Mizzou Arcade Center

Tyler Mironuck, ECE

Prepared for: ECE 4220 – Real Time Embedded Systems, University of Missouri

May 9, 2014

***Abstract -*** The Mizzou Arcade Center is a server-based gaming system that turns the Linux lab into an Arcade style game room. Each game is essentially a client process that is constantly sending information like current game status to a main server, which evaluates this information and then sends back instructions to the game. The server also stores the status of each game and organizes it into a “Status Report” that shows up-to-date statistical information of every game that is being used.

Objective and Project Description

The purpose of this project is to design a complete network communication system that uses multiple embedded systems that all run their own client process and are all monitored and controlled by a single server. Each client process represents a unique Arcade game based on the LED lights cycling through different colors at specific frequencies. The objective of the game is to hit the button during the exact time interval that the yellow LED is turned on. The difficulty of the game is associated with what frequency the lights are changing at. The faster the lights are changing, the faster your reflexes have to be in order to successfully hit the light at the precise time interval that the yellow light is on. The game has five levels and as you complete each level, you move on to a higher, more difficult level where the lights change more rapidly. While each game is running, the server is collecting current statistics and the status of each game including which game is currently being used, the current score of the player, how many misses the player has accrued, and the level that they are currently on. It is then constantly storing them in a “Status Report” that can be displayed at any time at the server station. The server also controls the progress of each game by determining which level of difficulty each game should be at. It does this by continuously receiving the score and then deciding which level the user should be at by sending an instruction back to the client game to change to the specified level.

Implementation

My project can essentially be broken down to three main parts: a kernel module that links the hardware and software for the game, the client program that represents the game logic, and a server program. For the kernel module I created code to set up a real time task, a hardware interrupt and a software interrupt. The purpose of the real time task is to make the LED lights cycle through periods of being on to create a sequence characterized by green being on, then yellow, then red and then repeating this sequence continuously. Since the game centers on the ability of the player to hit the button at the exact time interval that the yellow light is on, the timing of each light is essential to the validity of the game. Therefore this process needed to be implemented using a hard real time task. Inside the real time task I request a timestamp every time the yellow light turns on and store this value inside a global variable and then I also do the same thing for when the light turns off. Therefore every time the lights are on a new cycle the global variables for when the light turned on and off will be over written by the most recent timestamps of the events. The hardware interrupt is used to trigger a hardware handler to execute when the pin 0 button on Port B is pushed. The first thing the hardware handler does is it requests a timestamp of the event of the player pushing the button. Next it sends this timestamp to the client program via a FIFO. It then sends the variable holding the most recent time stamp of when the yellow light began to the client program. However it does not send the most recent time stamp of when the yellow light turned off for an important reason. This is discussed further in the discussion section. Instead, the real time task that controls the period of the lights writes the timestamp of when the yellow light ends to a FIFO. Since this takes place in the real time task, this FIFO gets written to every time the lights cycle through. Essentially the main goal of the hardware interrupt service routine and real time task is to collect the timestamps of three events – the yellow light turning on, the yellow light turning off, and the event of the button being pressed – and then send these timestamps to the client program where they will be evaluated. There are no calculations done in the kernel module to determine if the time stamp of the button press was in fact within the time interval that the yellow light was on. This was because of limitations with data types inside the kernel module. This is also a topic that will be expanded on in the discussion section. The last part of the kernel module is the software handler. The purpose of the software handler is to change the frequency that the lights are cycling at when the player changes levels. Each level is associated with it a certain frequency that the lights change at. When the client program is about to change levels, it triggers the software interrupt and then the handler for the software interrupt retrieves which level the client program is switching to via a FIFO and finally changes the period of the real time task to match that of the level.

The client program is the second part of the project and contains all of the logic for the game. This program is basically made up of the main section and then two pthreads. In the main section the first thing I do is retrieve the IP address of the computer I am running on. Since many games can be running at once, each game uses its own unique IP address as a way of identifying itself when sending information to the server. Then I create a socket for UDP connectionless communication and set socket permissions to allow broadcasting. I then go into an infinite while loop that begins the game by the player if they would like to play. If it gets a response from the standard input, then the program sends a message to the server that this game is in use. It then creates the two pthreads. The first is a thread is for receiving the timestamps from the kernel module. In this thread is the actual code to determine if the timestamp of the button was between the timestamp of the light turning on and the timestamp of the light turning off. In the beginning of the thread, it just reads from the FIFO that has the timestamp of the button being pressed. Since the read() function is a waiting function, the thread will just wait until the kernel module writes to the FIFO, meaning the button was pushed, and then it will receive the other two timestamps and calculate if the player hit the button at the right interval. If it determines the button was hit during the interval that the yellow light was on, then it broadcasts a “user scored” message to the server program so that it can decide which level the client program should change to. If the thread determines the player missed the interval, then it will also broadcast this record of a miss to the server program. The second thread is for receiving instructions from the server through a socket about which level to switch to. It runs in a continuous loop that receives a message, decides whether the level is a 1,2,3,4 or 5, puts the level number into a FIFO that is connected to the kernel module, and then finally triggers a software interrupt. Triggering the software interrupt causes the kernel module to immediately execute the software handler responsible for switching the period of lights.

The last part of my project is the server program. This program consists of a main and a separate thread. The main is just a continuous while loop that reads in any messages that have been broadcasted. It then goes through a series of checks to determine if the received message was a score, a game status (whether game is in used or not), or a miss. The pthread is created to monitor whether the operator of the server program has requested a Status Report of all the boards. It just waits for the operator to enter “status” and it will print out the status of each game that is currently in use. To achieve this I implemented an array of structures with each structure containing the game status, game ID, the score, misses, and the level. When the operator requests a status, the thread just checks the status variable of each structure in the array. If the status variable is a “1”, this means the game is being used and that the structure holds relevant data and statistics about the game.

