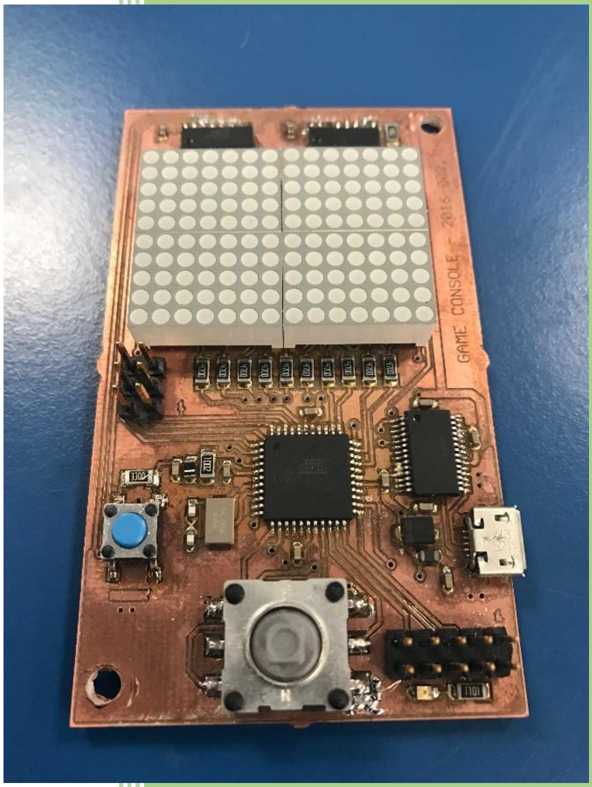
2016

Project Report For Game-Console



Jimmi Andersen, Marek Mikitovic and David Papp

Semester Project 4th Semester

16-12-2016

Table of Content

Abstract

This project addresses the learning experience of creating a two-player cross platform game. This report covers all the steps taken from the introduction of the system until the results and conclusion. In the first section of this report, the introduction provides a thorough description of the case, the purpose the system serves and how it will be relevant. Documentation of the analysis process follows, including the requirements made from the project description, the use cases created based on those requirements, the diagrams and descriptions regarding the systems tasks is found in the "Design" section. The "Implementation" chapter gives an overview of how a few Tasks work. Our collective opinion on the outcome of our Game system and the project is described in the "Discussion & Conclusion" section.

Introduction

Ping Pong Two Player Game has been created by a small group of students at VIA University College. The reason for this game is to help learn how to make small embedded systems and programming over different platforms using serial connections using a USB port. The task is to develop and program a small micro controller board and use it for making a small game of Ping Pong. Players will be implemented as player one and is controlled by the joystick on the micro control unit (further referred as MCU) and player two is controlled by the keyboard on a computer using the USART connection. The game will be displayed on the small dot-matrix display. There will not be displayed any graphical user interface on the computer part because in the future we would like to use two MCU for playing the Ping Pong game using only the computer as a relay station for transmissions between the MCU.

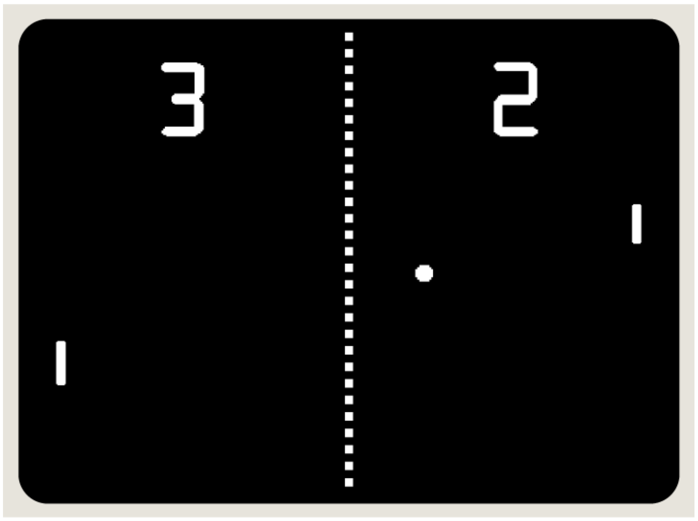
Analysis

The first step of developing our system is the analysis. All the stated requirements are read, analyzed, and discussed. The requirements are used to make tasks and then task diagrams to provide a decent overview of the actions the system should do when completed.

Requirements

The game we choose to create is based on *"Pong"* game made by Atari in 70's. It's a two player game in which players are trying to bounce of the ball by moving their avatars in our case pads on left and right side of the screen. Ball is flying around the screen and bouncing among the players, top and bottom of the screen. Point of the game is not letting the ball reach edge of the screen behind the player. Player one is controlling the pad using the up and down press on joystick on the Game-console board. Player two is using navigation keys on keyboard of PC.

Pong game created by Atari



Because we are limited by the resolution of the screen. We are not going to show real time representation of the score. It's going to be displayed after end of the game round as graphical bar representation on both sides. Also players are represented by two light up dots on the next to each other.

Having a clear list of requirements is here to provide the good overview on how to start creating the system and how to know when it is going to be finished.

Functional:

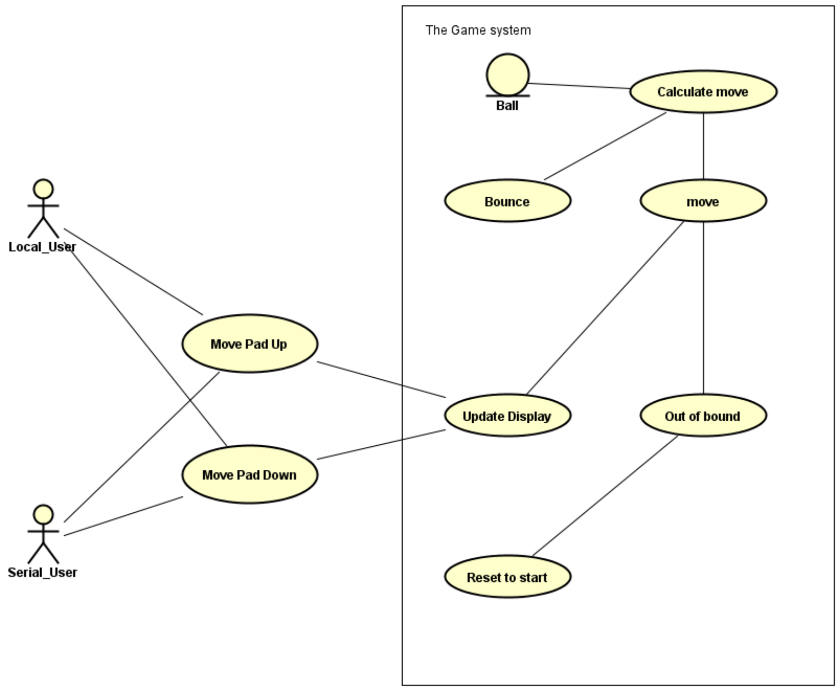
- System must use 3 tasks.
- System must have 2 real time tasks.
- System must have 2 task sharing resources.
- The system must use semaphores or mutexes.
- The pc must be able to control player two on game board.
- The system must give real time guarantee.

Non-functional:

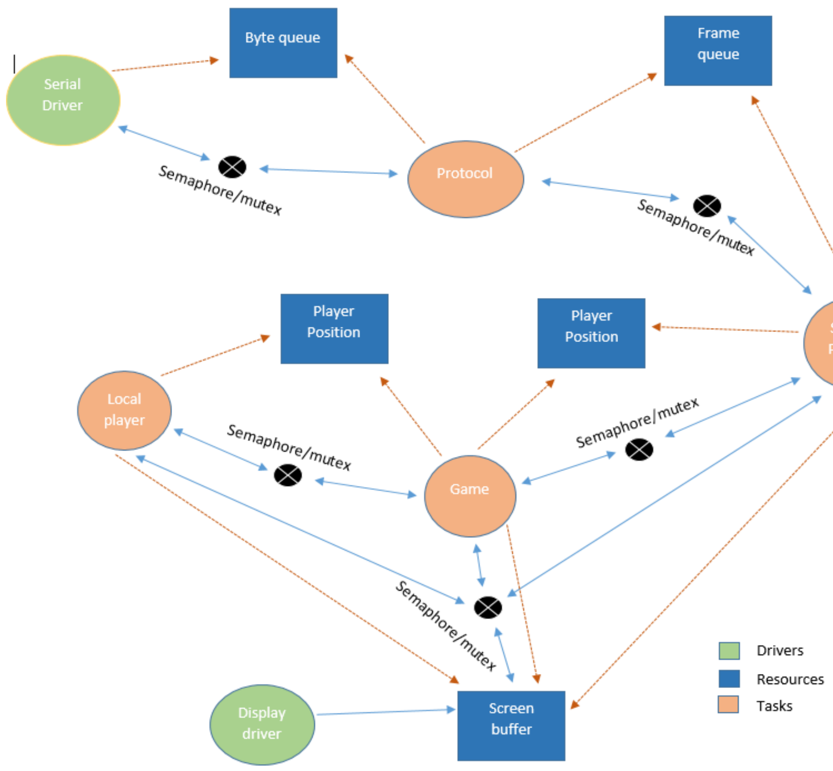
- Game must display the game on computer screen.
- Game gives score when ball hits paddle
- Game resets when ball is out of bounds

Use-Case diagram

After taking all the requirement into consideration, eight Use-Cases were made. There are two actors which is users of the system. The following diagram represents all the Use-Cases the users can operate. The users can only move up or down they have no control over everything else in the system.



Task Diagram



Design

The design part describes in detail what our different tasks in the game system are doing and why we chose to use this design over another. For better understanding the task diagrams are shown under the description.

