2016

Jimmi Andersen, Marek Mikitovic and David Papp

Semester Project fourth Semester

16-12-2016

Project Report For Game-Console

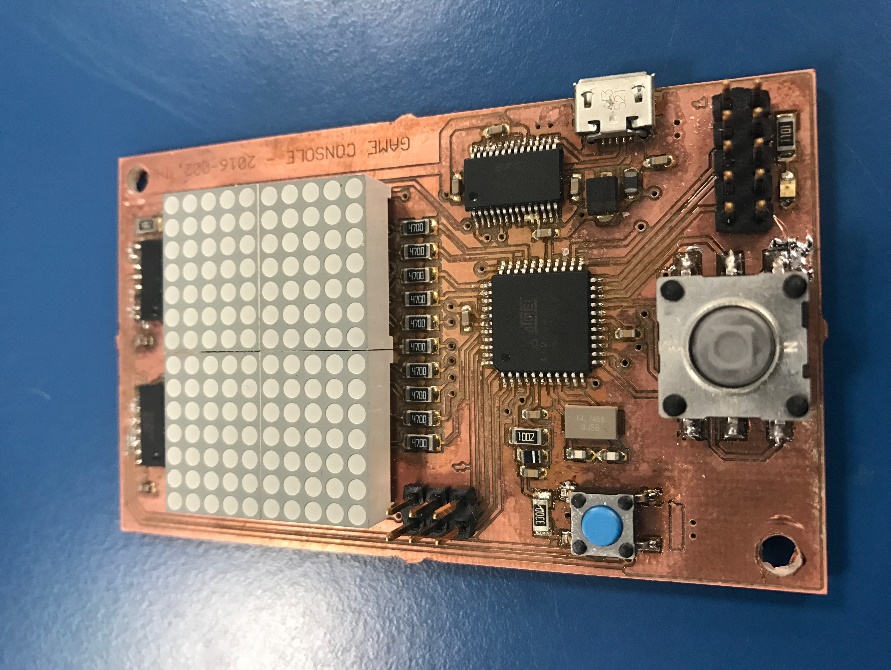


Table of Content.

[Abstract. 2](#_Toc469569433)

[Introduction. 2](#_Toc469569434)

[Analysis. 3](#_Toc469569435)

[Requirements. 3](#_Toc469569436)

[Functional: 3](#_Toc469569437)

[Non-functional: 3](#_Toc469569438)

[Use-Case diagram. 4](#_Toc469569439)

[Task Diagram. 5](#_Toc469569440)

[Design. 6](#_Toc469569441)

[Game Protocol 6](#_Toc469569442)

[Task 1 Serial Connection 7](#_Toc469569443)

[Task 1 Serial Connection Diagram 7](#_Toc469569444)

[Task 2 Game 8](#_Toc469569445)

[Task 2 State Machine 8](#_Toc469569446)

[Idle State Diagram. 9](#_Toc469569447)

[Task 3 Player on board 11](#_Toc469569448)

[Task 3 Player on board Diagram 11](#_Toc469569449)

[Task 4 Player Pc 12](#_Toc469569450)

[Task 4 Player Pc Diagram 12](#_Toc469569451)

[Implementation. 13](#_Toc469569452)

[Test. 18](#_Toc469569453)

[System timeline. 18](#_Toc469569454)

[Result. 18](#_Toc469569455)

[Working: 18](#_Toc469569456)

[Not working: 18](#_Toc469569457)

[Discussion. 18](#_Toc469569458)

[Conclusion. 18](#_Toc469569459)

[References 19](#_Toc469569460)

