EEL 4930 – Spring 2020

Lab 5: Incorporating IoT with an RTOS

OBJECTIVES

- Create a 2-player pong game played over Wi-Fi
- Learn how use UDP packets for wireless communication across a router within an RTOS
- Run up to 17 threads seamlessly with our new and improved RTOS!

REQUIRED MATERIALS

Hardware

- MSP432 Launchpad
- Sensors Booster Pack
- LED Array Module
- CC3100 Wi-Fi Booster Pack

Software

- BSP
- Lab 4 G8RTOS
- Lab 4 LCD Lib

Note: You will be working with one other partner to develop the game yourselves. The same program should be loaded onto both boards, and it is the player's choice to be either the host or client of the game using any of the buttons on the HKN Daughter Board.

Game Description:

The two-player pong game you will be creating is not your ordinary pong game! Once the game begins, there will be no active balls in the arena. Over time, more and more balls will randomly be created over time (you may use the time.h library for random variables). Initially, whenever a new ball is created, its color will be white (it doesn't belong to anyone). Once it collides with a player, the ball will take on the color of the player who hit it (that ball now belongs to that player). If an owned ball passes the other player, the owner of that ball gets a point and that ball's thread is killed (if a white ball passes, it neither hurts or benefits that player).

The max number of balls allowed in the game is defined in Game.h. To make the game even more interesting, the time interval for how often a new ball is created will be proportional to the current number of balls currently in play.

To keep track of each player's points, you will utilize the LED array. For each point the red player makes, you will light up another red LED from left to right, and vice versa for the blue player. The first player to reach the end of the LED array wins!

Once a new game is played, the winner's overall score will be incremented. The number of games won by each player will be displayed on the left side of the screen.

Introduction to Game.h

The header file Game.h that is provided to in the repo provides many helpful macros that you might find useful to you. Additionally, recommended prototypes for threads and functions are also defined with detailed descriptions for each (you're welcome). Please look over Game.h before reading the rest of the lab document to better understand how to approach programming the game.

Introduction to CC3100 Support Package

The CC3100 Support Package has 3 functions ready to use:

- `void InitCC3100 (playerType playerRole)`
- `void SendData (_u8 *data, _u32 IP, _u16 BUF_SIZE)`
- `_i32 ReceiveData (_u8 *data, _u16 BUF_SIZE)`

InitCC3100

This function takes in the argument "playerRole" which is either client or host, per the enumerator defined in Game.h. Depending on the role, the CC3100 will be initialized accordingly into its default state.

Host Initialization:

Each team has been assigned a specific static IP address for the host. The HOST_IP_ADDR #define needs to be modified per your assigned IP in the CC3100_usage.h file.

Client Initialization:

The client will obtain a dynamically assigned IP address from the router using a DHCP request. This way, we have ensured that there are no conflicting IP addresses between devices. Ideally, we would use the MDNS IP broadcasting protocol so that the host's IP address would be assigned in a similar way, but unfortunately the CC3100 does not support MDNS between the same two devices just yet.

SendData

This function takes in a pointer to a uint8_t array to be sent out to another device, specified by the IP address argument. The port that the data is sent out is statically assigned in the header file. Finally, the BUF_SIZE parameter is the number of bytes to be sent. This number can be calculated by using the C function sizeof(x). The first time SendData is called, a dedicated socket is opened specifically for sending, and is kept open until the CC3100 is configured into its default state again.

ReceiveData

This function is a bit more complicated than SendData. The reason being is that you would like other threads to run while there is no data to be read. To do this, we initialize the receiving socket to have a timeout. This timeout parameter is used by the sl_select() function, which determines whether or not there is data to be received. If true, the data is read into a buffer, otherwise, it returns, allowing the OS to run other threads while waiting for data.

For UDP packets, data is sent in bytes. For this reason, you can simply cast your structs as uint8_t pointers when passed into either function.