Game Protocol

In this project, we have used a protocol design that ensures the security of the system by sending acknowledgments to the sender for every message received. The system also uses timers to ensure the messages are received synchronously. To explain further what the system does is when sender is sending the message it uses a timer for handling the fact that the system does not send or the system has some interrupted communication of the ACK (message acknowledgment) if the sender manages to throw a timeout exception we then resend the message to the receiver. The sender listens after the ACK or the NACK (message not Acknowledged) if the sender receives the ACK it will send the next message to the receiver on the other hand if the sender receives the NACK it will send the current message again until ACK is received this gives you the security of knowing that the system will receive all messages synchronously. This kind of protocol design will take more computation time then the simple protocol design where you just send messages as fast as possible for instance if you don't care if the message is correct on the receiver side you just care that the system receives the messages. Game Protocol Diagram

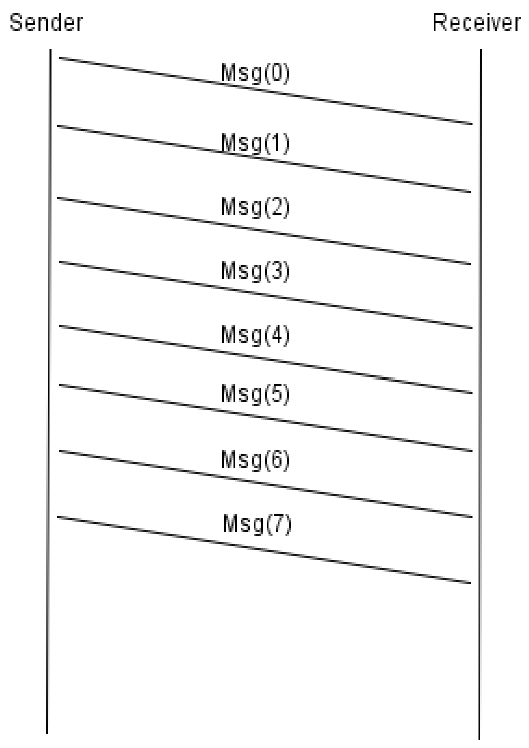


Diagram 1 Simple Protocol

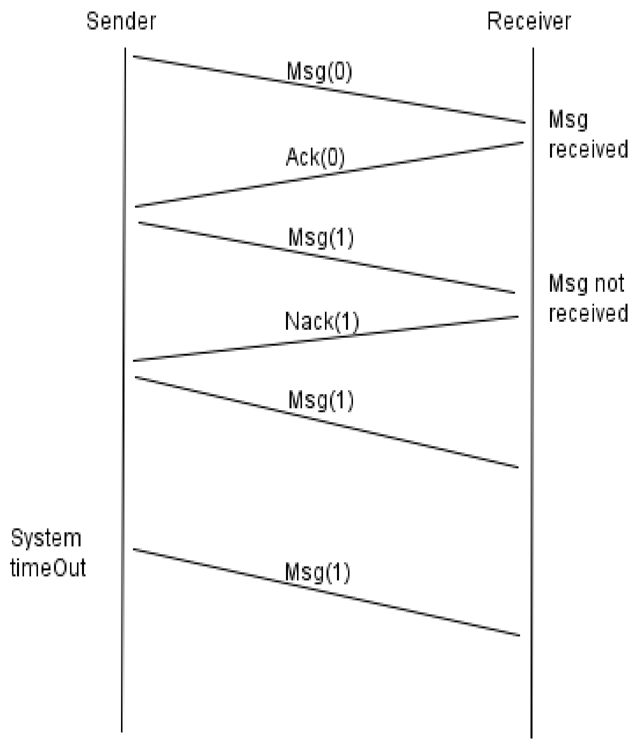
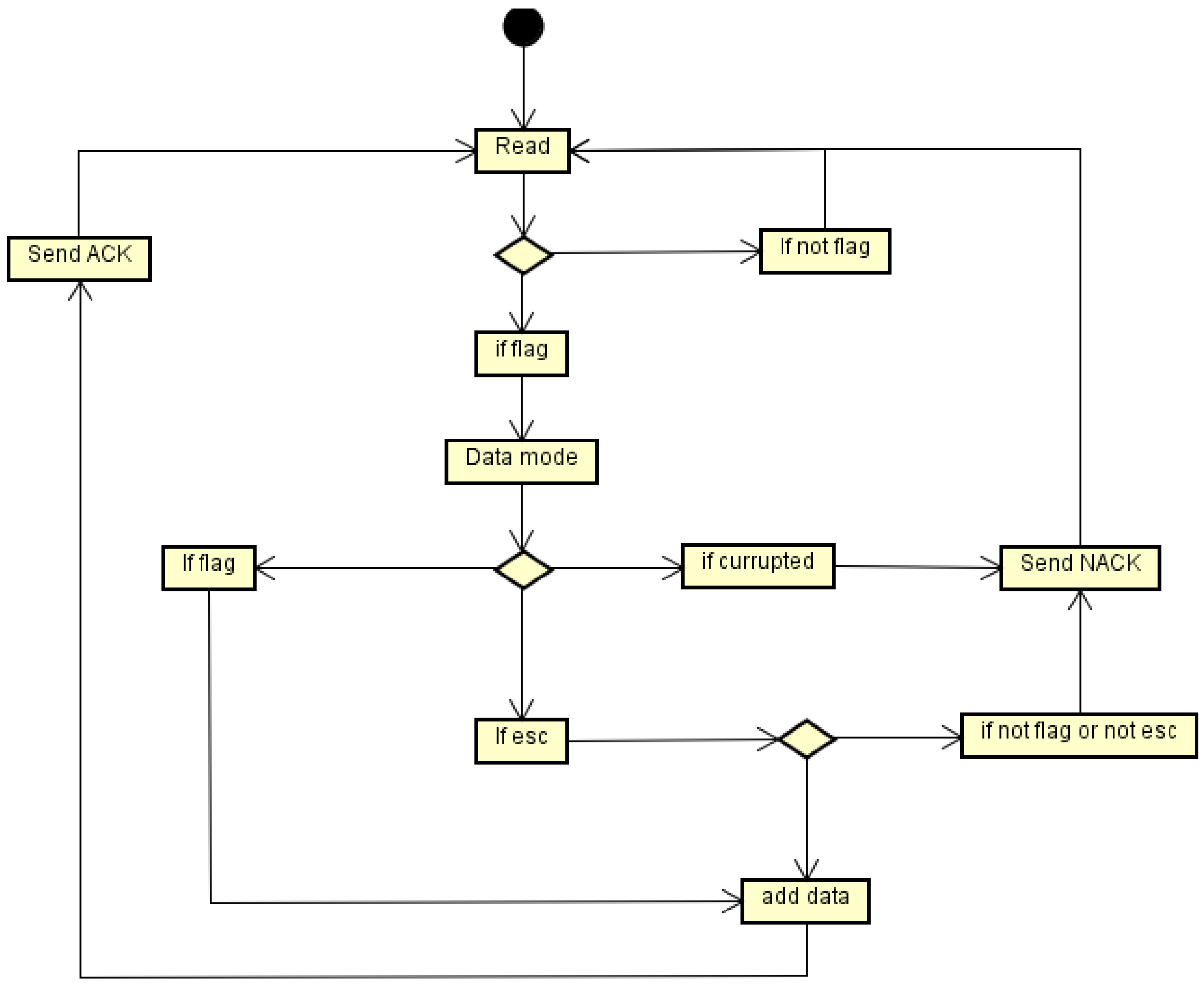


Diagram 2 Secure system protocol

Task 1 Serial Connection

The Serial connection is the task that takes care of the protocol part of communication between the Micro Control Unit (MCU) and the Computer. What it does is that it reads the byte queue applied from the Serial driver and if the first byte is the flag it goes to data mode, if it does not read a flag it will halt at read mode until the flag is read. In data mode, it will start reading the queue and here it will read until flag if it reads a flag it will keep the payload and send the ACK back to the sender. When the data mode receives an ESC, it will check for the flag or another ESC if nothing it will send the NACK to the sender, otherwise, it will add the data and send the ACK to the sender. If the data mode is corrupted it will send the NACK to the sender in the corrupted part, we could have used the CRC but we decided not to do that because we know what the messages are supposed to be and therefore do not need the CRC.

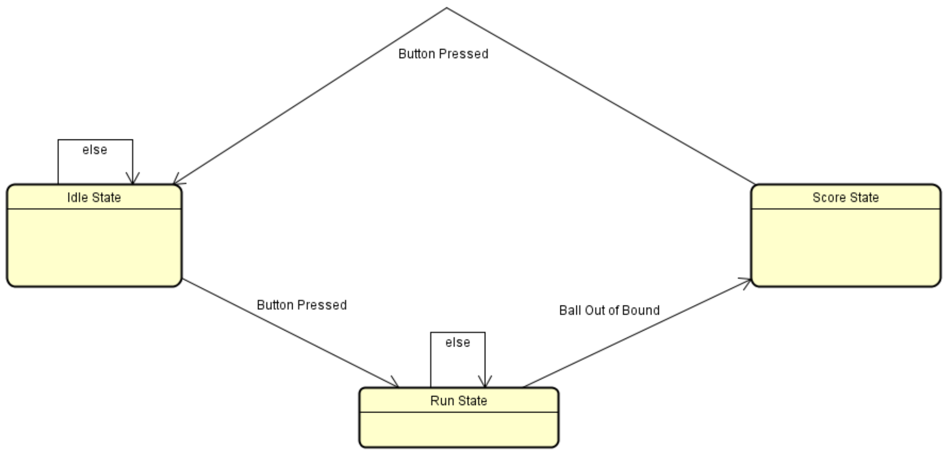
Task 1 Serial Connection Diagram



Task 2 Game State Machine

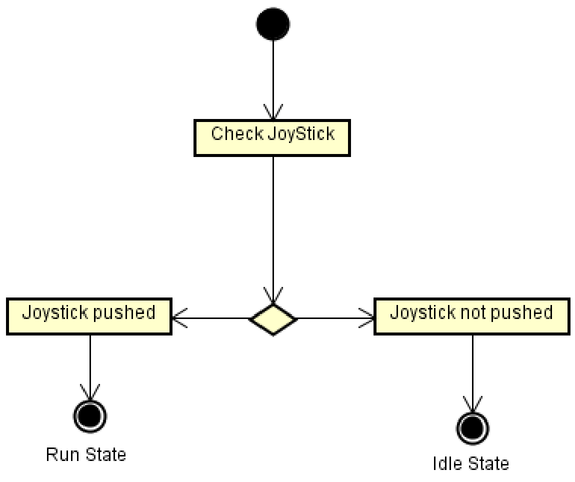
The Game task is implemented as the state machine. The task will be in idle state until the joystick is pressed and then it will go to the run state, where the ball and game are updated. If the ball gets out of the frame it will go to score state and add points to the player. In the score state, it waits for the button to be pressed before it goes back to idle state.