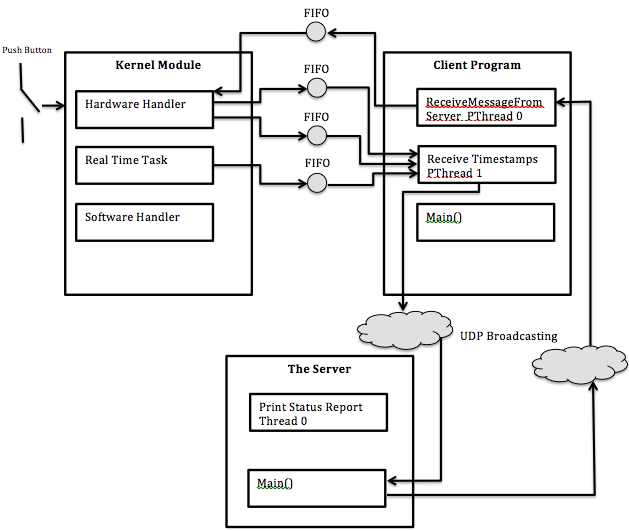


Figure 1 Software Flowchart

Figure 1 above shows a complete software flowchart of my project. You can see there are three FIFO’s for sending each timestamp to the client program. Also the client program can be controlled by the server program since it sends it the status of gameplay through UDP broadcasting and then receives instruction messages back and processes them in the pthread 0.

Results

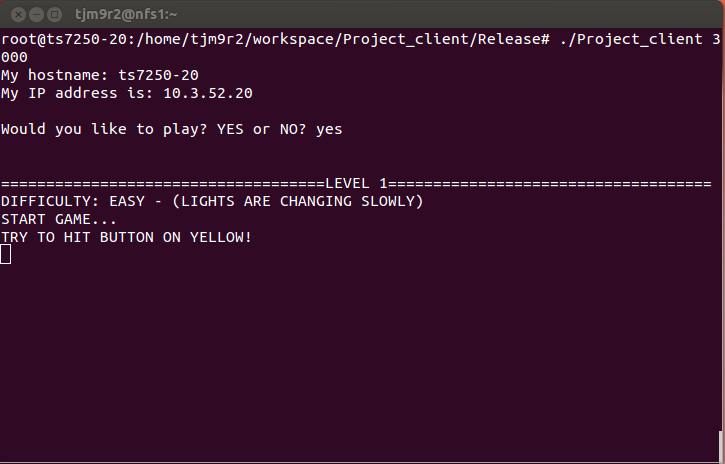


Figure 2 Player Just Starting the Game

Figure 2 shows the player choosing to play the game and starting at level 1. You can see that in level 1 the difficulty is set as “easy” which means the lights are changing relatively slow.

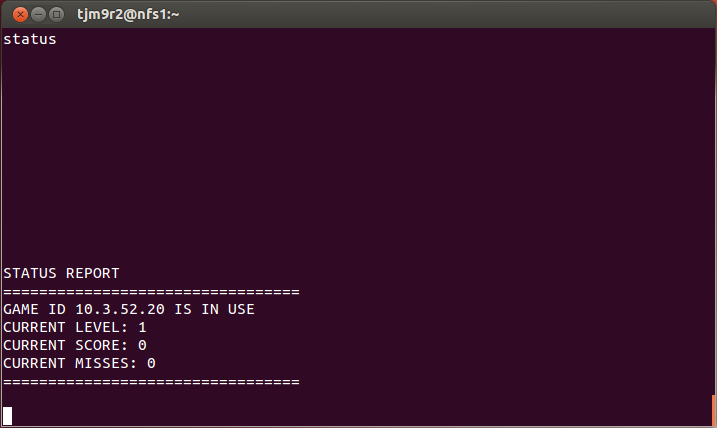


Figure 3 Status Report Showing Game 20 In Use

Figure 3 shows the results of when the operator of the server requests a Status Report. A status report sends the status of every game that is in use. Since game 20 is the only game in use, there is only one current Status Report. Notice also that the level matches the level in the game (client program) and that the score and misses are both zero since the player has not attempted hitting the button yet.

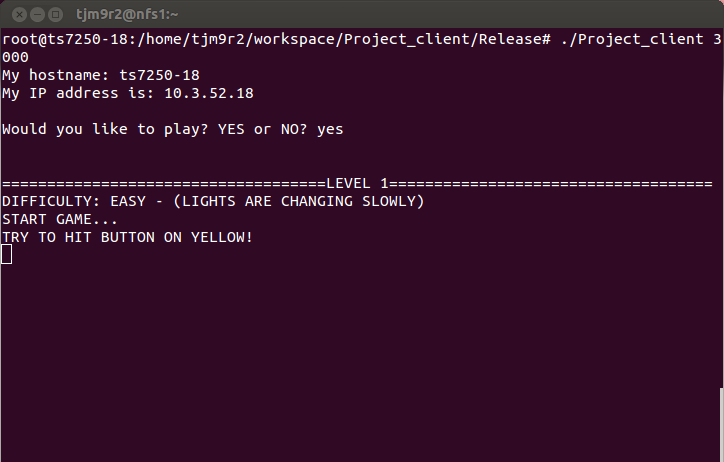


Figure 4 User On Game 18 Begins Play

In Figure 4 the player on game 18 begins play. At this time there are two games in play and if the operator of the server program requests a Status Report it should show both games in use.

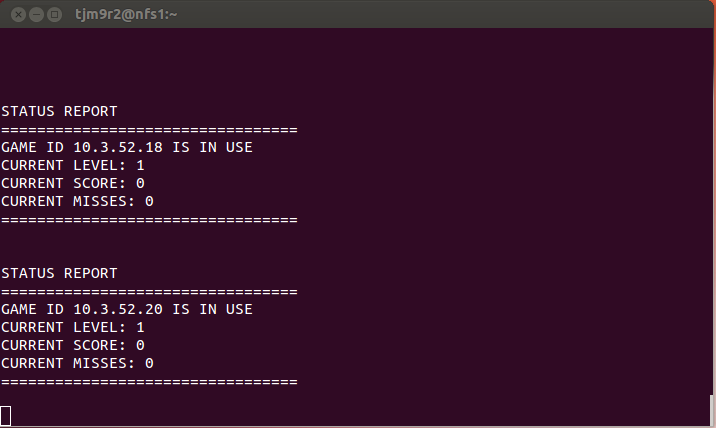


Figure 5 Showing Status of all Games in Use

Figure 5 shows all the games that are currently in use. Since the player on game 18 just started, the Status Report has both games showing.

