letters) would you like to start with?(Default=3, MAX=7): ");

char \*input = malloc(sizeof(char)\*3);

fgets(input, SIZE, stdin);

input = trimWhiteSpace(input);

if(atoi(input) == 0){

length = 3;

break;

}

else if(atoi(input) >7){

printf("That's and invalid input. Try again.\n");

}

else{

length = atoi(input);

break;

}

}

//clarifies for the user what letters are allowed currently

printf("Valid letters are:");

for(num=0;num<gameSize;num++){

printf(" %c", valInput[num]);

}

//starts up the game, first checks if you are the master or not and then goes from there.

//loops until the gameSize is set to 0

while(gameSize!=0){

//if the users is the master then they are the first to implement data

if(MasterFlag){

//prompts the user for their pattern

printf("\nInput %d letters(i.e. A D S)\nmake sure to include spaces: ", length);

//the double length and addition 1 accounts for the spaces and null terminator

fgets(gameArray,(length+length+1),stdin);

//checks if the user entered valid letters only

int valid = validateInput(gameArray, strlen(gameArray), gameSize);

//checks if they have entered valid letters and the length is appropriate

if(strlen(gameArray) == length+(length-1) && valid == 1){

//this will simulate how a computer would guess at the pattern

int outcome = computerPlay(gameArray, length, gameSize);

//debugging statement, wont do anything when the inards of debug are commented out

sprintf(gameArray, "%d", outcome);

debug("Outcome is -> ",gameArray);

//if the outcome 1 then the computer matched your pattern

if(outcome == 1){

length++;

printf("Computer matched letters.\n");

//if the outcome is 0 then the computer guessed wrong

}else if(outcome == 0){

printf("You won!!!\nThe computer did not match your code!\n");

gameSize = 0;

//if none of the above then there was an error with the computer play function

}else{

error("ERROR :: Problem with outcome of computer play\n");

}

//checks if the user has made it to the max allowed pattern

//if they have, then restart length at 3 and add a new valid letter

if(length==13){

printf("Congratulations, you've made it to the next round!\n");

gameSize++;

length = 3;

printf("Valid letters are now:");

for(num=0;num<gameSize;num++){

printf(" %c", valInput[num]);

}

//checks if the length is divisible by two, if it is then switch masters

}else if((length%2) == 0){

MasterFlag = 0;

printf("Switching up Master");

}

//if the pattern was not appropriate length or invalid data, output message saying so

}else{

printf("Sorry, there was a problem reading your input.\n");

printf("Please make sure you're using only valid letters and using spaces to separate them.\n");

}

//if the user is not the master

}else{

printf("\n");

//gets simulated data from the function simulateInput

gameArray = simulateInput(length, gameSize);

//flashes pattern on the screen then masks is

printSays(gameArray, length);

//was used to check the pattern when debugging

debug("gameArray is: ", gameArray);

//loop that checks if the user has submitted a valid input or not

while(1){

printf("Repeat the code: ");

fgets(buff, length+length+4, stdin);

//debugging stuff that only prints out when needed

strcpy(debugger,buff);

debug("buff string = ",debugger);

sprintf(debugger, "%d", strlen(buff));

debug("length of buff = ",debugger);

sprintf(debugger, "%d", strlen(gameArray));

debug("length of gA = ",debugger);

//checks if the user input anything or if they didnt input a long enough pattern

if(buff == NULL || strlen(buff) != (length+length)){

printf("Invalid input, try again.\n");

}

else{

//checks the results against the computers pattern using CheckAnswer function

int results = CheckAnswer(gameArray, buff, length);

//checks if they were equal to eachother

if(results == 1){

printf("Correct!\n");

//checks if the length is divisible by 2 and switches master if it is

if((length%2) == 0){

MasterFlag = 1;

printf("Switching up Master");

}

length++;

printf("Increasing the length to %d",length);

break;

//if they arent equal to each other then let the user know

}else if(results == 0){

printf("Wrong!\n");

printf("Looks like the computer beat you. Better luck next time\n");

gameSize = 0;

break;

}

}

}

}

}

break;

//case for when the user wants to play against other users across a LAN

case 2:

///NEEDS WORK: This does not work properly. The functionality is written different than

/// single player game. Some of the logic is wrong since I couldn't properly work

/// with as a client. I thought I had all the proper code after looking at the examples

/// provided, but turns out I was missing a few things such as the correct way of

/// setting up the client and how it would receive messages. Currently it will

/// continue to run, but the functionality isn't working properly. With more time

/// I would have been able to implement this functionality as well.