# 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.*

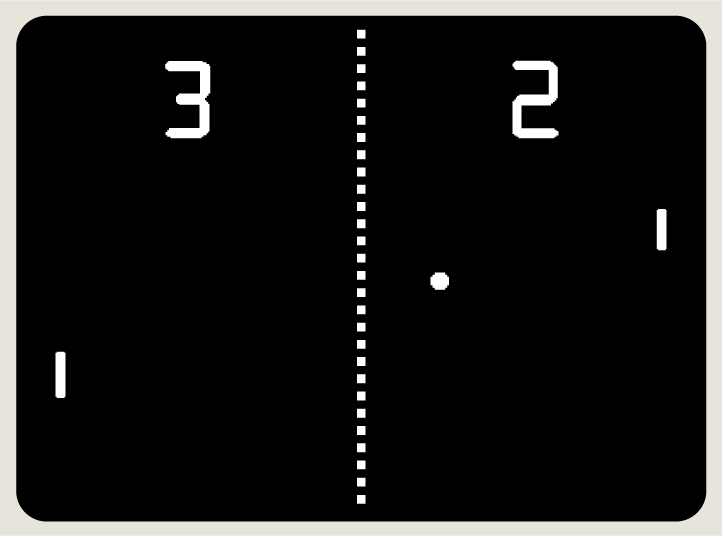
# Introduction.

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(MCU) and player two is controlled by the keyboard on a computer using the USB 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:

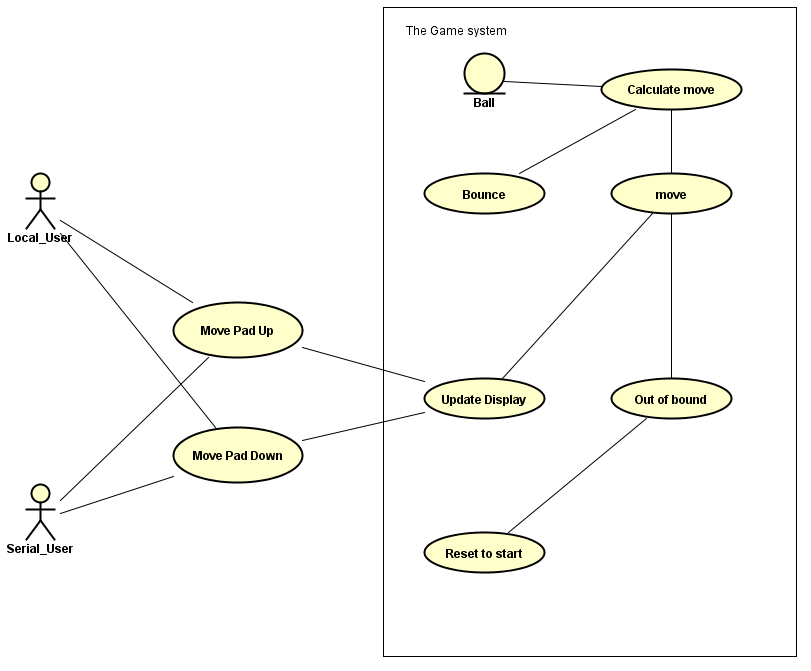
* System must use 3 tasks.
* System must have 2 real time tasks.
* System must have 2 task sharing resources.
* The system must use semaphores or mutexes.
* The pc must be able to control player two on game board.
* The system must give real time guarantee.