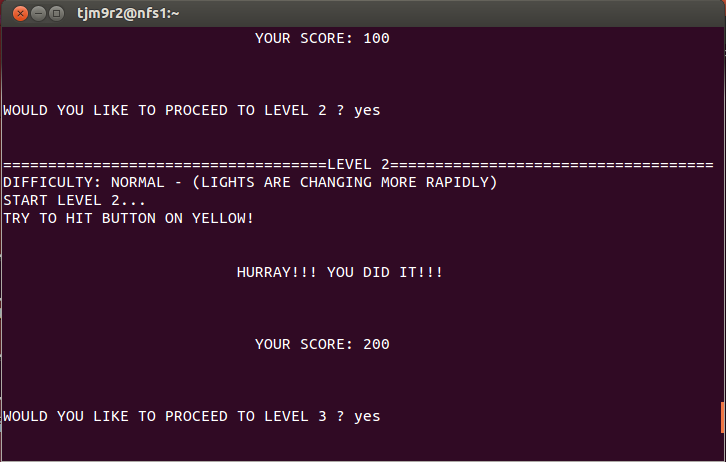


Figure 6 Player on Game 20 Just Completed Level 2 is Now on Level 3

Figure 6 shows that the player just beat level 2 where the skill level was “Normal” and the lights were changing faster. His score is now 200 and he is no level 3. A status report should reflect these current statistics.

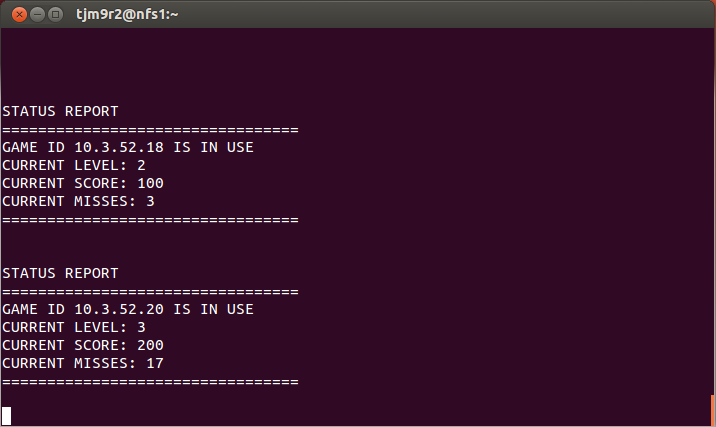


Figure 7 Status Report of all Boards In Use

As you can see from Figure 7 which shows a Status Report, the statistics of the player on game 20 coincides with the what is shown in the Status Report for game 20.

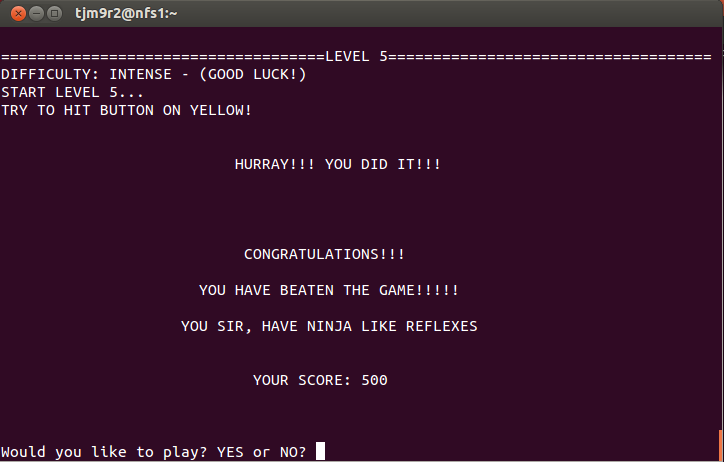


Figure 8 Player on board 20 Beat the Game

Figure 8 shows that the player on game 20 has completed level 5 and therefore has beaten the game. Now that the game is not in use it should not show up in the current Status Report.

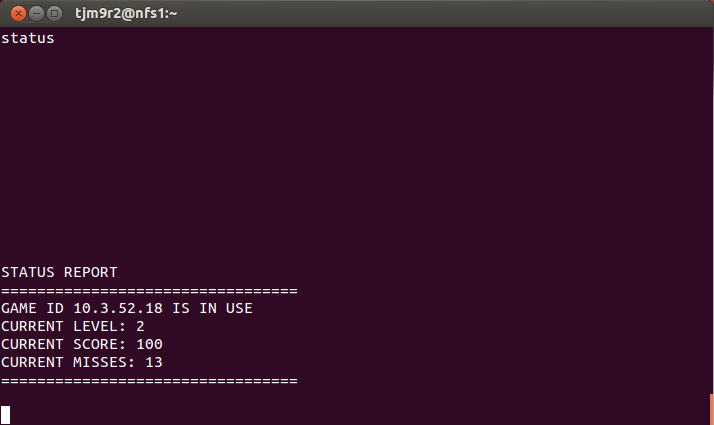


Figure 9 Status Report of all Games in Use

Figure 9 proves that the server programs successfully monitors relevant game data and since game 20 just beat the game and is no longer playing only game 18 is still running.

Experiments and Discussion

While creating this project I ran into many difficulties. I started designing the project from the kernel module and right off the bat ran into trouble. After reading the timestamps of the button being pushed and then the timestamp of when the light turned on and then off, I would try to calculate whether the button pushed was done in the interval that the light was on. The board continued to crash though every time. It took me quite a bit of trial and error to realize that the issue came from the type of variables I was saving the timestamps in. I was saving all these timestamps in variables of type “Double”. This is because these timestamps are of type struct timeval and are very large and therefore need data types bigger than integers. The problem is that the kernel module (TS-7250) cannot handle data types this large. I realized that I would have to send the raw timestamps to the client program and then do all the calculations in user space. This lead to even more complexity.