printf("In order to play against opponent, first must have a valid opponent.");

printf(" Once an opponent has been found, you can begin the game.\n");

printf("Announce your challenge, Enter a name: ");

char \*playerName = malloc(sizeof(char)\*SIZE);

fgets(playerName, SIZE, stdin);

bzero(buffer, SIZE);

sprintf(buffer, "%s wants to play you in Adam Says", playerName);

buffer[SIZE - 1] = '\0';

debug("buffer about to send -> ", buffer);

sendReceive = sendto(sock, buffer, SIZE, 0,(const struct sockaddr\*)&server, fromlen);

if(sendReceive < 0){

error("Error with sendto1\n");

}else{

debug("Sent successfully: ", buffer);

}

printf("Initializing game against opponent...\n");

printf("Waiting for opponent to enter game...\n");

while(game){

bzero(buffer, SIZE);

sendReceive = recvfrom(sock, buffer, SIZE, 0, (struct sockaddr \*)&from, &fromlen);

if(sendReceive < 0 ){

error("ERROR :: sendReceive error\n");

}else{

debug("success - > ", buffer);

}

debug("received -> ", buffer);

if(buffer[0]=='$'){

gameOn = 1;

debug("Set gameOn to high", NULL);

}

if(gameOn == 1){

debug("Inside gameOn", NULL);

strtok(buffer, " ");

char \*OppIP = strtok(NULL, " ");

printf("Opponent found at IP: %s\n", OppIP);

printf("Prepare to start the game...\n");

//sets the buffer bits to zero everytime, essentially clearing it

bzero(buffer, SIZE);

//Since this player initiates the game, they are the master at beginning

MasterFlag = 1;

int turn = 3;

int verify = 2;

while(game){

//sets the buffer bits to zero everytime

bzero(buffer, SIZE);

if(MasterFlag){

if(turn != 3){ ///NEEDS WORK

printf("Enter %d letters now: \n@",turn);

fgets(gameArray,turn,stdin);

debug("You entered: ", gameArray);

bzero(buffer,SIZE);

strcat(buffer, "@");

sprintf(buffer, " %s", gameArray);

buffer[SIZE-1] = '\0';

//This sends buffer after the input has been made\*\*\*\*\*\*\*\*\*\*

sendReceive = sendto(sock, buffer, SIZE, 0, (struct sockaddr \*)&server, fromlen);

if(sendReceive < 0){

error("Error with sendto1\n");

}else{

printf("Sent successfully: %s\n", buffer);

}

}else if(turn%2==0){ ///NEEDS WORK\*\*\*\*\*\*\*\*\*\*\*\*

MasterFlag = 0;

printf("Switching up the master...\n");

bzero(buffer,SIZE);

sprintf(buffer, " %s", "!!SWITCH!!");

buffer[SIZE-1] = '\0';

//This sends buffer after the input has been made\*\*\*\*\*\*\*\*\*\*

sendReceive = sendto(sock, buffer, SIZE, 0, (struct sockaddr \*)&server, fromlen);

if(sendReceive < 0){

error("Error with sendto\n");

}else{

debug("Sent successfully: ", buffer);

}

}else{ ///CHECK\*\*\*\*\*\*\*

if(buffer[0] == '#'){

verify = CheckAnswer(buffer, gameArray, turn);

if(verify == 1){

printf("Your opponent responded correctly...Try again\n");

printf("Enter %d letters now: \n@ ",turn);

fgets(gameArray,turn,stdin);

debug("You entered: ", gameArray);

bzero(buffer,SIZE);

strcat(buffer, "@");

sprintf(buffer, " %s", gameArray);

buffer[SIZE-1] = '\0';

//This sends buffer after the input has been made\*\*\*\*\*\*\*\*\*\*

sendReceive = sendto(sock, buffer, SIZE, 0, (struct sockaddr \*)&server, fromlen);

if(sendReceive < 0){

printf("Error with sendto1\n");

return 1;

}else{

printf("Sent successfully: %s\n", buffer);

}

}else if(verify==0){

printf("CONGRATULATIONS!!!\nYOU WON!!!\n\n");

printf("Returning to Menu\n\n");

game = 0;

gameOn = 0;

break;

}

}else{

debug("Error with '#' statement. buffer = ", buffer);

}

}

}else{ ///CHECK\*\*\*\*\*\*\*

if(buffer[0] == '@'){

//flashInput(buffer, turn); ////\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

debug("opponent entered -> ", buffer);

printf("Repeat patter: \n# ");

fgets(gameArray, turn, stdin);

debug("you entered -> ", gameArray);

bzero(buffer,SIZE);

strcat(buffer, "#");

sprintf(buffer, " %s", gameArray);

buffer[SIZE-1] = '\0';

//This sends buffer after the input has been made\*\*\*\*\*\*\*\*\*\*

sendReceive = sendto(sock, buffer, SIZE, 0, (struct sockaddr \*)&server, fromlen);

if(sendReceive < 0){

error("Error with sendto1\n");

}else{

debug("Sent successfully: ", buffer);