Game State Machine Diagram

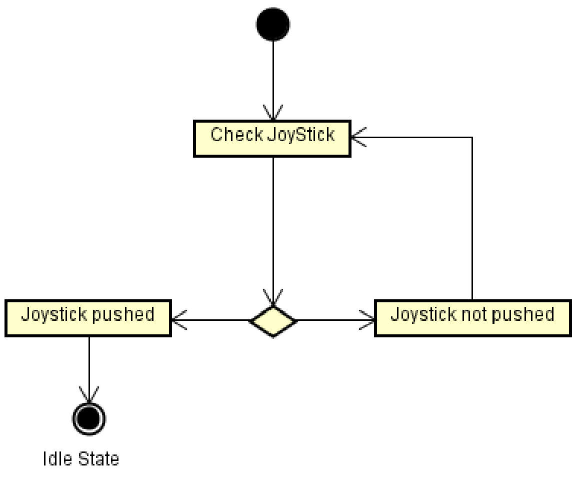


This is the activity diagram displaying the idle and the score state. The idle state is listening for the joystick to be pressed and if it is pressed it will go to run state, if it is not pressed it will return the idle state to the Game task where the loop will make it run again, therefore there are no loops in the idle state like in the score state. In the score state, we have a loop to take care of the not pressed state it will just loop until pressed and then go to idle state.

Idle State Diagram.

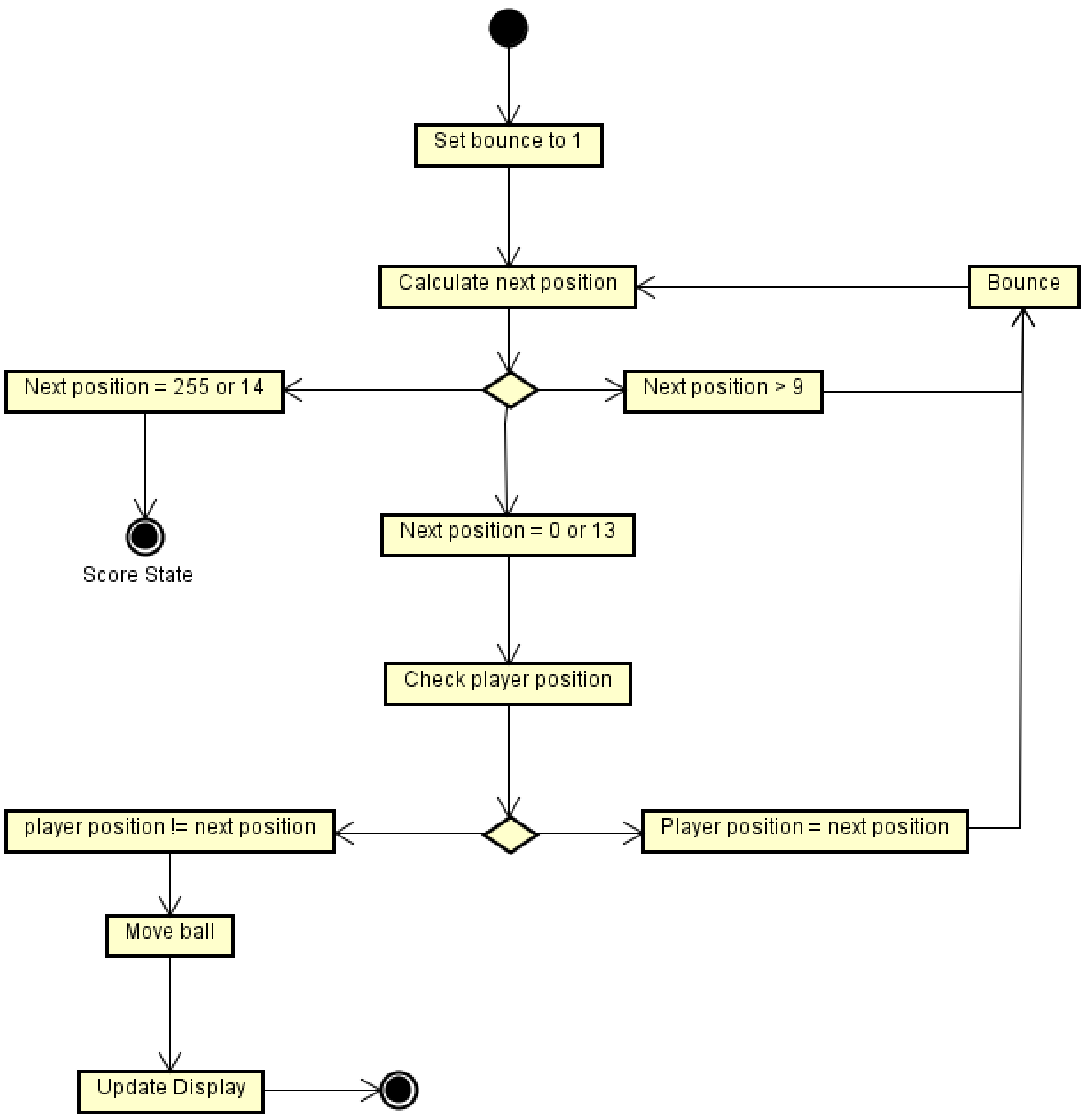


Score State Diagram



The run state is displayed below. It is more complex then idle and score state. When the run state starts, it sets the bounce to 1 to get into the loop. Then it calls the calculate next position function which gives the (x, y) coordinates for the ball to travel to. Then it checks the x coordinate and if this is equal to 14 or 255 then the ball will be out of the frame and go to score state. If the x coordinate is equal to 0 or 13 it will check the player's position and if the next position is the same as the player's position, then it will bounce back and start calculating a new position. If it is different from a player position, then it will move the ball and update display. When checking the coordinates if the y coordinate is more than 9 the ball will bounce back because it has, hit the frame top and if the y coordinate is less than 0 it will bounce because of the lower frame wall.

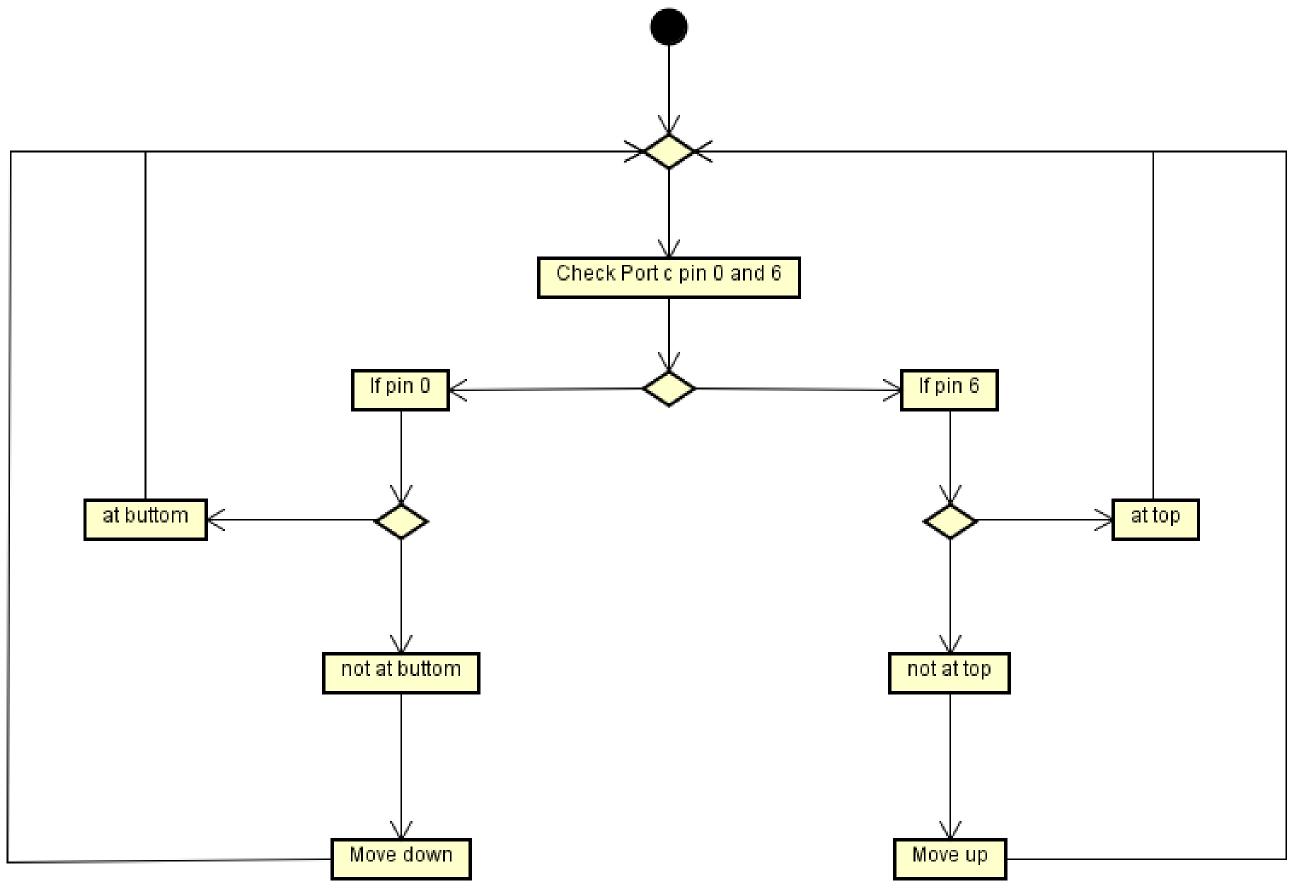
Run State Diagram.



Task 3 Player on board

Task 3 or as it is called in the system, Board player is used to read the input from the player using the boards joystick to control one of the pads of the ping pong game. What it does is that, when it's called the task checks what the input value of port c on the microcontroller. If the pin 6 on the port C is high, that means the joystick has been pressed upwards. Task then checks the current position of pads and if the pad is at the top border of the playground, it will do nothing and continue looping the task. If there was an downward press from the joystick, but the position of the player pad is not at the top most position, it will move one place up and then loop back to the beginning of the task to listen for new input from user. If pin 0 of port C is high at the beginning, we do the same as described for the upward movement but for the downward movement.