Previously in the implementation section I discussed the importance of not sending the timestamp of the most recent instance of the light turning off in the hardware handler. Consider this scenario. What if the yellow light has just turned on and so the global variable is holding this timestamp and then the player pushes the button and triggers the handler where it saves the timestamp of the button being pushed all before the yellow light has turned off? If this happens the global variable holding the value of when the light turned off is actually holding the timestamp of this event for the previous cycle since the event of the light turning off hasn’t happened yet. This would be the wrong timestamp and you would want to wait to send the time of when the light turns off in the current cycle. Overcoming this dilemma is not a trivial matter and requires some intricate programming. I first considered using a sleep function inside the handler to wait for when the yellow light turns off in the current cycle. But then that wouldn’t work because the period of the lights continues to change as the levels increase so you wouldn’t know how long to sleep for. I then experimented with putting a while loop in the handler that just repeats until the condition that the timestamp when the light is turned off is larger than the timestamp of when it is turned on. This would ensure that you are recording the correct timestamp. This resulted in the board crashing though every time I tested it. I think this is because it controlled the CPU and wouldn’t give it up for the real time task. Whatever the reason was, I found that while loops implemented in the handler section would always crash the board. The solution I finally arrived at came from the convenient waiting feature of the “read()” function for FIFOs. This feature allowed me to handle sending the timestamp of when the yellow light ends in the real time task, which would therefore put the timetamp in the FIFO each time the lights go through another cycle. Since the reading function is a waiting function though, these timestamps do not have to be read in the client program until the button is pressed. In the client program reading from the fifo that contains the timestamp of when the button was pressed will actually cause the program to wait until there is something in the FIFO. In this case there won’t be anything in the FIFO until the handler is triggered and then writes to the fifo. For this reason it doesn’t matter that that the real time task in the kernel module writes the end of light timestamp to the fifo over and over again.

Another issue I ran into was while writing my client program. I needed a way to terminate both pthreads when the player decided to stop playing. This was difficult since ReceiveLightTimes pthread had FIFO read functions that would prevent the while loop from repeating and checking the condition statement. I needed a way to pause this thread until the ReceiveMessageFromServer thread had determined whether the player wanted to proceed to the next level or not. I ended up solving this issue with a “quitFlag” and a “pauseFlag”. I wasn’t able to resolve the issue of the player being able to quit the game until I changed the quitFlag and pausFlage to “volatile int” data type. This prevented the compiler from changing the order in which these variables were written to which was essential being able to exiting the pthreads.

Code Section

Project\_KernelMod.c

#ifndef MODULE

#define MODULE

#endif

#ifndef \_\_KERNEL\_\_

#define \_\_KERNEL\_\_

#endif

#include <linux/module.h>

#include <linux/kernel.h>

#include <asm/io.h>

#include <rtai.h>

#include <rtai\_sched.h> //scheduler

#include <rtai\_fifos.h> //for Fifos

#include <linux/time.h>

#include <linux/time.h>

MODULE\_LICENSE("GPL");

unsigned long \*startPtr, \*startPtr2, \*PBDR, \*PBDDR, \*GPIOBIntEn, \*RawIntStsB;

unsigned long \*GPIOBEOI, \*GPIOBIntType1, \*GPIOBIntType2, \*GPIOBDB;

int portB\_Status;

//VIC2 handles Interrupt lines 32 through 63

unsigned long \*VIC2SoftIntClear, \*VIC2IntEnable;

//declare variables of struct timeval

struct timeval Time;

struct timeval TimeOfPress;

struct timeval YellowLightBegin;

struct timeval YellowLightEnd;

static RT\_TASK mytask;

RTIME period;

static void my\_handler(unsigned irq\_num, void \*cookie){

rt\_disable\_irq(59); //Disable request line so no new interrupts can trigger

portB\_Status = \*RawIntStsB; //Assign status of portB interrupt to portB\_Status

if((portB\_Status & 0x01) != 0x00){ //If bit 0 button of port B is pressed

do\_gettimeofday(&TimeOfPress); //collect timestamp of when button was pressed

rtf\_put(0, &TimeOfPress, sizeof(TimeOfPress)); //Send timeStamp of button press to program

}

rtf\_put(1, &YellowLightBegin, sizeof(YellowLightBegin)); //Send timeStamp of yellow light begin

//rt\_sleep(nano2count(100000));

\*GPIOBEOI |= 0x0000001F; //clear interrupt (EOI stands for End Of Interrupt)

rt\_enable\_irq(59); //Re-enable interrupt handling

}

static void my\_software\_handler(unsigned irq\_num, void \*cookie){

int level = 0;

rt\_disable\_irq(63); //Disable interrupt handling so no new interrupts can trigger

rtf\_get(3, &level, sizeof(level)); //Retrieve note from UserSpace program via FIFO

if(level == 1){

rt\_task\_make\_periodic(&mytask, rt\_get\_time()+10\*period, 4000 \*period); //Makes lowest frequency

}

else if(level == 2){

rt\_task\_make\_periodic(&mytask, rt\_get\_time()+10\*period, 2000 \*period); //Increase frequency

}

else if(level == 3){

rt\_task\_make\_periodic(&mytask, rt\_get\_time()+10\*period, 1300 \*period); //Increase frequency

}

else if(level == 4){

rt\_task\_make\_periodic(&mytask, rt\_get\_time()+10\*period, 1100 \*period); //Increase frequency

}

else if(level == 5){

rt\_task\_make\_periodic(&mytask, rt\_get\_time()+10\*period, 900 \*period); //Increase Frequency

}

else if(level == 5){

rt\_task\_make\_periodic(&mytask, rt\_get\_time()+10\*period, 700 \*period); //Makes highest frequency

}

\*VIC2SoftIntClear |= 0x80000000; //Clear software interrupt 63 (Clears bit 31 but it is in VIC2 so it actually clears line 63)

rt\_enable\_irq(63); //Re-enable interrupt handling

}

static void rt\_process(int t) //argument not used

{

while(1){

\*PBDR |= 0x80; //Turn on green light

rt\_task\_wait\_period(); //wait for a period

\*PBDR &= 0xFFFFFF7F; //Turn off green light

do\_gettimeofday(&YellowLightBegin); //Collect time stamp of when yellow light begins

//rtf\_put(1, &YellowLightBegin, sizeof(YellowLightBegin)); //Send timeStamp of yellow light begin

\*PBDR |= 0x40; //Turn on yellow light

rt\_task\_wait\_period(); //wait for a period

\*PBDR &= 0xFFFFFFBF; //Turn off yellow light

do\_gettimeofday(&YellowLightEnd); //Collect time stamp of when yellow light ends

rtf\_put(2, &YellowLightEnd, sizeof(YellowLightEnd)); //Send timeStamp of yellow light end

