

# Multiplayer Memory Game

By

Andrew Shannon

Course Instructor:

Dr. Guilherme DeSouza

ECE 4220 Real Time Embedded Computing

Department of Electrical and Computer Engineering

University of Missouri-Columbia

Columbia, Missouri

May 10, 2012

## Abstract

This multiplayer memory game is a variation of the game Simon over multiple computers. Players must try to recreate patterns entered by other players. The game uses three LEDs and three push buttons to interact with the user. This hardware is connected to a TS-7250 embedded computer that runs the program for the game to function. The game can be played across a network with multiple other computers. Included are a kernel module, client program, and server program. Concepts used from ECE 4220 include FIFO, socket communication, interrupt, and kernel module. The project uses these concepts to provide a memory game that can be played with many people.

## Introduction and Problem Statement

The basic Simon game is very simple. A set of lights emit in a certain order and the user must recreate the pattern by pressing the corresponding buttons. Each time the user gets the pattern correct the pattern length increases, making it more and more difficult to get the pattern correct. However, this does not give you the opportunity to play with multiple people and the pattern generation is usually random. My implementation of the game gives players a chance to play with friends and to try to outsmart them by entering their own patterns. These concepts make an addition to the classic game to make it more interesting. In order to implement the order of the game, socket communication is set up to act similar to semaphores, in that they control when a process can run. The game is set up so one player enters a pattern, which is then displayed on another board. The second player must then replicate the pattern using the push buttons and, if correct, enter a new pattern. The game continues with the length of the pattern increasing each time every player has had a turn. If a player enters an incorrect pattern, he is

eliminated from the game and the winner is the last remaining player. This version of the game is limited to ten players due to the amount of TS-7250s in the lab, but the game has the potential for hundreds of players. The idea is fairly simple but the implementation requires multiple programming aspects learned in this course.

## Implementation

### Overview

The program works by having a definite order set up by the server program. There is one server that connects to all of the clients and controls the order. The client programs are connected to the auxiliary boards and run by the players. The game is run by the server first connecting to all of the clients, determining the order, and getting the first pattern. The communication between the server and client must be carefully controlled so that no clients run when they should not. Because of this constraint, I developed a set of functions that broadcast information depending on what computer's turn it is to run. These functions work similarly to semaphores in that they control the order of multiple processes. However, unlike semaphores, the server program controls the overall order so the semaphore functions cannot act asynchronously. The server sends the message for the first client to begin, then sends the pattern to display. The client program displays the pattern and then waits for the user input. If the user inputs the pattern correctly, he must then enter a new pattern for the next player. The message for the server to continue is sent along with the new pattern. The client then goes back to the waiting stage while the other clients run. The flow chart of the game is shown in Figure 1.

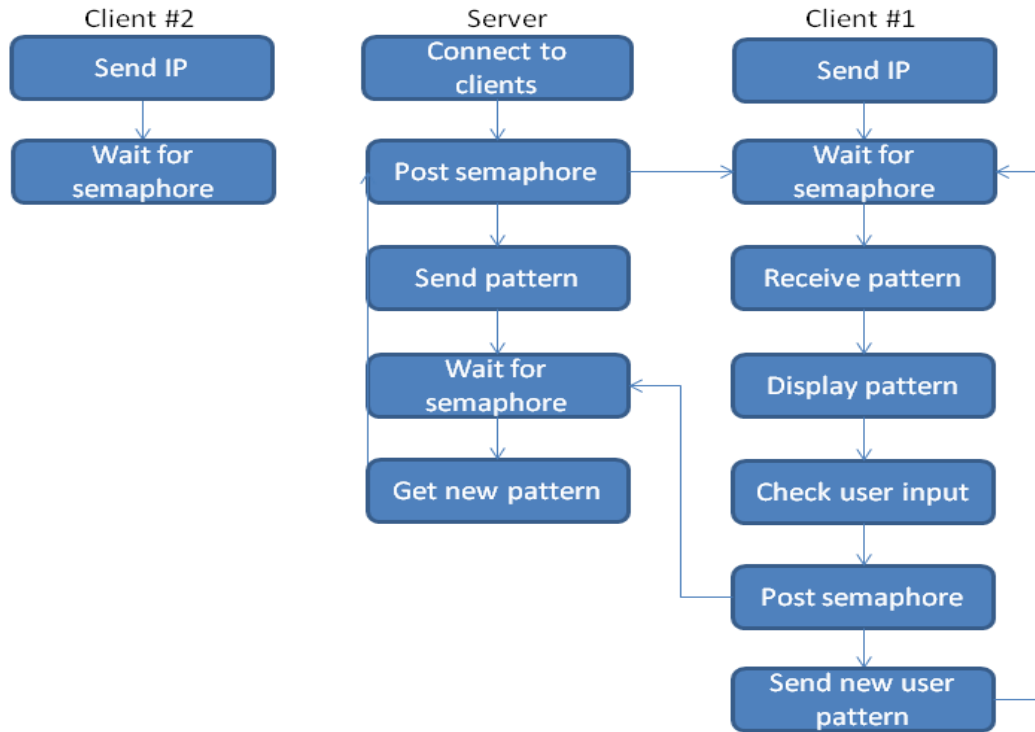


Figure 1 Flow chart

## Hardware

The hardware used for this project are all on the auxiliary board from the lab. The red, yellow, and green LEDs are used for the display, and the first three buttons are used for input from the user. The layout of the board can be seen in Figure 2.