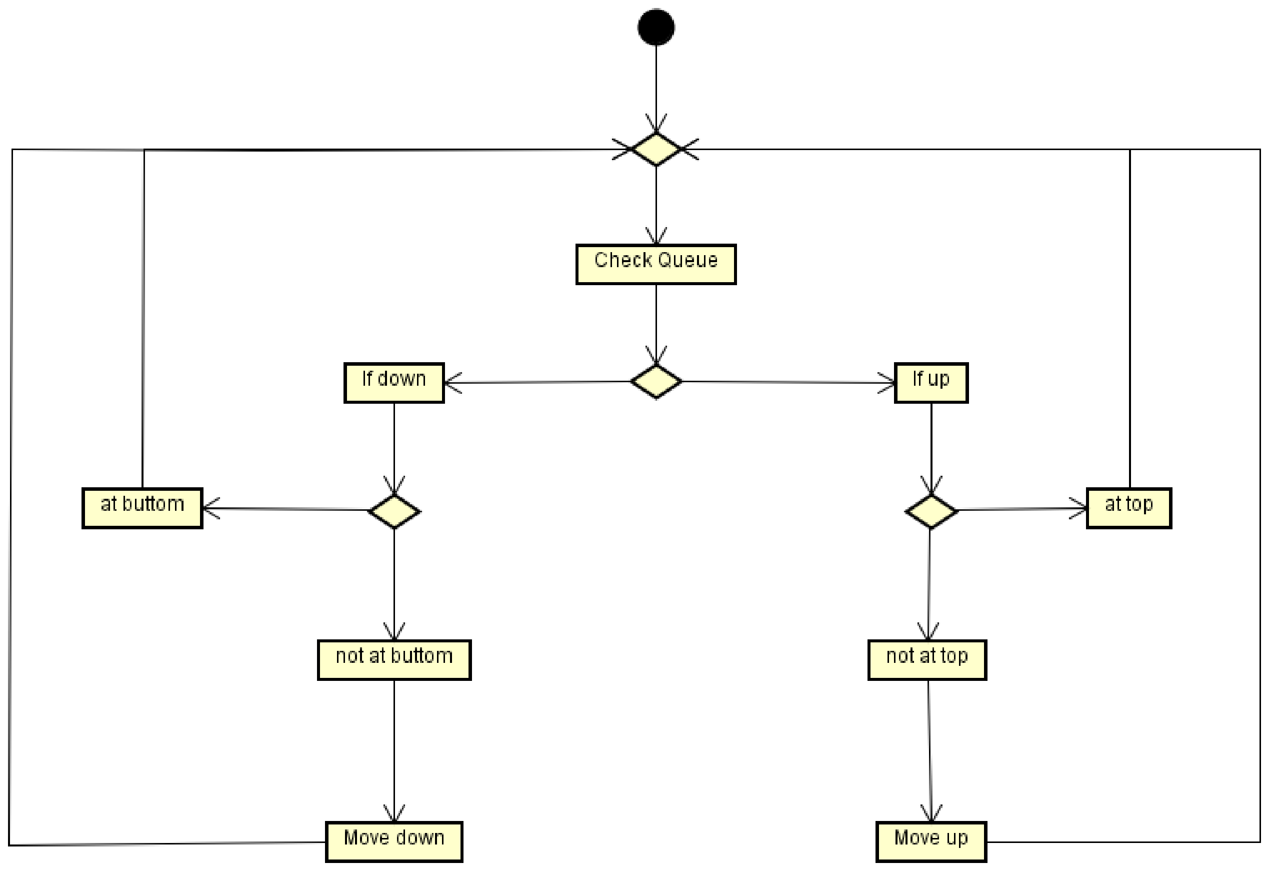
Task 3 Player on board Diagram



Task 4 Player Pc

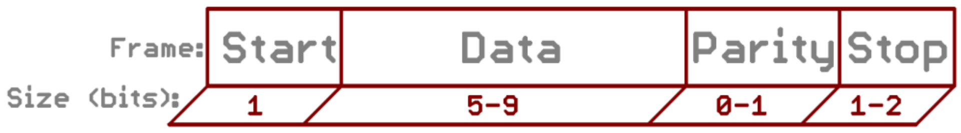
Task 4 or as it is called in the system, Serial player is used for reading the input for player two movement from the pc´s keyboard using the up and down arrows keys. What this task does is the same as previously described in task 3 but instead of listening for port C the task checks the message queue for the upward key stroke and the downward keystroke the message queue is derived from the serial connection task described earlier in this report.

Task 4 Player Pc Diagram.



Deadlines and priorities

The task with the smallest period will be the serial task. The original configuration for the serial connection is using 115200 baud rate, with no parity and one stop bits. This result in an effective data rate:



$$\frac{115200\ baud}{10\ bits} = 11520\ bytes/second$$

This means, that the serial ports rx register will be full in every 86.81 µs as:

$$\frac{1s}{11520} = 8.681 * 10^{-5}s$$

The queue for the serial driver is 30 bytes deep, therefore the empty queue will be filled in:

$$30 * 86.81\mu\text{s} = 2604.3\ \mu\text{s}\ \approx 2.6\ \text{ms}$$

The deadline for the serial task should be less than 2.6ms to prevent data loss. As the periods in FreeRTOS can be setup with system ticks and the given configuration uses 1000 ticks/s rate, that results in 1ms/tick, the period can only be either 1ms or 2ms. The 2ms is more preferable as it leaves more computation time for other tasks as its own computation time is expected to be minimal, even though the expected depth of the queue is around 23-24 bytes at continuous serial transmission. To guarantee that the serial task always gets the needed computation time, it should be the highest priority task.

The game tasks period should be set according to the desired ball speed. The game task however also controls the two player tasks and uses all of their shared resources, it is ideal for the task to have higher priority than them, to prevent deadlocks.

The two game tasks are non-critical. Their period should be set according to their desired movement speed. As they share the screen resource it would be good to have one of them at higher priority than the other.

The conclude this:

1.	Serial task	Priority: 4	Period: 2ms
2.	Game task	Priority: 3	Period: to be determined
3.	Local player task	Priority: 2	Period: to be determined
4.	Serial player task	Priority: 1	Period: to be determined

If the serial task uses so much computation time that it can empty the queue before the next cycle, or the other tasks can get enough computation time to finish, the baud rate has to be changed to smaller. Halving the baud rate would result in doubled deadline.

Both players and the game task are updated once per period, thus the relationship between the movement speed of the ball and the pads and their tasks' period is the following:

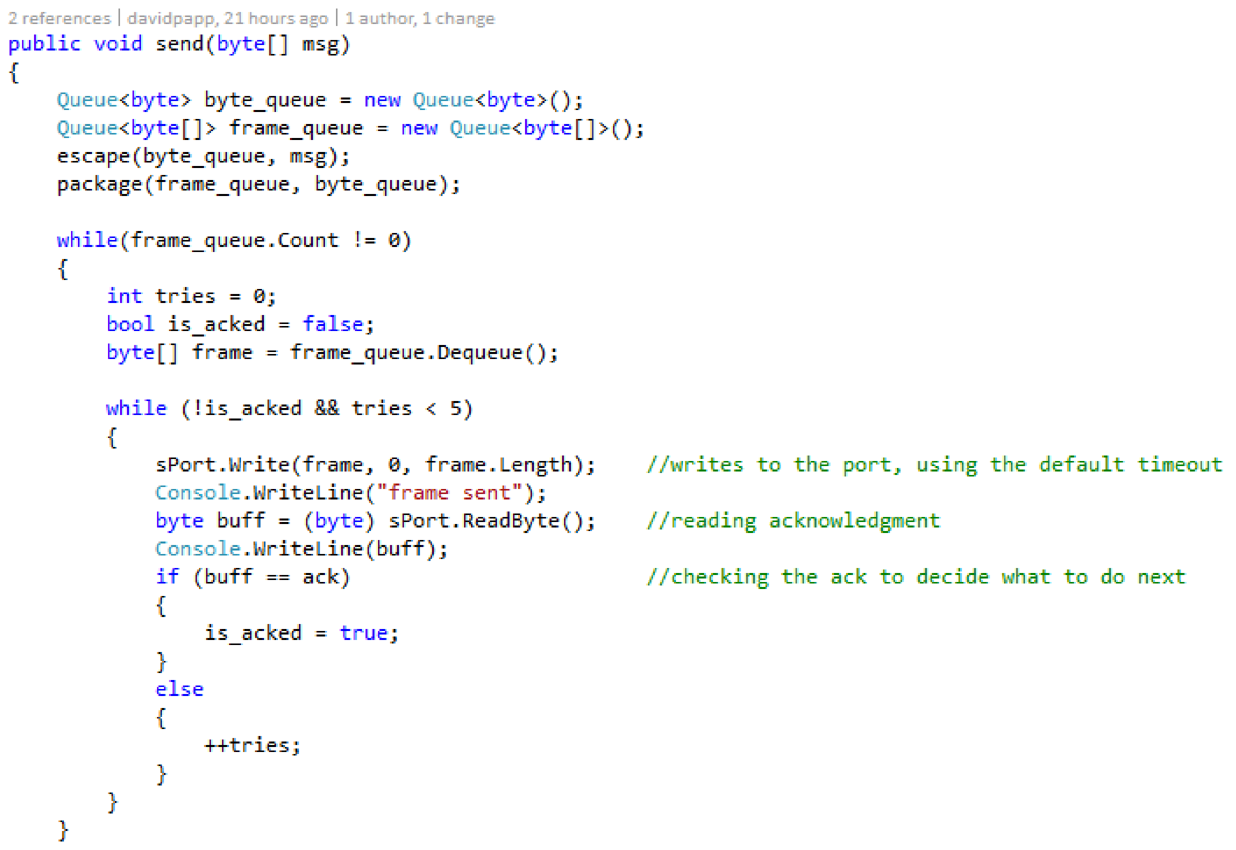
$$\frac{1s}{T_{period}}\ pixel/s$$

Implementation

The implementation was done in C utilizing the given FreeRTOS port for the game console part and C# with .NET libraries for the PC part.

PC protocol code

The PC part is implementing the protocol as a class that wraps around the .NET Syste.IO.Ports.SerialPorts class. It has only one public method besides the constructor, that handles takes the input, prepares it and writes it to the serial port as defined by the protocol. The receiver part on the PC is not implemented, as it isn't need for the project.

