2016

Jimmi Andersen & David Papp

Semester Project fourth Semester

16-12-2016

Project Report For Game-Console

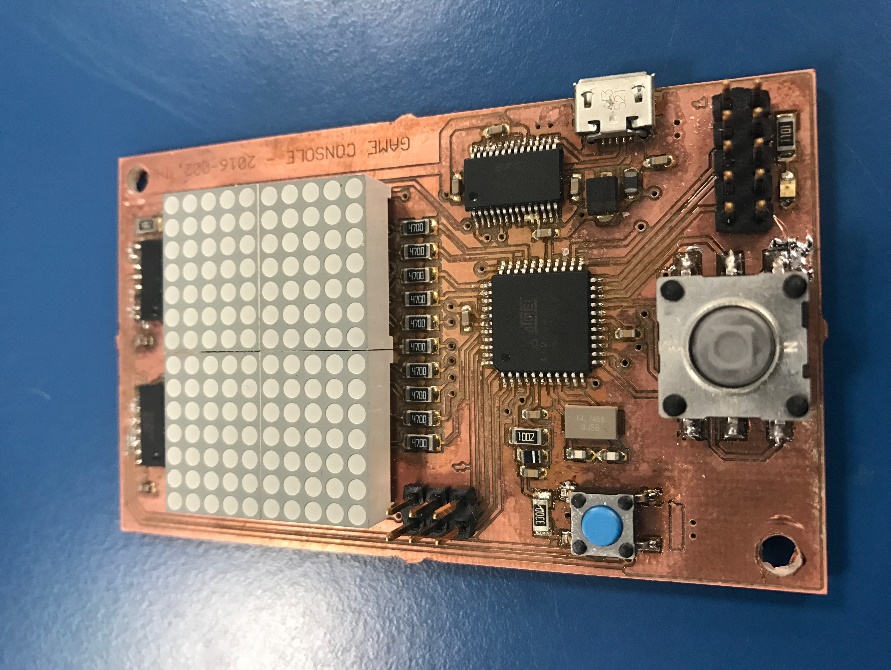


Table of Content.

[Abstract. 2](#_Toc469044438)

[Requirements. 3](#_Toc469044439)

[Analysis. 3](#_Toc469044440)

[Design. 3](#_Toc469044441)

[Task 1 Serial Connection 3](#_Toc469044442)

[Task 1 Serial Connection Diagram 3](#_Toc469044443)

[Task 2 Ball 3](#_Toc469044444)

[Task 2 Ball Diagram 3](#_Toc469044445)

[Task 3 Player on board 3](#_Toc469044446)

[Task 3 Player on board Diagram 3](#_Toc469044447)

[Task 4 Player Pc 3](#_Toc469044448)

[Task 4 Player Pc Diagram 3](#_Toc469044449)

[Test. 3](#_Toc469044450)

[Implementation. 3](#_Toc469044451)

# Abstract.

Putty speed = 115200

# Requirements.

The projects requirements are we must implement a two-player game system using the Game-console board we created in the real-time programming course to do this our system must have at least 3 tasks where two of them must be hard real time tasks we need to have data that are shared between more than one task. The system must use the functionalities of the board like the joystick, Dot-matrix display, and the serial connection to test our board and to make Shure that our system is schedulable we must use R-2R D/A converter to measure the time of the tasks.

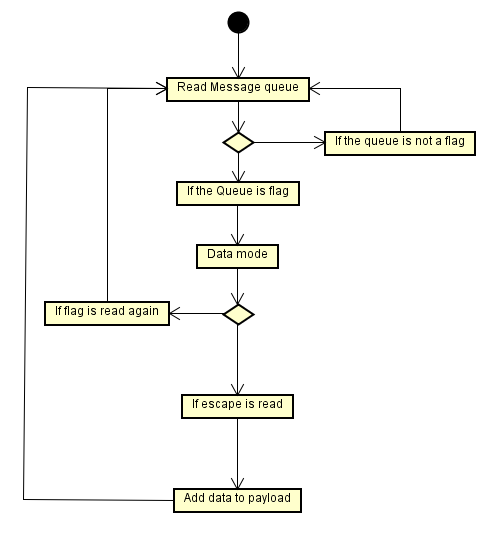
Testing our system, we will be using unit testing for testing the software. We will be using an oscilloscope to find the computation time of the tasks. Our system must use the computer as player one and the game board as player two the screen will be the Dot matrix on the game board.