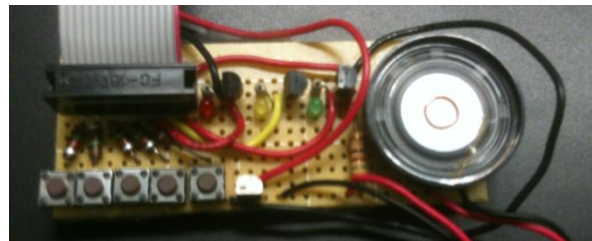


Figure 2 Auxiliary board

While simply using a computer monitor and a keyboard for output and input is possible, the board gives the feel of a controller for the game. Having different colored lights is also more appealing than shapes or text on a monitor. Most people interact with other users using a computer monitor on a daily basis. The idea of displaying another computer's input on a screen

has lost some of its luster. However, displaying and receiving data using peripheral devices is done much less frequently. After starting the program, everything can be done using only the board as a controller and display. Different LED flashes have different meanings that, once learned, can be used to tell exactly where one is in the game. The LED flash displays are as follows:

- Three flashes of all LEDs - beginning of turn,
- One flash of all LEDs - pattern is correct, enter new pattern,
- Two flashes of all LEDs - turn is over,
- LEDs off after turn - user is still in play,
- LEDs on after turn - user has been eliminated,
- Middle and outer LEDs alternating - winner.

Once the user learns the order of the game by viewing the LEDs, he can play entirely using hardware without any interaction with a computer.

## Software

The software is split up into three different files: module, client, and server. The module is used to interact with the hardware. It is run with the client program in order to read button presses. Interrupts are used to read the button to pass to the client program. The module maps to the auxiliary board using `__ioremap()` and sets up the buttons as interrupts by writing to specific registers. The module also contains the interrupt handler for the hardware interrupt. The handler checks the status of the interrupt to find which button was pressed. Then, it writes the data to a FIFO in order to communicate with the client program. The handler also has a safety check to ensure that no accidental button presses occur. The time is checked at the beginning and end of the handler and it will only run when 0.4 seconds have passed. This fixes any problems with

bouncing or buttons incorrectly sending multiple signals. The 0.4 second time was chosen because it is faster than a person can press the button twice, so there is no possibility of the button ignoring real input due to the time check. The handler then clears the interrupt and returns to waiting for user input.

The client program is loaded on players' computers. This program interacts closely with the server program as well. The client program first gets its own IP address to send to the server so the server knows which machines are playing. This is done using *popen()* to write the value of *ifconfig* to a temporary file, and then extract the address using *strtok()*. Then the client enters the main loop where it first calls the function *semaphoreWait()* in order to wait for the server to signal it to run. The other function, *semaphorePost()* is used when the client is finished. These functions are not really semaphores, but act as semaphores by making the program wait for a certain token before it continues running. In this case, *semaphoreWait()* uses the socket connection to wait until it receives the message “@<IP>” where IP is the computer’s IP address. The function *semaphorePost()* sends “!<IP>” which tells the server that the computer at that IP is finished with its turn. The actual functions can be seen in Figure 3 and Figure 4.

```
void semaphorePost(int myIP, int sock, struct sockaddr_in server, socklen_t fromlen) {  
  
    char message[MSG_SIZE];  
    bzero(message, MSG_SIZE); //clear message  
  
    //Send IP  
    message[0] = '!';  
    message[1] = myIP;  
  
    //Send message  
    if (sendto(sock, message, 40, 0, (struct sockaddr *)&server, fromlen) < 0)  
        error("Send message error");  
}
```