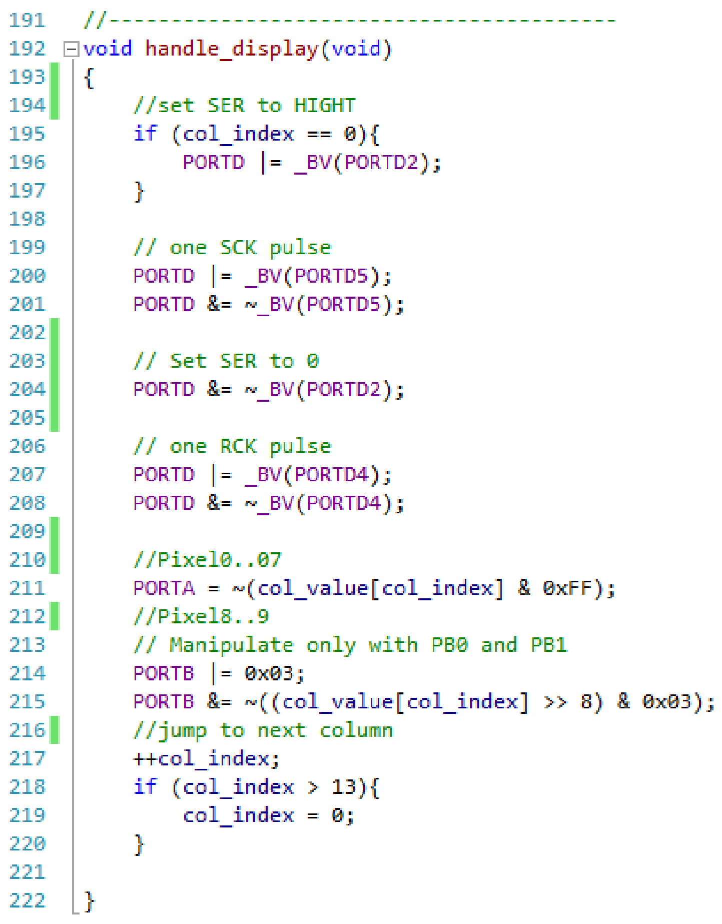


```
2 references | davidpapp, 21 hours ago | 1 author, 1 change
public void send(byte[] msg)
{
    Queue<byte> byte_queue = new Queue<byte>();
    Queue<byte[]> frame_queue = new Queue<byte[]>();
    escape(byte_queue, msg);
    package(frame_queue, byte_queue);

    while(frame_queue.Count != 0)
    {
        int tries = 0;
        bool is_acked = false;
        byte[] frame = frame_queue.Dequeue();

        while (!is_acked && tries < 5)
        {
            sPort.Write(frame, 0, frame.Length);    //writes to the port, using the default timeout
            Console.WriteLine("frame sent");
            byte buff = (byte) sPort.ReadByte();    //reading acknowledgment
            Console.WriteLine(buff);
            if (buff == ack)                        //checking the ack to decide what to do next
            {
                is_acked = true;
            }
            else
            {
                ++tries;
            }
        }
    }
```

Display Handler code

Here it can be seen how was the send method implemented. As it can be seen, every frame is tried maximum 5 times, after unsuccessfully try, the frame will be thrown away and the next one will be tried. This implementation is really simple, because both the Write() and ReadByte() method in System.IO.Ports.SerialPort are synchronous, thus the Send() method will hang up until they are finished. This implementation has a downside, that the frames can only be sent one-by-one. This also makes the labeling each frame unnecessary for identification for acknowledgement.

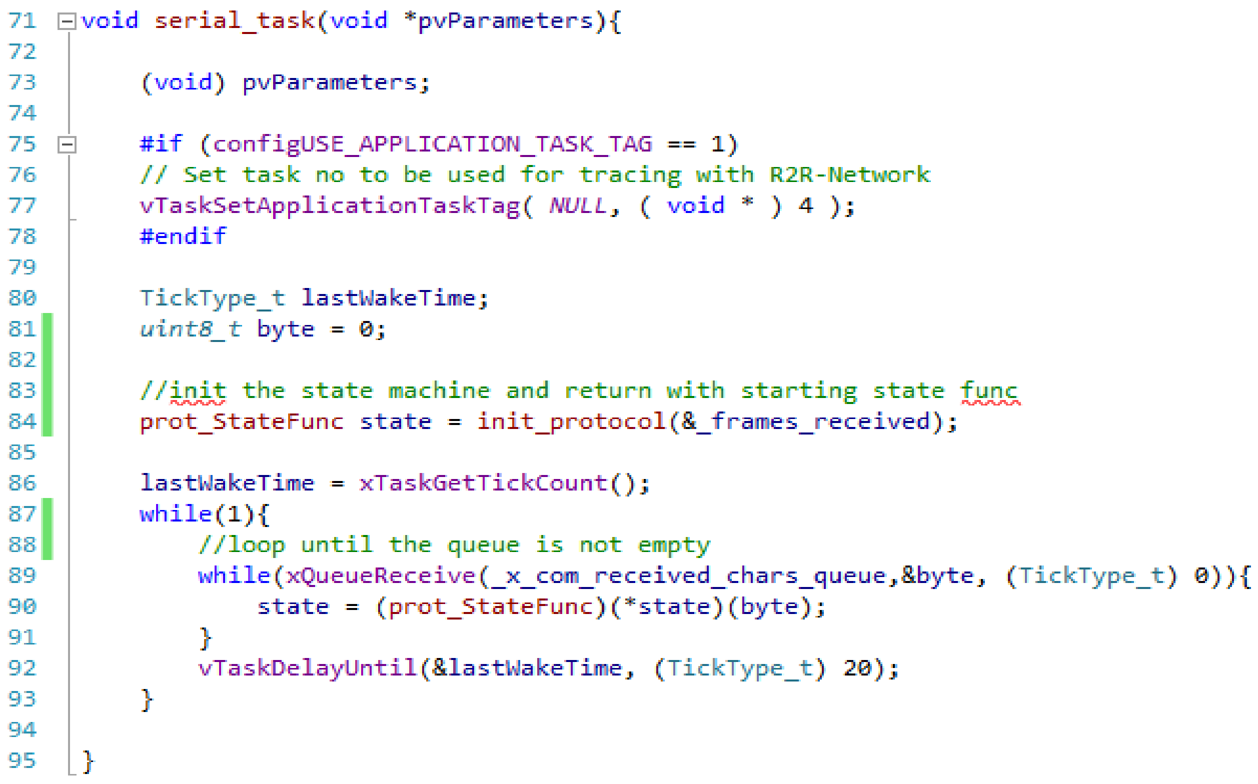
On the game console side, the serial driver was already implemented in the given FreeRTOS port, but the display driver only had the hardware timer setup.



```c
//-----------------------------------------
void handle_display(void)
{
    //set SER to HIGHT
    if (col_index == 0){
        PORTD |= _BV(PORTD2);
    }

    // one SCK pulse
    PORTD |= _BV(PORTD5);
    PORTD &= ~_BV(PORTD5);

    // Set SER to 0
    PORTD &= ~_BV(PORTD2);

    // one RCK pulse
    PORTD |= _BV(PORTD4);
    PORTD &= ~_BV(PORTD4);

    //Pixel0..07
    PORTA = ~(col_value[col_index] & 0xFF);
    //Pixel8..9
    // Manipulate only with PB0 and PB1
    PORTB |= 0x03;
    PORTB &= ~((col_value[col_index] >> 8) & 0x03);
    //jump to next column
    ++col_index;
    if (col_index > 13){
        col_index = 0;
    }

}
```

Serial connetion task

The hardware timer fires the handle_display() function every time it expires. First the shift registers' serial data is clocked to shift the data, then the output is clocked to display the shifted that. Lastly the LEDs in the column need to be lighted up according the columns value. As there are 10 pixels in each column and the ports of the MCU are eight bits, two ports are need. The values of the columns are 16 bits, the PORTA takes bit0 to bit7 and PORTB0 and PORTB1 take bit8 and bit9, as the rest of PORTB is used by other components. These bits then need to complemented as the LEDs' cathodes are connected to the pins, thus the current is sunk into the MCU. This mean the pins have to be low to light up the LEDs.



```c
void serial_task(void *pvParameters){

    (void) pvParameters;

    #if (configUSE_APPLICATION_TASK_TAG == 1)
    // Set task no to be used for tracing with R2R-Network
    vTaskSetApplicationTaskTag( NULL, ( void * ) 4 );
    #endif

    TickType_t lastWakeTime;
    uint8_t byte = 0;

    //init the state machine and return with starting state func
    prot_StateFunc state = init_protocol(&_frames_received);

    lastWakeTime = xTaskGetTickCount();
    while(1){
        //loop until the queue is not empty
        while(xQueueReceive(_x_com_received_chars_queue,&byte, (TickType_t) 0)){
            state = (prot_StateFunc)(*state)(byte);
        }
        vTaskDelayUntil(&lastWakeTime, (TickType_t) 20);
    }

}
```