}

}else if(strncmp(buffer,"!!SWITCH!!", 10)==0){

MasterFlag = 1;

}

}

turn++;

}

}

}

break;

//case for if the user wants to read the instructions again

case 3:

///DONE

printInstructions();

break;

//case for when the user is finished playing after getting to the menu

case 4:

///DONE

printf("Exiting the game....");

game = 0;

gameOn = 0;

printf("\nThanks for playing!\n");

return 3;

//default case if none of the above has been entered.

default:

///DONE

printf("Invalid input, please enter a value between 1 and 3.\n");

break;

}

//prompts the user asking if they want to coninue playing

printf("Continue playing(Y or N)? ");

fgets(buff, 2, stdin);

//checks if they input n or N, if they did then exit the game

if(buff[0] == 'N' || buff[0] == 'n'){

printf("Exiting the game....\n");

printf("Thanks for playing!\n");

return 0;

}

}

//frees up the memory that was allocated for the pointers

free(gameArray);

free(buff);

free(debugger);

return 0;

}

//function that just prints out the instructions when prompted to

//Parameters:

// None

//Returns:

// Void

void printInstructions(){

printf("\n\n\n\*\*\*\*\*\*\*\*\*\*\*Welcome to Adam Says\*\*\*\*\*\*\*\*\*\*\*\n\n\n");

printf("This is a simple memory game that will test your memory skills and push them to \ntheir limits.\n\n");

printf("Once a game has been started you will use the following keys in order to play \n(CASE SENSITIVE): A, W, S, D. ");

printf("At the start of the game, you will only be \nallowed to enter 3 of the allowed letters and then ");

printf("if your opponent gets them \nright, you can enter 4 allowed letters. If your opponent gets those right,\n");

printf("then it will switch to the opponent inputting allowed letters as well as \nincrementing to 5 letters\n");

printf("\nThe pattern will continue this way switching back and forth after every even \nround (i.e. after 4, 6, 8, 10, etc.)");

printf(" until you or your opponent enters a \ncommand wrong.\n\n");

printf("If a wrong pattern is not input after using 12 letters, the game will reset \nthe pattern counter back to 3 ");

printf("and then add another valid letter, which \nwill only be seen once you've reached this point\n\nGood Luck!!\n");

}

//This function is meant to print out the pattern to the opponent by flashing each letter,

//waiting a while and then backspacing and prints out an X to mask the pattern as it goes.

//Parameters:

// char \*array -> the array that is to be printed out

// int length -> the length of the array that needs printing

//Returns:

// Void

void printSays(char \*array, int length){

//NEEDS WORK: I've tried to do this a few different ways and even looked up several ways to do this,

// many have said to use the printf("\033[2D") method which would back track the cursor

// two positions then write over that. The second way is to just backspace over the letter

// and then rewrite the mask to terminal. When I try doing either method, it doesn't print out

// the pattern and will instead rapidly switch to the X. Couldn't get this fully working yet

int x,y;

for(x=0;x<length; x++){

printf("%c ",array[x]);

for(y=0;y<1000000;y++){ }

//printf("\033[%dD",1);

printf("\b\b");

//for(y=0;y<100000;y++){ }

printf("X ");

}

}

//This function is used to clean up any string from having spaces in it

//Parameters:

// char \*str -> the string that needs spaces cleared

//Returns:

// char\* -> returns the newly created spaceles string

char \*trimWhiteSpace(char \*str){

char \*end;

// Trim leading space

while(isspace(\*str)) str++;

if(\*str == 0) // All spaces?

return str;

// Trim trailing space

end = str + strlen(str) - 1;

while(end > str && isspace(\*end)) end--;

// Write new null terminator

\*(end+1) = 0;

return str;

}

//CheckAnswer tokenizes the opponents input and players input, then compares the two strings,

//if at any point the two don't match up then return 0. If they do match, then return 1.

//Parameters:

// char \*OppInput -> the opponents pattern

// char \*PlayerInput -> the players input pattern

// int length -> the length of both patterns

//Returns:

// integer -> this integer returns 1 if correct answer, 0 if wrong answer

int CheckAnswer(char \*OppInput, char \*PlayerInput,int length){

int x;

char \*O = malloc(sizeof(char)\*SIZE);

char \*P = malloc(sizeof(char)\*SIZE);

O = trimWhiteSpace(OppInput);

P = trimWhiteSpace(PlayerInput);

debug("DEBUG :: O = ",O);

debug("DEBUG :: P = ",P);

if(strcmp(O, P) == 0)

return 1;

else

return 0;

}

//validateInput is used to verify that the user has input valid letters into the pattern.

//returns 0 if an invalid letter was used, returns 1 if all were valid.