\*PBDR |= 0x20; //Turn on red light

rt\_task\_wait\_period(); //wait for a period

\*PBDR &= 0xFFFFFFDF; //Turn off red light

}

}

int init\_module(void)

{

startPtr = (unsigned long \*) \_\_ioremap(0x80840000,4096,0);

//set startPtr2 to VIC2 Base Address: 0x800C0000 (beginning of second mapped area)

startPtr2 = (unsigned long \*) \_\_ioremap(0x800C0000,4096,0);

//offsets to port B at address location 0x80840004

PBDR = (unsigned long \*) ((char \*) startPtr + 0x04);

//offsets to DDR for port B at address location 0x80840014

PBDDR = (unsigned long \*) ((char \*) startPtr + 0x14);

GPIOBIntEn = (unsigned long \*) ((char \*) startPtr + 0xB8); //offset to GPIOBIntEn

GPIOBIntType1 = (unsigned long \*) ((char \*) startPtr + 0xAC); //offset to GPIOBIntType1

GPIOBIntType2 = (unsigned long \*) ((char \*) startPtr + 0xB0); //offset to GPIOBIntType2

GPIOBEOI = (unsigned long \*) ((char \*) startPtr + 0xB4); //offset to GPIOBEOI

GPIOBDB = (unsigned long \*) ((char \*) startPtr + 0xC4); //offset to GPIOBDB

RawIntStsB = (unsigned long \*) ((char \*) startPtr + 0xC0); //offset to RawIntStsB

//Offset from startPtr2

VIC2SoftIntClear = (unsigned long \*) ((char \*) startPtr2 + 0x1C);

VIC2IntEnable = (unsigned long \*) ((char \*) startPtr2 + 0010);

\*PBDDR &= 0xFFFFFFE0; //Set data driection bits to low for bits 0 through 4 of port B

\*GPIOBIntType1 |= 0x1F; //Set bits 0 through 4 of Port B to EDGE sensitive

\*GPIOBIntType2 &= 0xFFFFFFE0; //Set bits 0 through 4 of Port B to FALLING edge sensitive

\*GPIOBDB |= 0x1F; //Set bits 0 through 4 of Port B to debounce

//Set data direction bits 5, 6 and 7 to high for port B to output

\*PBDDR |= 0xE0;

//Turns all lights off

\*PBDR &= 0xFFFFFF1F;

//set to periodic mode

rt\_set\_periodic\_mode();

period = start\_rt\_timer(nano2count(100000));

//initialize real time task, and make it periodic

rt\_task\_init(&mytask, rt\_process, 0, 256, 0, 0, 0);

rt\_task\_make\_periodic(&mytask, rt\_get\_time()+10\*period, 4000\*period);

rtf\_create(0,1\*sizeof(Time)); //create fifo 0

rtf\_create(1,1\*sizeof(Time)); //create fifo 1

rtf\_create(2,1\*sizeof(Time)); //create fifo 2

rtf\_create(3,1\*sizeof(int)); //create fifo 3

rt\_request\_irq(59, my\_handler, 0, 1); //Attach my\_handler to request line 59

\*GPIOBEOI |= 0x0000001F; //Clear interrupt before enabling interrupt handling

\*GPIOBIntEn |= 0x0000001F; //Configure bits 0 through 4 of Port B as interrupts

rt\_enable\_irq(59); //Enable interrupt handling on interrupt line 59

rt\_request\_irq(63, my\_software\_handler, 0, 1); //Attach my\_software\_handler to request line 63

\*VIC2SoftIntClear |= 0x80000000; //Clear software interrupt 63

\*VIC2IntEnable |= 0x80000000; //Enable line 63 as interrupt

rt\_enable\_irq(63); //Enable interrupt handling on interrupt line 63

return 0;

}

void cleanup\_module(void)

{

unsigned long \*startPtr, \*PBDR;

startPtr = (unsigned long \*) \_\_ioremap(0x80840000,4096,0);

//offsets to port B at address location 0x80840004

PBDR = (unsigned long \*) ((char \*) startPtr + 0x04);

rtf\_destroy(0); //destroy fifo

rtf\_destroy(1); //destroy fifo

rtf\_destroy(2); //destroy fifo

rtf\_destroy(3); //destroy fifo

\*PBDR &= 0xFFFFFF1F; //Turn off all lights

rt\_disable\_irq(59); //disable interrupt request line 59

rt\_release\_irq(59); //release interrupt request line 59

rt\_disable\_irq(63); //disable interrupt request line 63

rt\_release\_irq(63); //release interrupt request line 63

rt\_task\_delete(&mytask); //delete real time task

stop\_rt\_timer(); //stop timer

}

Project\_client.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <pthread.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <semaphore.h>

#define MSG\_SIZE 40 // message size

struct timeval TimeOfPress;

struct timeval YellowLightBegin;

struct timeval YellowLightEnd;

double time\_of\_press[1000000];

double light\_begin;

double light\_end;

struct sockaddr\_in anybody, from;

int sock, n;

unsigned int length;

socklen\_t fromlen;

int num\_of\_wins = 0;

int misses = 0;

char num\_wins[20], num\_miss[20];

char buffer[MSG\_SIZE]; // to store received messages or messages to be sent.

char userInput[MSG\_SIZE]; //to store user input from keyboard

int i;

volatile int quitFlag = 0;

volatile int pauseFlag = 0;

sem\_t sem1; //declar semaphore

void \*RecieveMessageFromServer(void \*arguments){

int level = 1;

int fd, fd\_fifo\_out;

unsigned long \*startPtr, \*VIC2SoftInt;

//open /dev/mem file to map the areas of physical address space into process user address space

fd = open("/dev/mem", O\_RDWR | O\_SYNC);

//set startPtr to VIC2 Base Address: 0x800C0000

startPtr = mmap(NULL, getpagesize(), PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0x800C0000);

VIC2SoftInt = (unsigned long \*) ((char \*) startPtr + 0x18); //offset to VIC2SoftInt

fd\_fifo\_out = open("/dev/rtf/3", O\_RDWR); //To access fifo 3

write(fd\_fifo\_out, &level, sizeof(level)); //write to fifo 3 that was created in the kernel module