Figure 3 Client function semaphorePost()

```

int semaphoreWait(int myIP, int sock, struct sockaddr_in from, socklen_t length) {

    char message[MSG_SIZE];

    while(1) {
        bzero(message, MSG_SIZE); //clear message

        //Receive message
        if(recvfrom(sock, message, MSG_SIZE, 0, (struct sockaddr *)&from, &length) < 0)
            error("Receive message error");

        //Check if this computer
        if(message[0] == '@') { //check if IP message
            if(message[1]-48 == myIP) //check if this IP
                return 1;
        }
    }
}

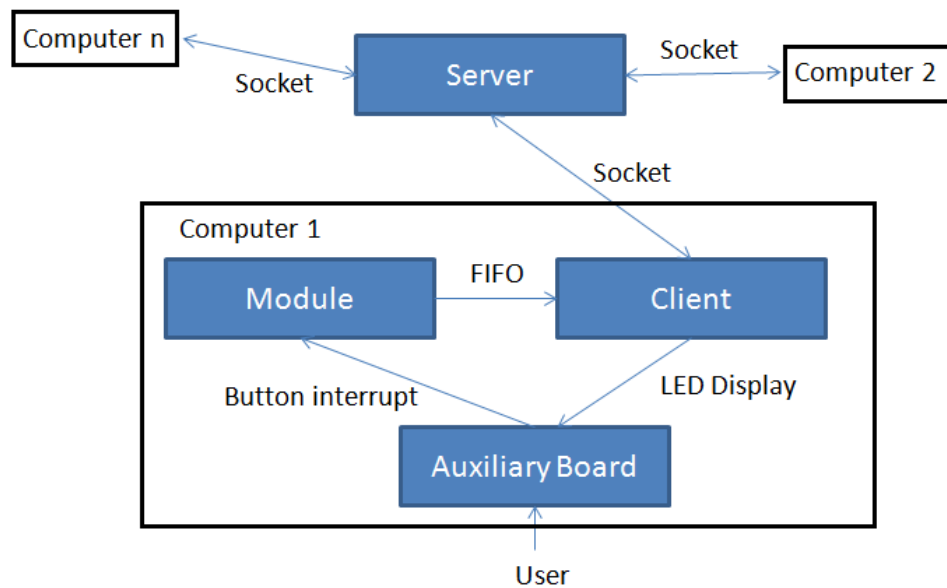
```

Figure 4 Client function semaphoreWait()

Again, these functions do not act as semaphores in that they are asynchronous, but because they control the order the programs run similar to two semaphores controlling alternating threads in a program. Next, the client program checks for a #, I, or W symbol from the server. “#” represents a new pattern which the client receives and then displays. “W” represents the game is over and the user wins. “I” represents a new round of the game, so when entering the new pattern, the length will be one more. The program interacts with the module to get the buttons from the user and check if the user enters the pattern correctly and get a new pattern. If correct, the client sends “#” followed by the new pattern. If incorrect the client sends “Fail” to the server and quits.

The server program first sets the game up by reading all of the client’s IPs and putting them in an array. Then it goes through a loop that continues until there is one computer remaining or the maximum pattern length of 30 is reached. Most of the functions of the server were all ready described in the client section. It sends a message for the client to run, sends the pattern, waits for the client to finish, and receives the new pattern before moving to the next client. If the client fails, the server removes it from the array of IPs involved in the game. If the

client that fails is the last client of the round, so the next pattern should be one button longer, the server adds a random button to the end of the previous pattern. The server compares the current iteration with the length of IPs and if it is the last client in the round, the server sends “I” along with the pattern to alert the client. The loop continues until a winner is found, then the server sends the message that the client is the winner and closes the game. The interaction between server, client, and module can be seen in Figure 5.



**Figure 5** Program interaction

## Results and Demonstration

The program functions as expected in most cases. While the program can handle up to ten players, it was only tested with two and three players at a time. Therefore, some unforeseen errors could arise when adding more players. The LEDs display patterns correctly and take user input of the buttons correctly as well. Sometimes the push buttons do not register, but this is a hardware problem, not a software problem. The buttons give similar results when tested with



other programs. It is very rare that the buttons register one press more than once, so the timing be seen in the attached video “Demonstration.MOV.” The demonstration shows one complete round with no incorrect input from the user. The pattern length incrementing can also be seen as the game starts with a length of three and eventually moves up to six. After the first round, the pattern is purposely input incorrectly, which shows that the board is removed and the previous pattern is passed onto the next board to display. These two boards play back and forth a few times until one is the winner. The LED flashes all perform correctly. The video is more easily followed if referencing the LED flash definitions listed earlier.

## Conclusion

Overall, the project works as intended. The multiplayer memory game can be played between multiple users, eliminating those who enter incorrect patterns until there is one winner. The project uses a server, client, and kernel module to implement the game. Socket communication, FIFOs, and interrupts are used to communicate between user space programs, kernel modules, and hardware devices. The TS-7250 connects with the auxiliary board to create a controller that the user can play the entire game on, without interacting with the computer. The “semaphore” functions are not truly semaphores but they act to control the flow of the game. They could be used in further programs as actual distributed semaphores if the server program made them asynchronous. This, however, does not work with this game because a consistent order is required. The demonstration shows the game working successfully. This project implements concepts learned in this class and utilizes them in a practical, user-friendly way.