### Non-functional:

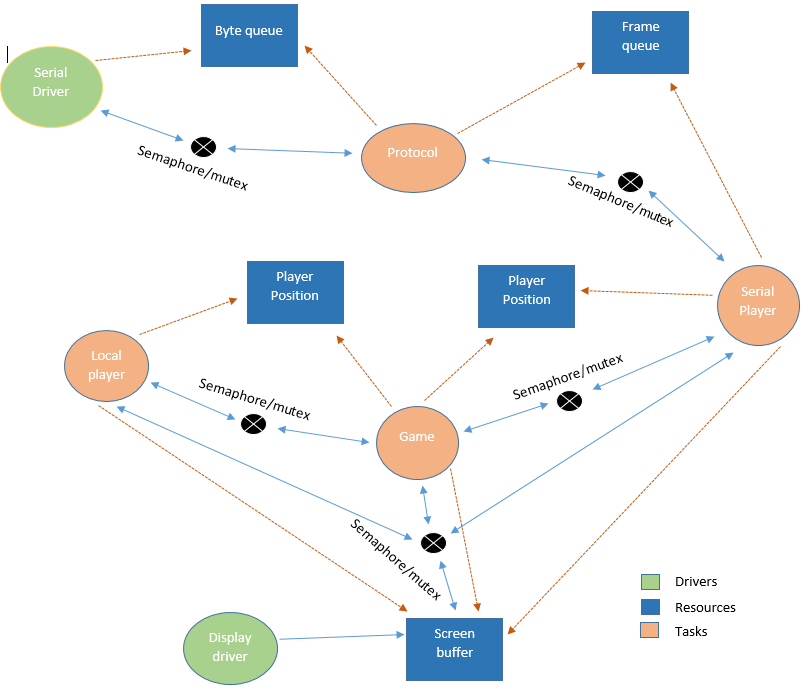
* Game must display the game on computer screen.
* Game gives score when ball hits paddle
* Game resets when ball is out of bounds

### 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.

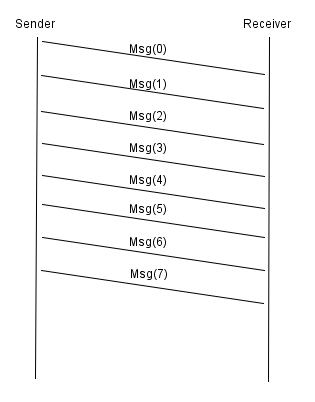
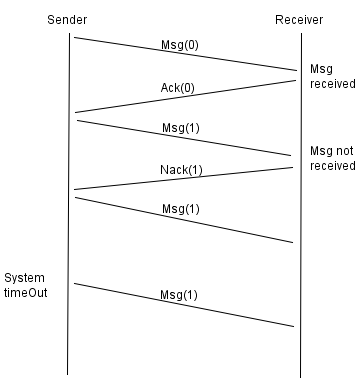


# Design.

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

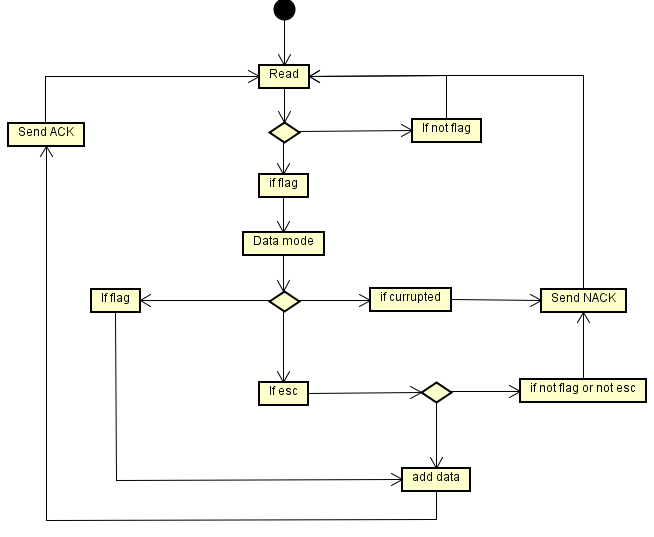
 Diagram 1 Secure system protocol Diagram 2 Simple Protocol 1