\*VIC2SoftInt |= 0x80000000; //Trigger software interrupt in user space

while(quitFlag == 0){

char difficulty[30], comment[100];

// receive message

n = recvfrom(sock, buffer, MSG\_SIZE, 0, (struct sockaddr \*)&from, &length);

if (n < 0)

error("recvfrom");

if(strncmp(buffer,"LEVEL2",6) == 0){ //if message is LEVEL2, change frequency to level 2

level = 2;

strcpy(difficulty, "NORMAL");

strcpy(comment,"LIGHTS ARE CHANGING MORE RAPIDLY");

printf(" YOUR SCORE: %d\n\n\n\n", num\_of\_wins\*100);

}

if(strncmp(buffer,"LEVEL3",6) == 0){ //if message is LEVEL3, change frequency to level 3

level = 3;

strcpy(difficulty, "HARD");

strcpy(comment,"LIGHTS ARE CHANGING VERY FAST");

printf(" YOUR SCORE: %d\n\n\n\n", num\_of\_wins\*100);

}

if(strncmp(buffer,"LEVEL4",6) == 0){ //if message is LEVEL4, change frequency to level 4

level = 4;

strcpy(difficulty, "VERY HARD");

strcpy(comment,"LIGHTS ARE CHANGING EXTREMELY FAST");

printf(" YOUR SCORE: %d\n\n\n\n", num\_of\_wins\*100);

}

if(strncmp(buffer,"LEVEL5",6) == 0){ //if message is LEVEL5, change frequency to level 5

level = 5;

strcpy(difficulty, "INTENSE");

strcpy(comment,"GOOD LUCK!");

printf(" YOUR SCORE: %d\n\n\n\n", num\_of\_wins\*100);

}

if(strncmp(buffer,"END",2) == 0){ //if message is LEVEL6, USER HAS BEAT THE GAME

printf("\n CONGRATULATIONS!!!\n\n YOU HAVE BEATEN THE GAME!!!!!\n\n"

" YOU SIR, HAVE NINJA LIKE REFLEXES\n\n\n");

printf(" YOUR SCORE: %d\n\n\n\n", num\_of\_wins\*100);

quitFlag = 1;

pauseFlag = 0;

}

write(fd\_fifo\_out, &level, sizeof(level)); //write to fifo 3 that was created in the kernel module

\*VIC2SoftInt |= 0x80000000; //Trigger software interrupt in user space

if(quitFlag != 1){

printf("WOULD YOU LIKE TO PROCEED TO LEVEL %d ? ", level);

fgets(userInput, MSG\_SIZE, stdin);

while( strncmp(userInput,"YES",3) && strncmp(userInput, "yes", 3) && strncmp(userInput,"NO",2) && strncmp(userInput, "no", 2)){

printf("Please enter \"YES\" or \"NO\"\n");

fgets(userInput, MSG\_SIZE, stdin);

}

if( !strncmp(userInput,"YES",3) || !strncmp(userInput, "yes", 3) ){

pauseFlag = 0;

printf("\n\n====================================LEVEL %d====================================\n", level);

printf("DIFFICULTY: %s - (%s)\n", difficulty, comment);

printf("START LEVEL %d...\n", level);

printf("TRY TO HIT BUTTON ON YELLOW!\n");

}

else if(!strncmp(userInput,"NO",2) || !strncmp(userInput, "no", 2)){

quitFlag = 1;

pauseFlag = 0;

printf("\n\n\n YOUR SCORE: %d\n\n", num\_of\_wins\*100);

printf(" THANK YOU FOR PLAYING");

printf("\n\n\n\n\n\n");

//printf("been here..quitFlag = %d, pauseFlag = %d\n", quitFlag, pauseFlag);

}

}

//printf("Received something: %s\n", buffer);

}

pthread\_exit(0);

}

void \*RecieveLightTimes(void \*arguments){

int fd\_fifo\_in = open("/dev/rtf/0", O\_RDWR);

int fd\_fifo\_in1 = open("/dev/rtf/1", O\_RDWR);

int fd\_fifo\_in2 = open("/dev/rtf/2", O\_RDWR);

i = 0;

int a;

while(quitFlag == 0){

//read in timestamp of when button was pressed

read(fd\_fifo\_in, &TimeOfPress, sizeof(TimeOfPress));

//Convert timestamp to all seconds and store in time\_of\_press

time\_of\_press[i] = (TimeOfPress.tv\_sec) + (TimeOfPress.tv\_usec/1000000.0);

//printf("time of press in seconds: %f\n", time\_of\_press);

//read in timestamp of when yellow light began

read(fd\_fifo\_in1, &YellowLightBegin, sizeof(YellowLightBegin));

//convert timestamp to seconds

light\_begin = (YellowLightBegin.tv\_sec) + (YellowLightBegin.tv\_usec/1000000.0);

do{

//read in timestamp of when yellow light ended

read(fd\_fifo\_in2, &YellowLightEnd, sizeof(YellowLightEnd));

light\_end = (YellowLightEnd.tv\_sec) + (YellowLightEnd.tv\_usec/1000000.0);

//printf("In loop\n");

}while(light\_end < light\_begin); //keep looping and reading timestamp until until end

//of light timestap is larger than beginngin of light timestamp

//printf("Time between presses: %f\n", time\_of\_press[i] - time\_of\_press[i-1]);

if( (time\_of\_press[i] > light\_begin) && (time\_of\_press[i] < light\_end) ){ //If player hit the button during the interval

if( i != 0 && (time\_of\_press[i] - time\_of\_press[i-1] < 3) ){ //Install a period of 3 seconds between button presses

printf("You can't hit the button repeatedly. That is cheating!!\n");

}

else{

//Player hit the button in the interval that the yellow light was on

printf("\n\n HURRAY!!! YOU DID IT!!!\n\n\n\n");

num\_of\_wins++;

sprintf(num\_wins,"$%d", num\_of\_wins); //build a string that holds number of wins

//broadcast number of wins to everyone so that server can read it

n = sendto(sock, num\_wins, strlen(num\_wins), 0,

(const struct sockaddr \*)&anybody, length);

if(n < 0)

error("sendto");