Game state machine code

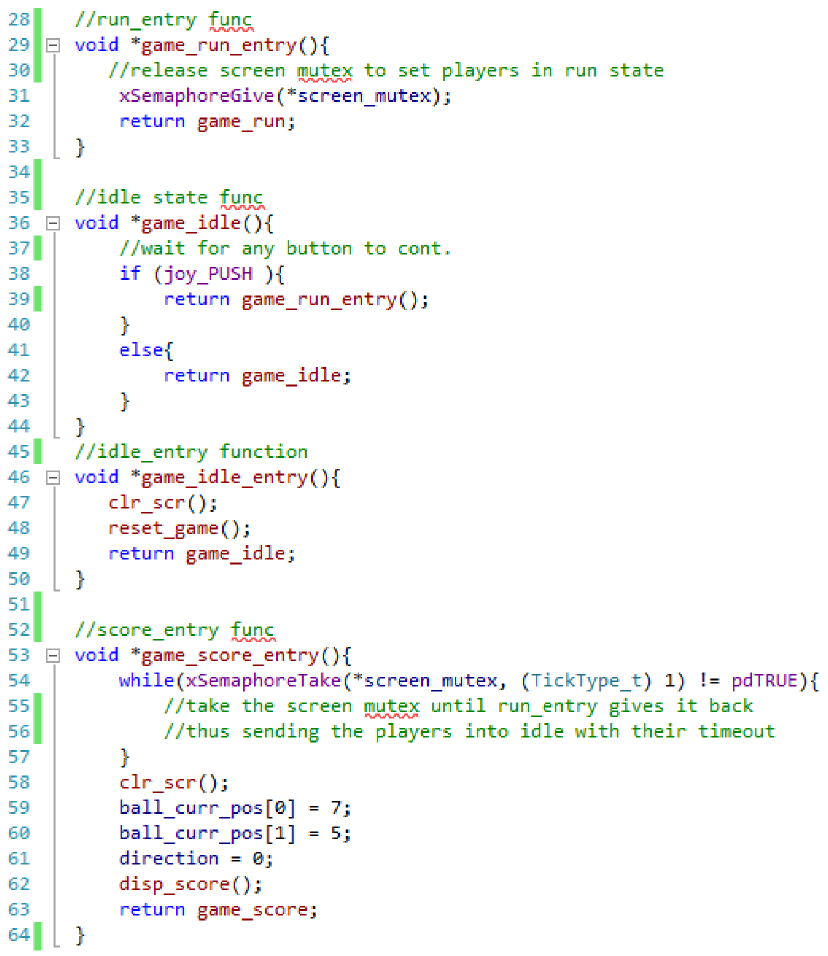
The serial task and game task implement their designed state machines with function pointers. Their FreeRTOS tasks initialize the state machine and variables, then they enter where they call the state function then delay themselves using vTaskDelayUntil(). The vTaskDelayUntil() was chosen over the simpler vTaskDelay(), because the vTaskDelay can't guarantee a fixed cycle.

The state machine themselves are implemented in a way that only the state functions and the initialization can be accessed from outside. In case a state has an entry or exit function, these are implemented as a function, that after execution are returning with the desired state function's pointer.

The two task for the players are simple, two-state state machine. They are mostly identical, they differ only in inputs and accessed resources.

All of the shared resources between the task are protected either with a mutex or with a queue. In every cases these are taken only for the least amount of time. There is only one case when they are accessed unprotected. The display driver reads the screen buffer without taking the mutex. This can result in displaying a wrong value, but as wrong value is the previous value and the refresh rate is ca. 93Hz, no artifact is produces only a hardly noticeable lag. If it was hardly protected by a semaphore the lag could be noticeable.

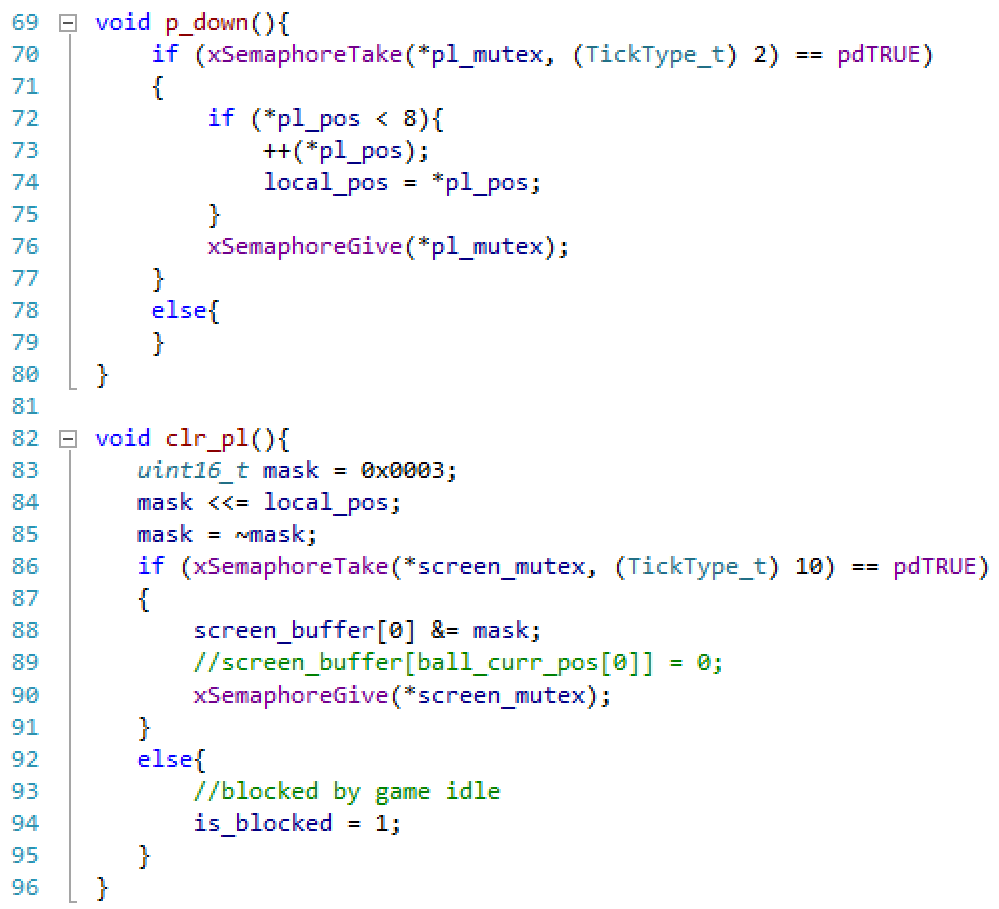
The mutex protecting the screen buffer is also used for signaling. When the game enters score state it takes the mutex and not release it until it enters to run state again. This forces the player tasks to timeout on their screen update and send them to idle.



```c
//run_entry func
void *game_run_entry(){
   //release screen mutex to set players in run state
    xSemaphoreGive(*screen_mutex);
    return game_run;
}

//idle state func
void *game_idle(){
    //wait for any button to cont.
    if (joy_PUSH ){
        return game_run_entry();
    }
    else{
        return game_idle;
    }
}
//idle_entry function
void *game_idle_entry(){
   clr_scr();
   reset_game();
   return game_idle;
}

//score_entry func
void *game_score_entry(){
    while(xSemaphoreTake(*screen_mutex, (TickType_t) 1) != pdTRUE){
        //take the screen mutex until run_entry gives it back
        //thus sending the players into idle with their timeout
    }
    clr_scr();
    ball_curr_pos[0] = 7;
    ball_curr_pos[1] = 5;
    direction = 0;
    disp_score();
    return game_score;
}
```

Player task code

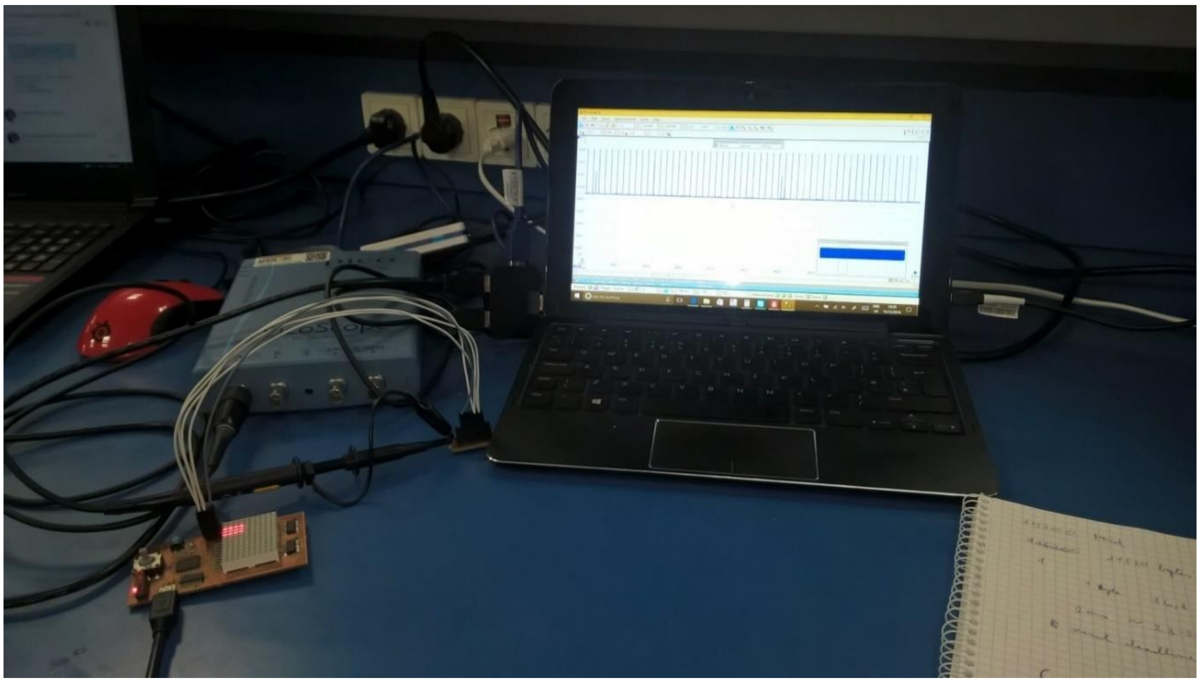
In some case the tasks also use a local copy of the protected resource. This is used for calculations and predictions if the resource isn't available. For example the game task using the local copy for the player positions and if the shared data isn't available, it uses the local to predict the movement of the ball. This local copies are updated as soon as the task get access to the shared one.



```
void p_down(){
    if (xSemaphoreTake(*pl_mutex, (TickType_t) 2) == pdTRUE)
    {
        if (*pl_pos < 8){
            ++(*pl_pos);
            local_pos = *pl_pos;
        }
        xSemaphoreGive(*pl_mutex);
    }
    else{
    }
}

void clr_pl(){
   uint16_t mask = 0x0003;
   mask <<= local_pos;
   mask = ~mask;
   if (xSemaphoreTake(*screen_mutex, (TickType_t) 10) == pdTRUE)
   {
       screen_buffer[0] &= mask;
       //screen_buffer[ball_curr_pos[0]] = 0;
       xSemaphoreGive(*screen_mutex);
   }
   else{
       //blocked by game idle
       is_blocked = 1;
   }
}
```