## 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

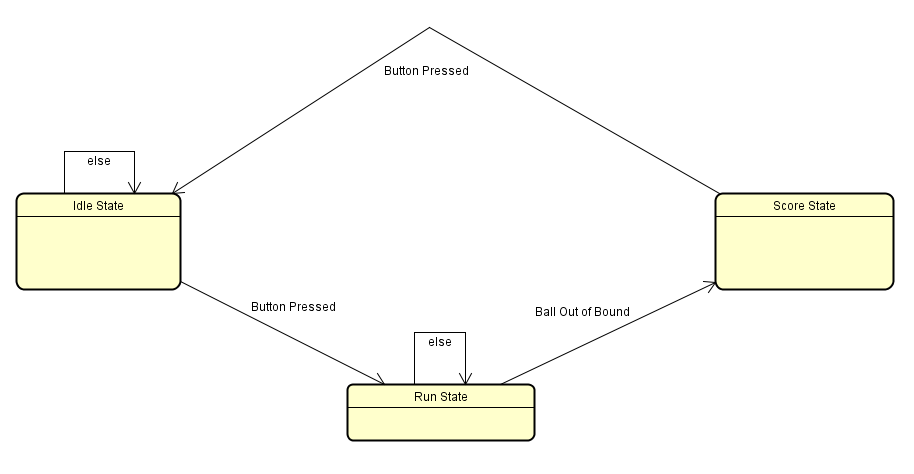


## Task 2 Game

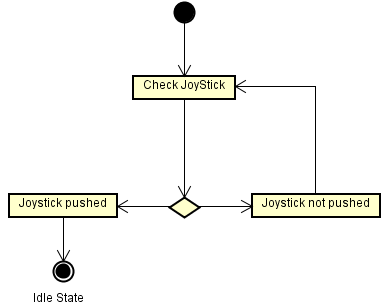
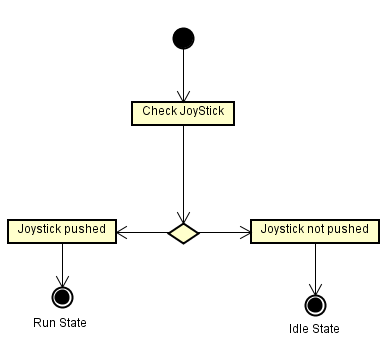
### The Game task is implemented as the state machine. The task will be in idle state until the joystick is pressed and then it will go to the run state, where the ball and game are updated. If the ball gets out of the frame it will go to score state and add points to the player. In the score state,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

### Task 2 State Machine

### 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

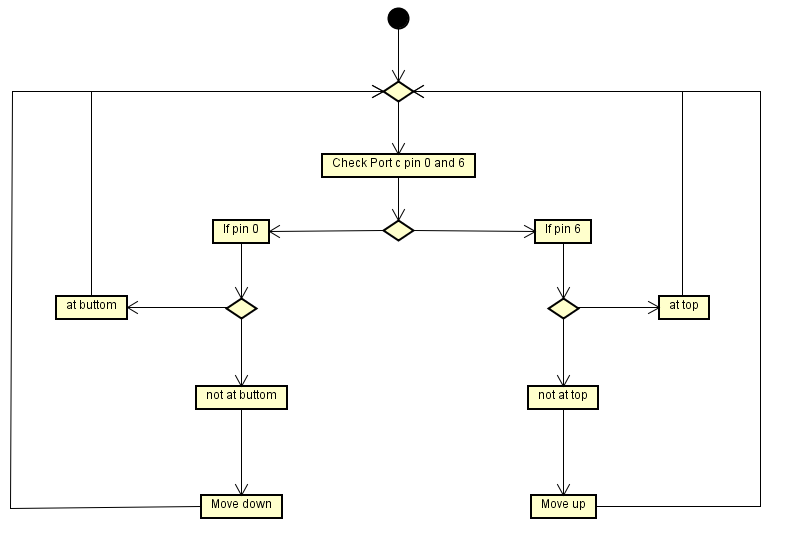
### The run state is displayed below. It is more complex then idle and score state. When the run state starts, it sets the bounce to 1 to get into the loop. Then it calls the calculate next position function which gives the (x, y) coordinates for the ball to travel to. Then it checks the x coordinate 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.