//Make thread pause until ReceiveMessageFromServer thread determines if player continued

//on to the next level

pauseFlag = 1;

while(pauseFlag == 1){

usleep(250000);

}

}

}

else{

//record misses

misses++;

sprintf(num\_miss,"@ %d", misses); //build a string that holds number of misses

//broadcast number of misses to everyone so that server can read it

n = sendto(sock, num\_miss, strlen(num\_miss), 0,

(const struct sockaddr \*)&anybody, length);

if(n < 0)

error("sendto");

}

i++;

}

pthread\_exit(0);

}

void error(const char \*msg)

{

perror(msg);

exit(0);

}

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

{

int boolval = 1; // for a socket option

char hostname[128], IP\_Address[15], IP\_Address\_token[15];

char \*received\_IP;

char game\_id[20];

int i;

int my\_ip\_last\_digit, recvd\_ip\_last\_digit;

int bufTemp[2];

char gameStatus[5];

if (argc < 2)

{

printf("usage: %s port\n", argv[0]);

exit(0);

}

gethostname(hostname, sizeof hostname); //Retrieve local hostname of computer

printf("My hostname: %s\n", hostname); //Display hostname

struct hostent \*hstnm = gethostbyname(hostname);

//Copy ip address into IP\_address buffer in the

/\*The inet\_ntoa function converts an ipv4 internet network address into an ASCII string in internet

\* standard dotted-decimal format as in the example "192.168.16.0".

\* h\_addr is the member of struct hostent that holds the host address

\*/

strcpy(IP\_Address, inet\_ntoa(\*((struct in\_addr\*)hstnm->h\_addr))); //Copy ip address into IP\_Address array

printf("My IP address is: %s\n", IP\_Address);

strcpy(IP\_Address\_token, IP\_Address);

strtok(IP\_Address\_token, ".");

for(i=0; i<3; i++){

my\_ip\_last\_digit = atoi(strtok(NULL,"."));

}

sock = socket(AF\_INET, SOCK\_DGRAM, 0); // Creates socket. Connectionless.

if (sock < 0)

error("Opening socket");

// change socket permissions to allow broadcast

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

{

printf("error setting socket options\n");

exit(-1);

}

anybody.sin\_family = AF\_INET; // symbol constant for Internet domain

anybody.sin\_port = htons(atoi(argv[1])); // port field

anybody.sin\_addr.s\_addr = inet\_addr("10.3.52.255"); // broadcast address

length = sizeof(struct sockaddr\_in); // size of structure

printf("\n");

while(1){

quitFlag = 0;

printf("Would you like to play? YES or NO? ");

fgets(userInput, MSG\_SIZE, stdin);

if( strncmp(userInput,"YES",3) && strncmp(userInput, "yes", 3) && strncmp(userInput,"NO",2) && strncmp(userInput, "no", 2)){

printf("Please enter \"YES\" or \"NO\"\n");

}

if( !strncmp(userInput,"YES",3) || !strncmp(userInput, "yes", 3) ){

strcpy(gameStatus, "#1");

sprintf(game\_id,"# %s",IP\_Address); //build a string that holds the game ID info

printf("\n\n====================================LEVEL 1====================================\n");

printf("DIFFICULTY: EASY - (LIGHTS ARE CHANGING SLOWLY)\n");

printf("START GAME...\n");

printf("TRY TO HIT BUTTON ON YELLOW!\n");

//printf("%s\n", game\_id);

//Send message to anyone there...

n = sendto(sock, gameStatus, strlen(gameStatus), 0,

(const struct sockaddr \*)&anybody, length); //send game status to client program

if(n < 0)

error("sendto");

pthread\_t thread1, thread2; //Declare thread1 and thread2 variables of type pthread\_t

pthread\_create(&thread1, NULL, RecieveLightTimes, NULL);

pthread\_create(&thread2, NULL, RecieveMessageFromServer, NULL);

pthread\_join(thread1, NULL); //Wait for thread to finish

pthread\_join(thread2, NULL); //Wait for thread to finish

}

strcpy(gameStatus, "#0");

n = sendto(sock, gameStatus, strlen(gameStatus), 0,

(const struct sockaddr \*)&anybody, length); //send game status to client program

if(n < 0)

error("sendto");

}

close(sock);

return 0;

}

Project\_TheServer.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

//#include <arpa/inet.h>

#include <pthread.h>

#include <fcntl.h>

#include <sys/mman.h>

#define MSG\_SIZE 40 // message size

struct sockaddr\_in addr;

int sock, length, n;

socklen\_t fromlen;

char userInput[MSG\_SIZE];

struct Game\_ID{

char id[15]; //The ID of the game being used

int status; //Whether game is being used or not. 1=game in use, 0=not in use

int score; //represents the user's score. Each hit is 100 points

int misses; //How many misses the user accrued

int level; //The current level the user is on

};

//declare an array of struct Game\_ID to hold the Status Report of each game

struct Game\_ID game[2];

void error(const char \*msg)

{

perror(msg);

exit(0);

}

void \*Status\_Of\_Games(void \*arguments){

int i;

printf("\n\nInstructions:\n");

printf("At any time enter \"status\" to receive a status report of games. \n");

while(1){

fgets(userInput, MSG\_SIZE, stdin);

while( strncmp(userInput,"STATUS",6) && strncmp(userInput, "status", 6) ){

printf("Please enter \"status\"\n");

fgets(userInput, MSG\_SIZE, stdin);

}

printf("\n\n\n\n\n\n\n\n\n\n\n");

for(i=1;i<=20;i++){

if(game[i].status == 1){

printf("\nSTATUS REPORT\n");

printf("=================================\n");

printf("GAME ID %s IS IN USE\n", game[i].id);

printf("CURRENT LEVEL: %d\n", game[i].level);

printf("CURRENT SCORE: %d\n", game[i].score);

printf("CURRENT MISSES: %d\n", game[i].misses);

printf("=================================\n\n");

}

}

}

pthread\_exit(0);

}

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

{

int boolval = 1; // for a socket option

struct sockaddr\_in server;

char buffer[MSG\_SIZE]; // to store received messages or messages to be sent.

srand(time(NULL)); //randomize random number generator using current time

char hostname[128], IP\_Address[15], IP\_Address\_token[15];

char received\_IP[15], received\_IP\_token[15];

int i;

int my\_ip\_last\_digit, recvd\_ip\_last\_digit;

int flag = 0;

int bufTemp[2];

char level[8];

//If user forgets to include port when executing program

if (argc < 2)

{

printf("usage: %s port\n", argv[0]);

exit(0);

}

gethostname(hostname, sizeof hostname); //Retrieve local hostname of computer

printf("My hostname: %s\n", hostname); //Display hostname

struct hostent \*hstnm = gethostbyname(hostname);

//Copy ip address into IP\_address buffer in the

/\*The inet\_ntoa function converts an ipv4 internet network address into an ASCII string in internet

\* standard dotted-decimal format as in the example "192.168.16.0".