//Parameters:

// char \*input -> the pattern that is to be checked

// int length -> the length of the array that needs checking

// int game -> defines how many valid letters are currently allowed in the game, reference valInput array

int validateInput(char \*input, int length, int game){

int x,y;

for(x=0;x<length;x++){

for(y=0;y<game;y++){

if(input[x] != valInput[y])

return 0;

}

}

return 1;

}

//simulateInput is used to create a randomly generated pattern based on random probabilities

//that are created within the function. Once the pattern is created, it is then returned.

//Parameters:

// int len -> the length of the pattern that is needed

// int gameSize -> the valid letters currently allowed, referencing the valInput array

char \*simulateInput(int len, int gameSize){

int prob[gameSize], r,x,y,z;

char result[len];

char \*sendInput = malloc(sizeof(char)\*13);

char \*debugger = malloc(sizeof(char)\*20);

//loop that will set up the random probabilities for each in use letter

for(y=0;y<gameSize;y++){

r = (rand()%gameSize)+1;

sprintf(debugger, "%c is %d",valInput[y],r);

debug("\n",NULL);

debug("Probability of ",debugger);

prob[y] = r;

}

z=0;

y = 0;

//loop that will go through and check the highest probability in the prob array,

//once the highest is found, it will use that letter then subtract from that

//probability until another letter has a higher probability.

while(z < len){

for(x=0;x<gameSize;x++){

//debugger stuff

sprintf(debugger,"%d",prob[x]);

debug("x = ",debugger);

sprintf(debugger,"%d",prob[y]);

debug("yB = ",debugger);

//checks for highest probability

if((prob[x] > prob[y])){

y = x;

}

}

//sets result then subtracts from that probabality

result[z] = valInput[y];

prob[y]--;

//debugger stuff

sprintf(debugger,"%d",prob[y]);

debug("yA = ",debugger);

sprintf(debugger, "%c ",result[z]);

debug("result[z] = ", debugger);

//concatenates the result to the sendInput

strcat(sendInput, " ");

strcat(sendInput, debugger);

//increments the result aray

z++;

}

//returns the final product

return sendInput;

}

//computerPlay simulates how the computer would go about responding to the user's pattern.

//the probability of the computer matching the pattern is assumed to be 70% chance to allow

//a reasonably fair game so it's not a never ending game.

//Parameters:

// char \*userInput -> the users pattern that was created

// int length -> the length of the pattern

// int game -> refers to the valid letters of the current game, references valInput array

int computerPlay(char \*userInput, int length, int game){

//int comp[length],y;

int x,r;

int prob[] = {0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0}; //70% probability (10 chances of right out of 13)

int count1 =0, count0=0; //count1 stores number of correct guesses, and count 0 stores number of wrong

char \*debugger = malloc(sizeof(char)\*11);

for(x=0;x<length;x++){

//gets a random integer to reference prob array

r = rand()%12;

//debugging stuff

sprintf(debugger, "%d",r);

debug("random = ", debugger);

sprintf(debugger, "%d", prob[r]);

debug("prob = ", debugger);

//if it hits a 1 in the prob array, add to count1

if(prob[r] == 1){

debug("adding to count1", NULL);

count1++;

//if it hits a 0 in the prob array, add to count0

}else{

debug("adding to count0", NULL);

count0++;

}

}

//debugging stuff

sprintf(debugger, "1-%d vs 0-%d", count1, count0);

debug("compare ::", debugger);

//checks if the computer had more correct than wrong,

//returns 1 if they had more correct.

//returns 0 if they har more wrong.

if(count1 > count0)

return 1;

else if(count1 < count0)

return 0;

//returns -1 if there was an error

return -1;

/\*

\* This was the first algorithm I came up with, but it was not as effective since it

\* only used random numbers and not a probability aspect. Since it's a computer, the

\* computer should be able to recognize

for(x=0;x<length;x++){

for(y=0;y<game;y++){

r = (rand()%game);

sprintf(debugger, "%d",r);

debug("random = ", debugger);

sprintf(debugger,"%c", valInput[r]);

debug("valInput = ", debugger);

if(userInput[x] == valInput[r]){

debug("inside comparison!", NULL);

comp[x] = 1;

break;

}

}

if(comp[x] == 1){

debug("adding to count1", NULL);

count1++;

}else{

debug("adding to count0", NULL);

count0++;

}

}

sprintf(debugger, "1-%d vs 0-%d", count1, count0);

debug("compare ::", debugger);

if(count1 > count0)

return 1;

else if(count1 < count0)

return 0;

else{

int tie = rand()%2;

sprintf(debugger, "%d", tie);

debug("TIEBREAKER->", debugger);

return tie;

}

\*/

}

# Bibliography

[1] National Museum of American history, ‘Simon Electronic Game, 1978’, 2015.[Online]. Available: http://americanhistory.si.edu/collections/search/object/nmah\_1302005. [Accessed: 12-May-2015].