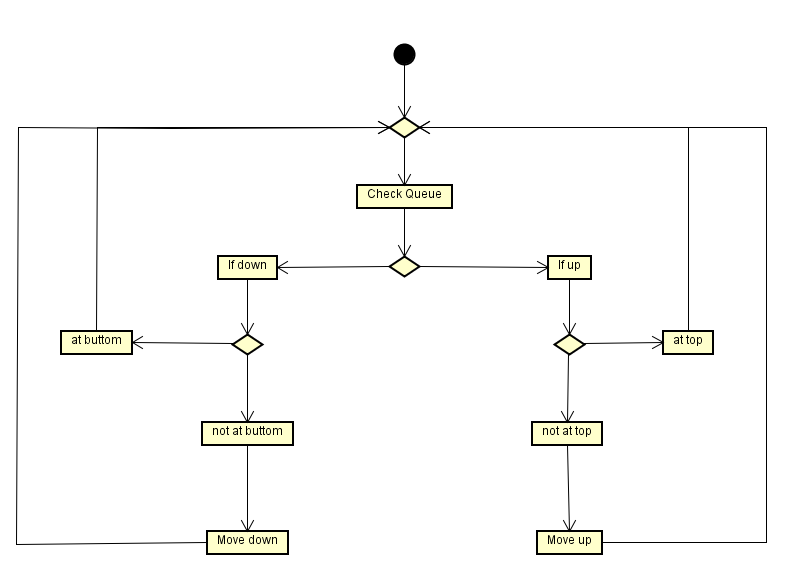
### Task 3 Player on board Diagram



## 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:



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

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

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:

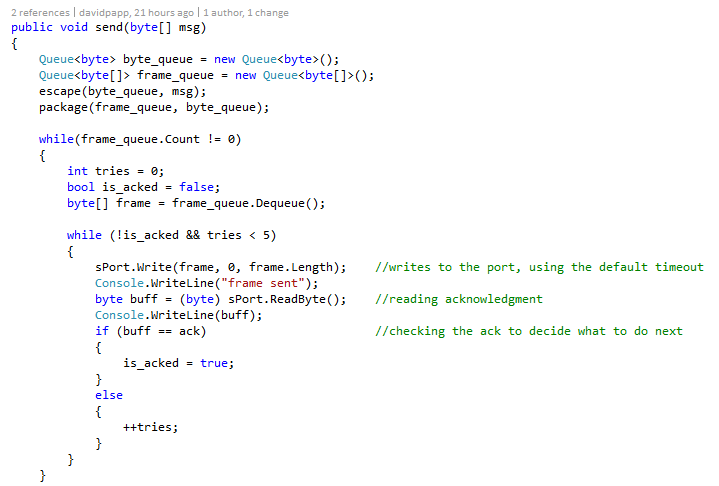
1. Serial task Priority: 4 Period: 2ms
2. Game task Priority: 3 Period: to be determined
3. Local player task Priority: 2 Period: to be determined
4. Serial player task Priority: 1 Period: to be determined

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.

# Implementation.

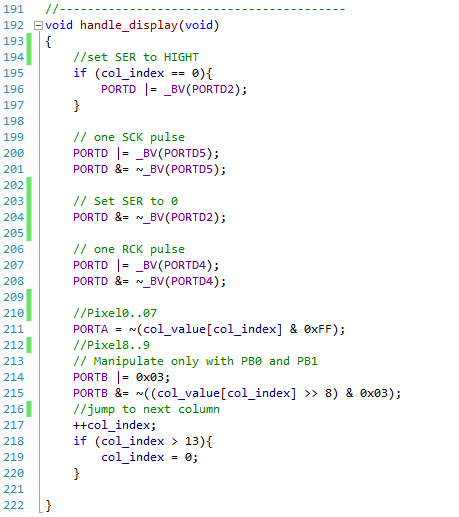
The implementation was done in C utilizing the given FreeRTOS port for the game console part and C# with .NET libraries for the PC part.

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.