\* h\_addr is the member of struct hostent that holds the host address

\*/

strcpy(IP\_Address, inet\_ntoa(\*((struct in\_addr\*)hstnm->h\_addr))); //Copy ip address into IP\_Address array

printf("My IP address is: %s\n", IP\_Address);

strcpy(IP\_Address\_token, IP\_Address);

//Tokenize IP\_Address to get last digit

strtok(IP\_Address\_token, ".");

for(i=0; i<3; i++){

my\_ip\_last\_digit = atoi(strtok(NULL,"."));

}

sock = socket(AF\_INET, SOCK\_DGRAM, 0); // Creates socket. Connectionless.

if (sock < 0)

error("Opening socket");

length = sizeof(server); // length of structure

bzero(&server,length); // sets all values to zero. memset() could be used

server.sin\_family = AF\_INET; // symbol constant for Internet domain

server.sin\_addr.s\_addr = INADDR\_ANY; // IP address of the machine on which

// the server is running

server.sin\_port = htons(atoi(argv[1])); // port number

// binds the socket to the address of the host and the port number

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

error("binding");

// change socket permissions to allow broadcast

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

{

printf("error setting socket options\n");

exit(-1);

}

fromlen = sizeof(struct sockaddr\_in); // size of structure

pthread\_t thread1;

pthread\_create(&thread1, NULL, Status\_Of\_Games, NULL);

while(1)

{

// to "clean up" the buffer. The messages aren't always the same length...

bzero(buffer,MSG\_SIZE);

// receive from a client

n = recvfrom(sock, buffer, MSG\_SIZE, 0, (struct sockaddr \*)&addr, &fromlen);

if (n < 0)

error("recvfrom");

strcpy(received\_IP, inet\_ntoa(addr.sin\_addr.s\_addr));

strcpy(received\_IP\_token, received\_IP);

//printf("Message came from %s\n", received\_IP);

strtok(received\_IP\_token, ".");

for(i=0; i<3; i++){

recvd\_ip\_last\_digit = atoi(strtok(NULL,"."));

}

printf("Message came from board number: %d\n", recvd\_ip\_last\_digit);

if(strncmp(buffer,"#1",2) == 0){ //if message received is a game id status, store status

strcpy(game[recvd\_ip\_last\_digit].id, received\_IP);

game[recvd\_ip\_last\_digit].status = 1;

game[recvd\_ip\_last\_digit].level = 1;

//printf("Game ID %s is in use\n", buffer);

}

else if(strncmp(buffer,"#0",2) == 0){ //if message received is a game id status, store status

game[recvd\_ip\_last\_digit].status = 0;

//printf("Game ID %s is in use\n", buffer);

}

else if( (strncmp(buffer,"$",1) == 0) ){ //if message received is a score, continue

if(strncmp(buffer, "$1", 2) == 0){

//printf("The user has scored. Broadcast to client game to move to level 2.\n");

game[recvd\_ip\_last\_digit].level = 2;

game[recvd\_ip\_last\_digit].score = 100;

//send message to the specific client (game) only

strcpy(level,"LEVEL2"); //build a string that holds number of wins

n = sendto(sock, level, strlen(level), 0,

(struct sockaddr \*)&addr, fromlen);

}

else if(strncmp(buffer, "$2", 2) == 0){

//printf("The user has scored twice. Broadcast to client game to move to level 3.\n");

game[recvd\_ip\_last\_digit].level = 3;

game[recvd\_ip\_last\_digit].score = 200;

//send message to the specific client (game) only

strcpy(level,"LEVEL3"); //build a string that holds number of wins

n = sendto(sock, level, strlen(level), 0,

(struct sockaddr \*)&addr, fromlen);

}

else if(strncmp(buffer, "$3", 2) == 0){

//printf("The user has scored 3 times. Broadcast to client game to move to level 4.\n");

game[recvd\_ip\_last\_digit].level = 4;

game[recvd\_ip\_last\_digit].score = 300;

//send message to the specific client (game) only

strcpy(level,"LEVEL4"); //build a string that holds number of wins

n = sendto(sock, level, strlen(level), 0,

(struct sockaddr \*)&addr, fromlen);

}

else if(strncmp(buffer, "$4", 2) == 0){

//printf("The user has scored 4 times. Broadcast to client game to move to level 5.\n");

game[recvd\_ip\_last\_digit].level = 5;

game[recvd\_ip\_last\_digit].score = 400;

//send message to the specific client (game) only

strcpy(level,"LEVEL5"); //build a string that holds number of wins

n = sendto(sock, level, strlen(level), 0,

(struct sockaddr \*)&addr, fromlen);

}

else if(strncmp(buffer, "$5", 2) == 0){

//printf("The user has scored 5 times. Broadcast to client game that user has won.\n");

game[recvd\_ip\_last\_digit].score = 500;

//send message to the specific client (game) only

strcpy(level,"END"); //build a string that holds number of wins

n = sendto(sock, level, strlen(level), 0,

(struct sockaddr \*)&addr, fromlen);

}

}

else if(strncmp(buffer,"@",1) == 0){

int miss;

strtok(buffer, " "); //begin tokenizing buffer by removing "@"

miss = atoi(strtok(NULL, " ")); //get next token which is the number of misses

game[recvd\_ip\_last\_digit].misses = miss;

}

else {

printf("Error: Received message is invalid. Learn how to type!\n");

}

}

pthread\_join(thread1, NULL); //Wait for thread to finish

return 0;

}