Lab 5: Incorporating IoT with an RTOS

Handling UDP Packets

You'll notice in Game.h that there is a struct called GameState_t. This struct holds all the information a host needs to send out to the client. **Note:** All the structs used for sending and receiving are all aligned by 1 byte, where the order of each parameter in the struct begins with the datatype with the most number of bytes, and goes down in decreasing order. Otherwise, the packets are packed inefficiently resulting in a larger data structure, and we begin to experience lag in the game since the game state drawn by the client is sent out by the host.

As the host, you need to fill the packet with the necessary information to be sent out to the client. Similarly, as the client, you will need to empty the packet into the appropriate global variables. This allows us to only have hold of the CC3100 semaphore while reading/writing as opposed to holding onto both. I recommend creating two static inline functions for filling and emptying packets.

Collisions between Players and Balls

You are free to detect the collisions between the paddles and balls any which way you want, however we recommend utilizing the Minkowski algorithm to do so, which is described in the appendix section of this lab document. The minimum requirement is that it works efficiently. I also recommend giving the paddles a virtually longer length than they appear. This gives the collisions some "wiggle room" to appear as if the ball is colliding with the extremities of the paddles. You will split the paddles into thirds for determining the direction of the velocity of the ball once it collides with the paddle. This can be done by negating the x or y velocity of the ball accordingly. If the ball hits the left third of the paddle, it should move left once it changes direction, and vice versa for the right third of the paddle. If it hits the center, it should go either up or down (depending on if it hits the bottom or top player).

Drawing the Moving Balls

The balls will be a simple NxN square (you choose). To move the ball, you will add its center X and Y coordinate to its corresponding X and Y velocities, draw the rectangle accordingly using your DrawRectangle function. To move the ball, you will need to erase the ball at time *t-1* (draw a black rectangle), and draw a new colored rectangle at the coordinates at time *t.* The centers are moved by using an x and y velocity, with values of either -1 or 1 initially (these values can be initially randomly set, but are changed and multiplied by some scalar to increase its speed once a ball hits a paddle). Note: only the direction should change when the balls hits the walls, not the speed.

Drawing the Moving Paddles

The best way to draw and redraw the paddles is to use the DrawRectangle function. However, if you try to erase the entire previous paddle and redraw the new one, this process will be too slow and you will see the paddle flicker between black and the paddle's color. To solve this problem, you only need to erase the difference in the paddle's tail, and draw the difference in the paddle's head (The tail refers to the side of the paddle opposite to the direction the paddle is moving, and vice versa with the head). This way, we are not required to erase and draw the entire paddle, only its extremities per its delta position. You'll notice in the Game.h file that there is a #define called PRINT_OFFSET. This is used to print more of the background on the opposite side of the direction the paddle is moving. Without this, you might encounter some blips cause by previously deleted balls, so you might find this idea useful.

Initial Game State and Board size

The arena size for the game is included in Game.h. It will be a square, where the sides of the arena are defined by two vertical white lines. The initial score for each player will be "00". This number should be displayed on the bottom left and top left side of the game (to the left of the white line). Additionally, they should be the same color as the player. Once someone wins a game, their score should be incremented and shown in the next game. The players should also start in the center of the screen upon starting the game.

High-level Thread Descriptions

NOTE: Remember to use semaphores appropriately

Common Threads:

DrawObjects:

- Should hold arrays of previous players and ball positions
- Draw and/or update balls (you'll need a way to tell whether to draw a new ball, or update its position (i.e. if a new ball has just been created – hence the alive attribute in the Ball_t struct.
- Update players
- Sleep for 20ms (reasonable refresh rate)

MoveLEDs:

- Responsible for updating the LED array with current scores

Host Threads:

CreateGame:

- Only thread created before launching the OS
- Initializes the players
- Establish connection with client (use an LED on the Launchpad to indicate Wi-Fi connection)

    - Should be trying to receive a packet from the client
    - Should acknowledge client once client has joined
- Initialize the board (draw arena, players, and scores)
- Add the following threads:
    - GenerateBall
    - DrawObjects
    - ReadJoystickHost
    - SendDataToClient
    - ReceiveDataFromClient
    - MoveLEDs (lower priority)
    - Idle
- Kill self