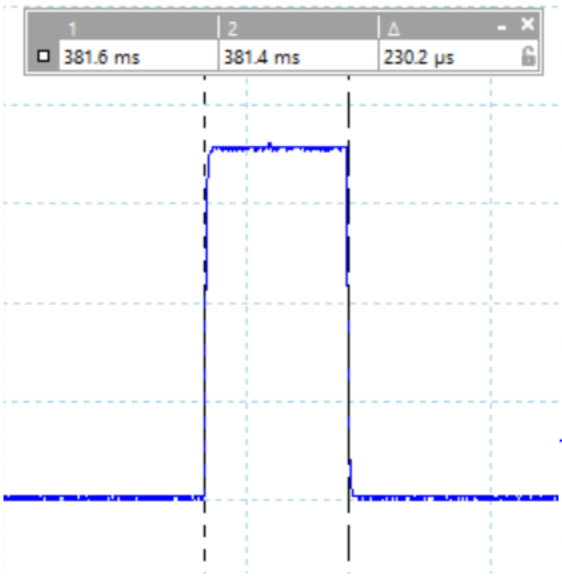
Test

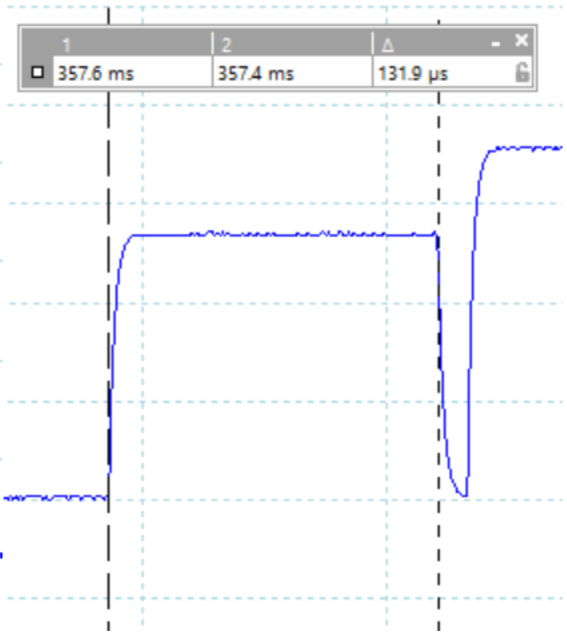
To test the system and to see if the deadlines are met, the computation time for every task had to be measured. This is possible as there are 5 spare IO pins on the board and the FreeRTOS port already have a driver implemented for an ADC. After all tasks were given a unique value and the monitoring enabled, a 4-bit digital-to-analog converter was connected to the spare IO and an oscilloscope was connected to the analog output.

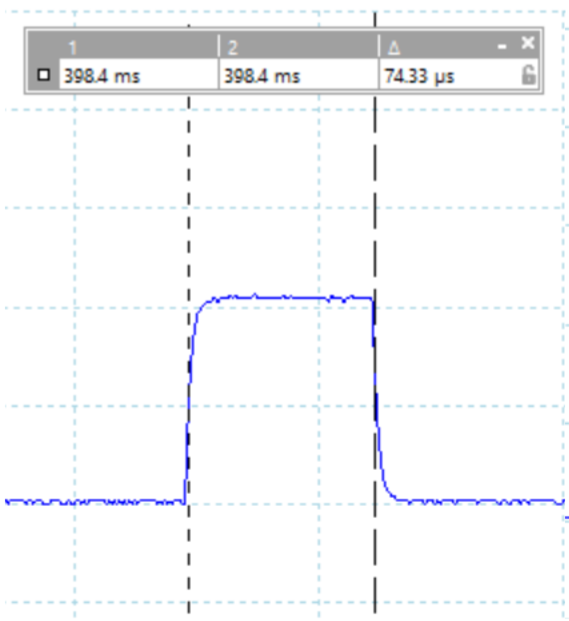


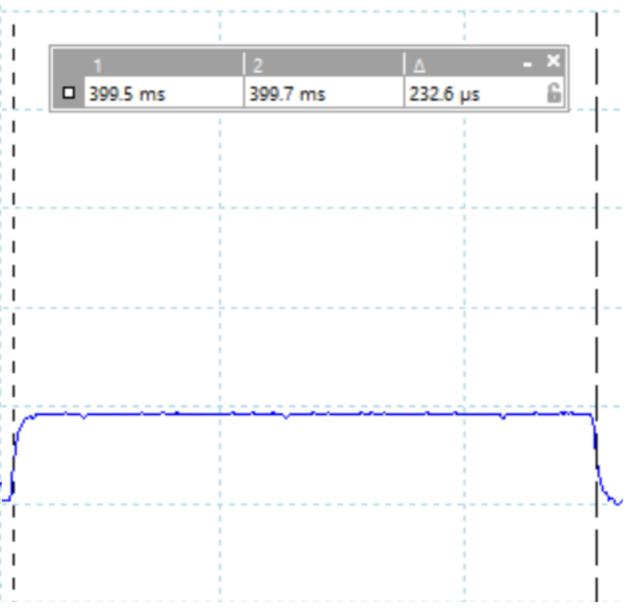
Using the oscilloscope the computation time of each task was measured. Multiple measurements were recorded and for each the largest value was taken as the computation time.

Measurements of computation time.









The largest measured values for each task were:

• Serial task	253µs
• Game task	147µs
• Local player task	84µs
• Serial player task	255µs

Calculation of utilization

These values can be used to calculate the utilization of the total available computation time.

$$U_{task} = \frac{computation\ time}{period}$$

$$U_{serial} = \frac{253\mu s}{2000\mu s} = 0.1265$$

$$U_{game} = \frac{147\mu s}{1.2 * 10^5 \mu s} = 0.0012$$

$$U_{local\ player} = \frac{84\mu s}{6 * 10^4 \mu s} = 0.0014$$

$$U_{serial\ player} = \frac{255\mu s}{6 * 10^4 \mu s} = 0.0043$$

$$U_{total} = U_{serial} + U_{game} + U_{local\ player} + U_{serial\ player}$$

$$U_{total} = 0.1265 + 0.0012 + 0.0014 + 0.0043 = 0.1334 \approx 13.4\%$$

The 13.4% total utilization is under the threshold of 75.7% limit for 4 tasks using utilization based analysis. This proves that the tasks can be scheduled, however it doesn't prove that all task will respond under their deadline. To prove that response time analysis can be used as FreeRTOS uses a fixed priority based scheduling.

Calculations of response time

	Period(T, µs)	Computation time (C, µs)	Priority (bigger is higher)
Serial task	253µs	2000µs	4
Game task	147µs	$1.2 * 10^5$µs	3
Local player task	84µs	$6 * 10^4$µs	2
Serial player task	255µs	$6 * 10^4$µs	1

$$R = response\ time$$

$$R_{serial} = C_{serial} = 253\text{µs}$$

The worst case response time of 253µs is under the period of 2000µs, so the Serial task hits its deadline every time.

$$w_{game}^{0} = C_{game} = 147\text{µs}$$

$$w_{game}^{1} = C_{game} + \left\lceil \frac{w_{game}^{0}}{T_{serial}} \right\rceil * C_{serial} = 147\text{µs} + \left\lceil \frac{147\text{µs}}{2000\text{µs}} \right\rceil * 253\text{µs} = 147\text{µs} + 1 * 253\text{µs} = \ 400\text{µs}$$

$$w_{game}^{2} = C_{game} + \left\lceil \frac{w_{game}^{1}}{T_{serial}} \right\rceil * C_{serial} = 147\text{µs} + \left\lceil \frac{400\text{µs}}{2000\text{µs}} \right\rceil * 253\text{µs} = 147\text{µs} + 1 * 253\text{µs} = \ 400\text{µs}$$

$$as\ w_{game}^{1} = w_{game}^{2} \rightarrow R_{game} = w_{game}^{1} = 400\text{µs}$$

The worst case response time of 400µs is under the period of $1.2 * 10^5$µs, so the Game task hits its deadline every time.

$$w_{local\ player}^{0} = C_{local\ player} = 84\text{µs}$$

$$w_{local\ player}^{1} = C_{local\ player} + \left\lceil \frac{w_{local\ player}^{0}}{T_{serial}} \right\rceil * C_{serial} + \left\lceil \frac{w_{local\ player}^{0}}{T_{game}} \right\rceil * C_{game} =$$

$$= 84\text{µs} + \left\lceil \frac{84\text{µs}}{2000\text{µs}} \right\rceil * 253\text{µs} + \left\lceil \frac{84\text{µs}}{1.2 * 10^5\text{µs}} \right\rceil * 147\text{µs} = 84\text{µs} + 1 * 253\text{µs}\ + 1 * 147 == \ 484\text{µs}$$

$$w_{local\ player}^{2} = 84\text{µs} + \left\lceil \frac{484\text{µs}}{2000\text{µs}} \right\rceil * 253\text{µs} + \left\lceil \frac{484\text{µs}}{1.2 * 10^5\text{µs}} \right\rceil * 147\text{µs} = 84\text{µs} + 1 * 253\text{µs}\ + 1 * 147 == \ 484\text{µs}$$

$$as\ w_{local\ player}^{1} = w_{local\ player}^{2} \rightarrow R_{local\ player} = w_{local\ player}^{1} = 484\text{µs}$$

The worst case response time of 484µs is under the period of $6 * 10^4$µs, so the Local Player task hits its deadline every time.

$$w^0_{serial\ player} = C_{serial\ player} = 84\mu s$$

$$w^1_{serial\ player} =$$

$$= C_{serial\ player} + \left\lceil \frac{w^0_{serial\ player}}{T_{serial}} \right\rceil * C_{serial} + \left\lceil \frac{w^0_{serial\ player}}{T_{game}} \right\rceil * C_{game} + \left\lceil \frac{w^0_{serial\ player}}{T_{local\ player}} \right\rceil * C_{local\ player} =$$

$$= 255\mu s + \left\lceil \frac{255\mu s}{2000\mu s} \right\rceil * 253\mu s + \left\lceil \frac{255\mu s}{1.2 * 10^5\mu s} \right\rceil * 147\mu s + \left\lceil \frac{255\mu s}{6 * 10^4\mu s} \right\rceil * 84\mu s =$$

$$= 255\mu s + 1 * 253\mu s \ + 1 * 147 + 1 * 84 = \ 739\mu s$$

$$w^2_{serial\ player} = 255\mu s + \left\lceil \frac{739\mu s}{2000\mu s} \right\rceil * 253\mu s + \left\lceil \frac{739\mu s}{1.2 * 10^5\mu s} \right\rceil * 147\mu s + \left\lceil \frac{739\mu s}{6 * 10^4\mu s} \right\rceil * 84\mu s =$$