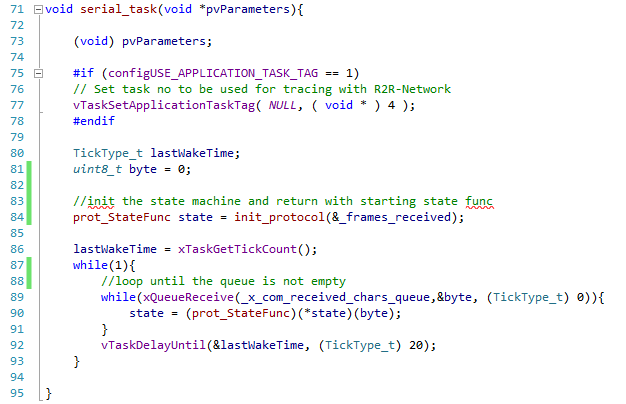


Here it can be seen how was the send method implemented. As it can be seen, every frame is tried maximum 5 times, after unsuccessfully try, the frame will be thrown away and the next one will be tried. This implementation is really simple, because both the Write() and ReadByte() method in System.IO.Ports.SerialPort 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.



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.



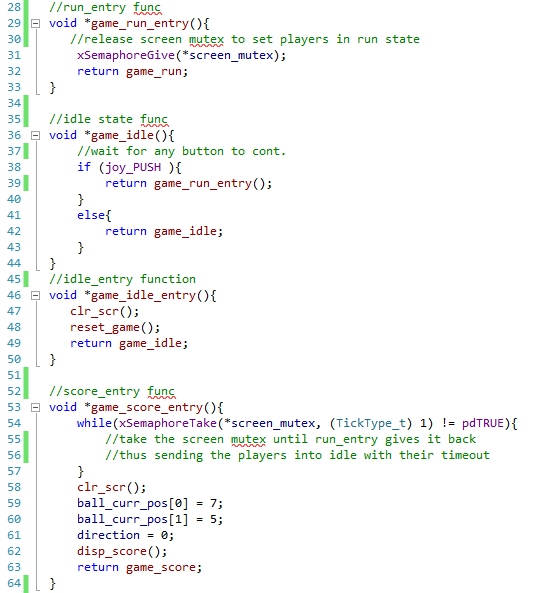
The serial task and game task implement their designed state machines with function pointers. Their FreeRTOS tasks initialize the state machine and variables, then they enter where they call the state function then delay themselves using vTaskDelayUntil(). The vTaskDelayUntil() was chosen over the simpler 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.

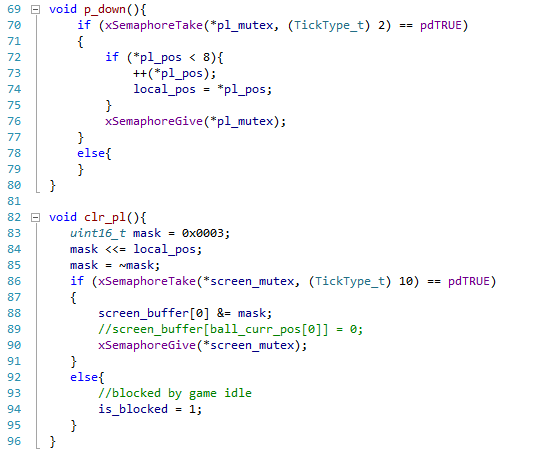
The two task for the players are simple, two-state state machine. They are mostly identical, they differ only in inputs and accessed resources.

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.



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.



# Test.

Picoscope and calculations here

## System timeline.

# 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.

In the development of this game we have put our minds to the test we started out as a two-man group but later got to be tree in the group. We were supposed to start the project by making diagrams of the system but because no one were sure about how those diagrams were supposed to look like we ended up starting to program the board instead. Our approach was to start getting the display to display the game and then work from there. This we meant were the best approach for our group we ended up making the game work as simple as possible in the beginning we then decided to split the code up and make it work efficiently and to make it easier to debug.

# Conclusion.

# References

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.