GenerateBall:
- Adds another *MoveBall* thread if the number of balls is less than the max
- Sleeps proportional to the number of balls currently in play

MoveBall:
- Go through array of balls and find one that's not alive
- Once found, initialize random position and X and Y velocities, as well as color and alive attributes
- Checking for collision given the current center and the velocity
- If collision occurs, adjust velocity and color accordingly
- If the ball passes the boundary edge, adjust score, account for the game possibly ending, and kill self
- Otherwise, just move the ball in its current direction according to its velocity
- Sleep for 35ms

ReadJoystickhost:
- You can read the joystick ADC values by calling *GetJoystickCoordinates*
- You'll need to add a bias to the values (found experimentally) since every joystick is offset by some small amount displacement and noise
- Change Self.displacement accordingly (you can experiment with how much you want to scale the ADC value)
- Sleep for 10ms
- Then add the displacement to the bottom player in the list of players (general list that's sent to the client and used for drawing) i.e. players[0].position += self.displacement
- By sleeping before updating the bottom player's position, it makes the game more fair between client and host

SendDataToClient:
- Fill packet for client
- Send packet
- Check if game is done
    - If done, Add EndOfGameHost thread with highest priority
- Sleep for 5ms (found experimentally to be a good amount of time for synchronization)

ReceiveDataFromClient:
- Continually receive data until a return value greater than zero is returned (meaning valid data has been read)
    - Note: Remember to release and take the semaphore again so you're still able to send data
    - Sleeping here for 1ms would avoid a deadlock
- Update the player's current center with the displacement received from the client
- Sleep for 2ms (again found experimentally)

EndOfGameHost:
- Wait for all the semaphores to be released
- Kill all other threads (you'll need to make a new function in the scheduler for this)
- Re-initialize semaphores
- Clear screen with the winner's color
- Print some message that waits for the host's action to start a new game
- Create an aperiodic thread that waits for the host's button press (the client will just be waiting on the host to start a new game
- Once ready, send notification to client, reinitialize the game and objects, add back all the threads, and kill self

Client Threads:

JoinGame:
- Only thread to run after launching the OS
- Set initial SpecificPlayerInfo_t strict attributes (you can get the IP address by calling getLocalIP()
- Send player into to the host
- Wait for server response
- If you've joined the game, acknowledge you've joined to the host and show connection with an LED
- Initialize the board state, semaphores, and add the following threads
    - ReadJoystickClient
    - SendDataToHost
    - ReceiveDataFromHost
    - DrawObjects

    - MoveLEDs
    - Idle
- Kill self

ReadJoystickClient:

- Read joystick and add offset
- Add Displacement to Self accordingly
- Sleep 10ms

SendDataTohost:

- Send player info
- Sleep for 2ms

ReceiveDataFromHost:

- Continually receive data until a return value greater than zero is returned (meaning valid data has been read)
    - Note: Remember to release and take the semaphore again so you're still able to send data
    - Sleeping here for 1ms would avoid a deadlock
- Empty the received packet
- If the game is done, add *EndOfGameClient* thread with the highest priority
- Sleep for 5ms

EndOfGameClient:

- Wait for all semaphores to be released
- Kill all other threads
- Re-initialize semaphores
- Clear screen with winner's color
- Wait for host to restart game
- Add all threads back and restart game variables
- Kill Self

Note: These are high-level thread descriptions. It is your job to structure the flow of the program, handle the gameplay, and use appropriate semaphores.

Appendix

Minkowski Algorithm for Collision:

```
Int32_t w = 0.5 * (A.width() + B.width());
Int32_t h = 0.5 * (A.height() + B.height());
Int32_t dx = A.centerX() - B.centerX();
Int32_t dy = A.centerY() - B.centerY();

if (abs(dx) <= w && abs(dy) <= h)
{
   /* collision! */
   Int32_t wy = w * dy;
   Int32_t hx = h * dx;

   if (wy > hx)
      if (wy > -hx)
         /* collision at the top */
      else
         /* on the left */
   else
      if (wy > -hx)
         /* on the right */
      else
         /* at the bottom */
}
```