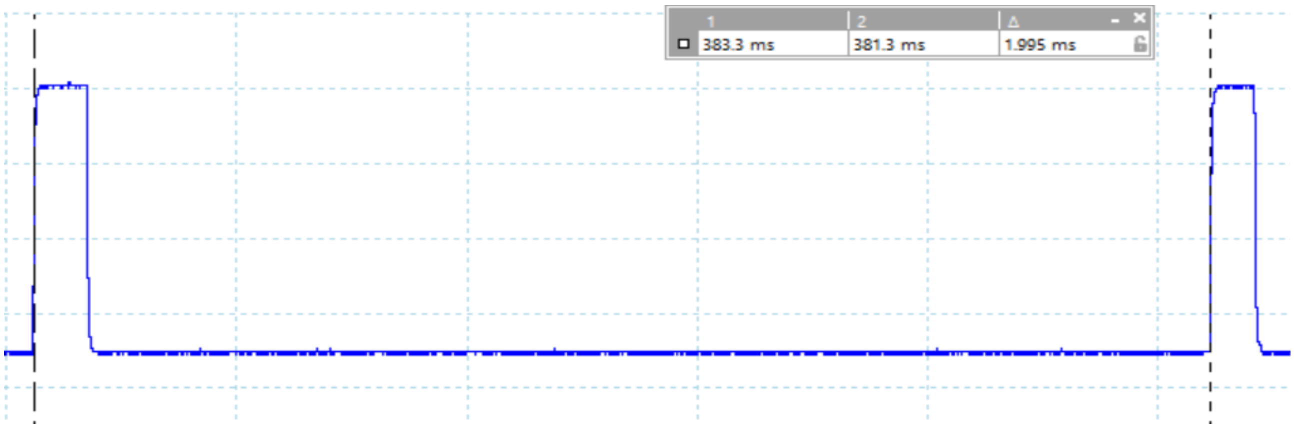
$$= 255\mu s + 1 * 253\mu s \ + 1 * 147 + 1 * 84 = \ 739\mu s$$

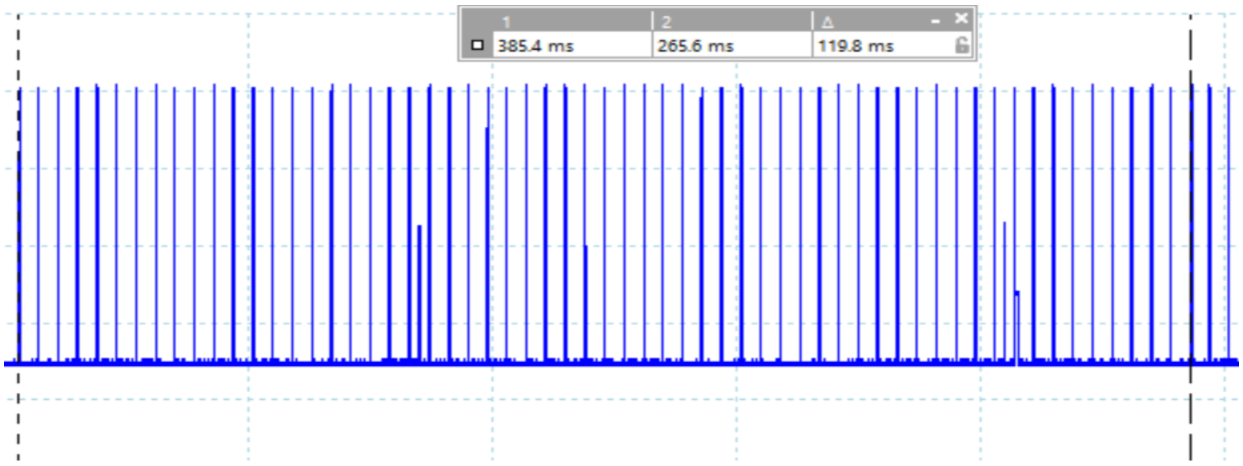
$$as\ w^1_{serial\ player} = w^2_{serial\ player} \rightarrow R_{serialplayer} = w^1_{serialplayer} = 739\mu s$$

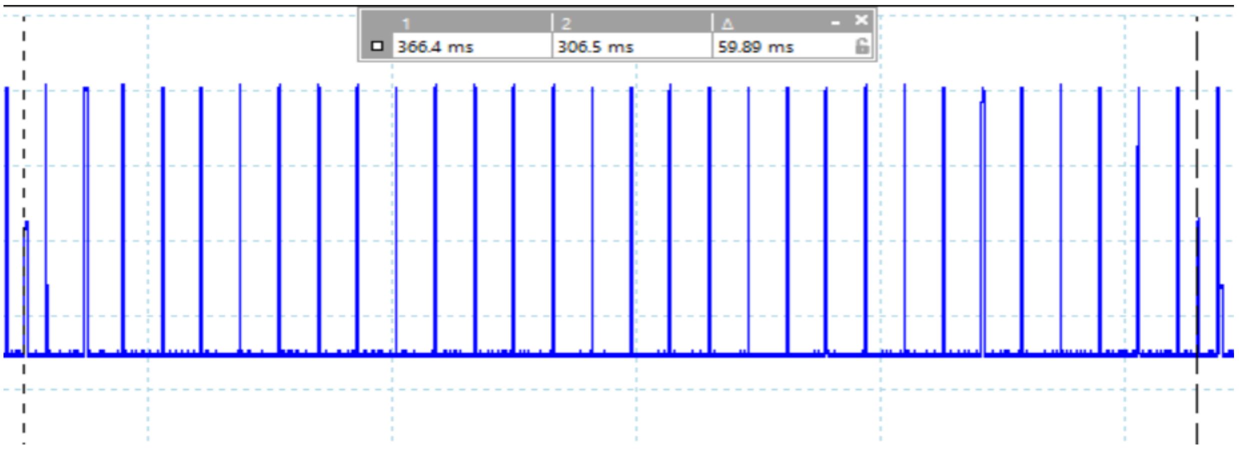
The worst case response time of 739µs is under the period of $6 * 10^4$µs, so the Local Player task hits its deadline every time.

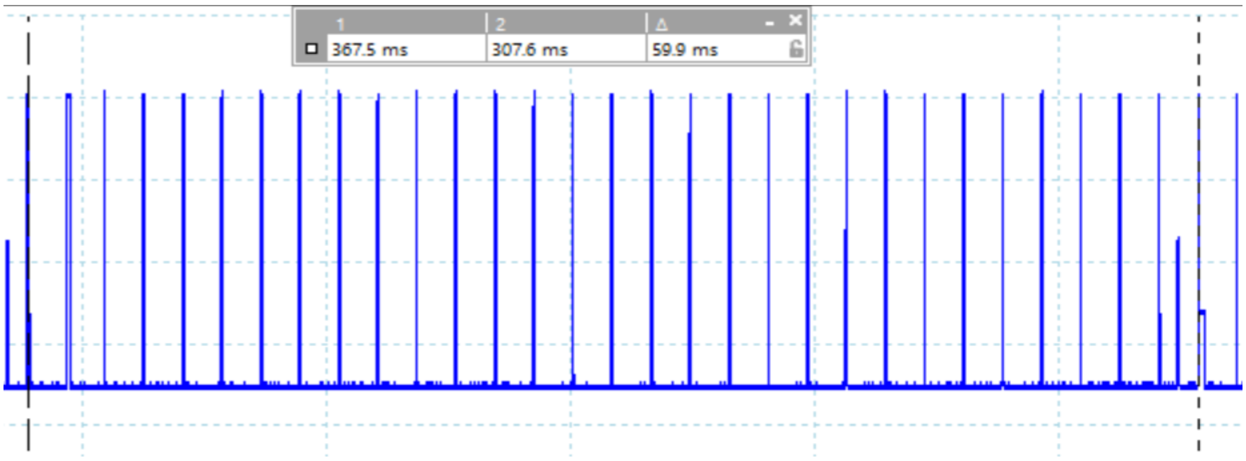
Measurement of Periods

The calculations show that all of the tasks finish their work before reaching the end of their periods. It can be argued, that the both the utilization and the response time analysis aren't correct as they are omitting the computation times needed by the serial and the display driver and overhead from FreeRTOS. This is true, however it can be seen from the oscilloscope measurements below that these processes aren`t an issues for the deadlines.









Result

In the result part, we have concluded from our testing of the Ping pong game. What functions are working and what functions are not working. Every required and necessary function are listed below as working or not working.

Working:

- All Four tasks
- Shared resources (Display, serial connection)
- semaphores/mutexes
- input from joystick (Local player moves)
- input from Computer (Serial player moves)
- Game state machine (ball moves and resets)
- Display working as interrupt
- Protocol working

Not working:

- CRC not implemented
- Point system not completed (displays points but does not reset)
- Use of two boards as players (not implemented)

Discussion and Conclusion

To summarize the project, the purpose of this project were to be familiarized with real time programming and cross platform development. At the beginning there were some discrepancies about how to start this project and since some of the knowledge needed for the project were not taught during the semester. The project then had to start out by learning new theory and how to connect to the real-time systems. After the first week, the project was going good there were connection to the board and the Micro Controller were working. The project was then well under way and begun to be structured for better understanding, computation and debugging purposes.

At the end, all the requirements were met, however there is place for some improvements. The protocol can be extended to use CRC for error detection and sending side buffering so multiple messages can be sent at once. The game can also be improved to bounced the ball when the player moves into the ball.

To conclude the program ended up working as expected and with minimal amount of errors the project even ended with the cross-platform player working. the project ended up being educational and with motivation to learn more. The experience gained from this project will be useful in the future.

References

FreeRTOS. (u.d.). from http://www.freertos.org

labrosse, j. j. (2002). *Micro c/ os 2 the real time kernel.* CMPBooks.

monk, P. s. (2006). *practical electronics for inventors.* McGraw Hill Professional.

Wellings, A. B. (2014). *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX.* addison wesley.

Appendices

Appendix A - Use Case, Task activity diagrams

Appendix B - protocol diagrams.

Appendix C - Task diagram.

Appendix D - first part of project description

Appendix E - second part of project description.

Appendix F – ComputerControlUnit code

Appendix G – MicroControlUnit code