# Analysis.

# Design.

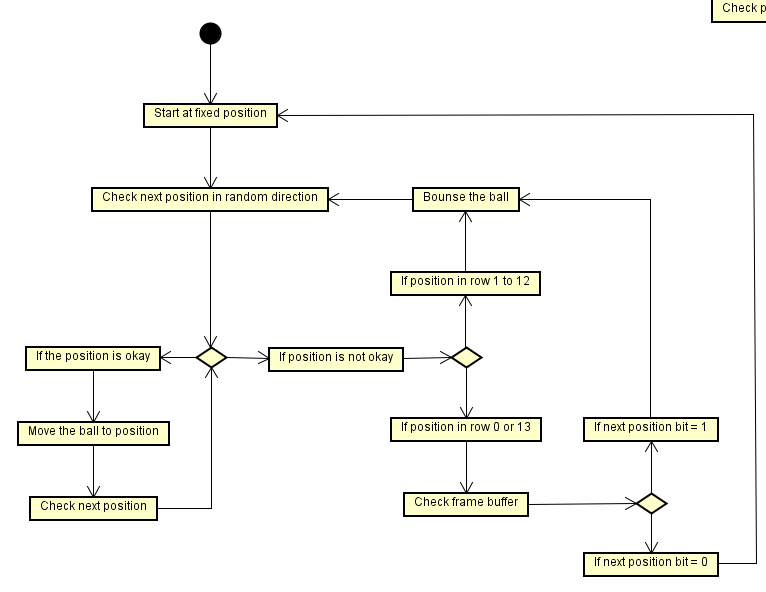
## Task 1 Serial Connection

### Task 1 Serial Connection Diagram



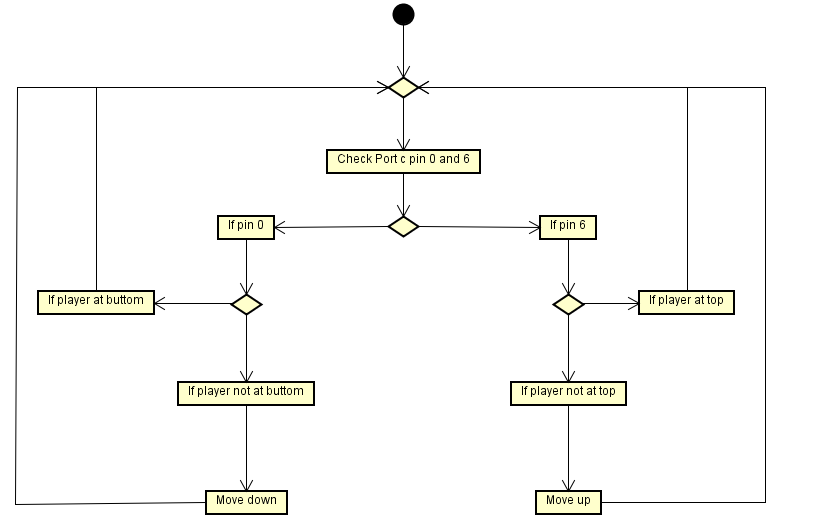
## Task 2 Ball

### Task 2 Ball Diagram



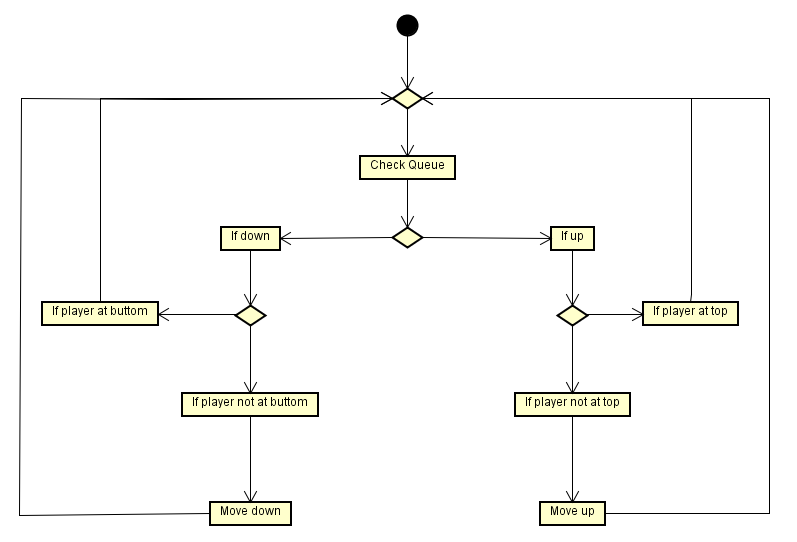
## Task 3 Player on board

### Task 3 Player on board Diagram



## Task 4 Player Pc

### Task 4 Player Pc Diagram

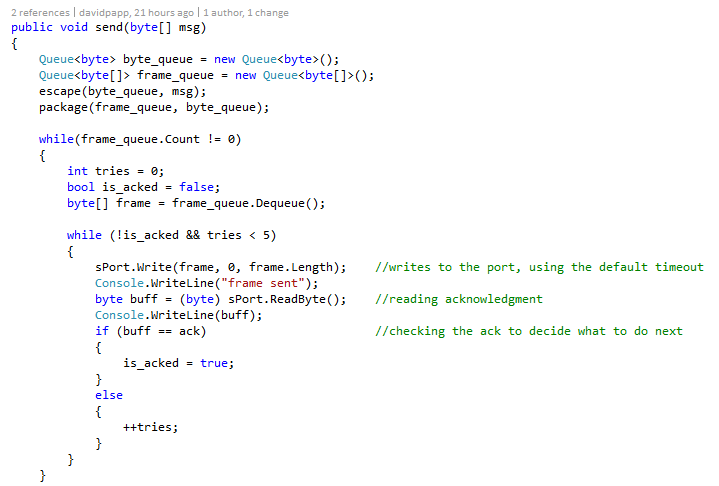


# Test.

# Implementation.

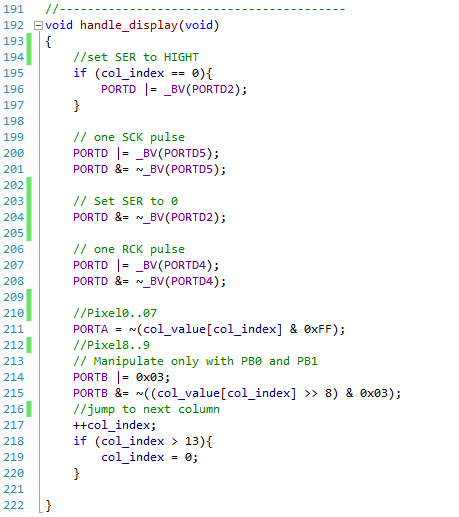
The implementation was done in C utilizing the given FreeRTOS port for the game console part and C# with .NET libraries for the PC part.

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.

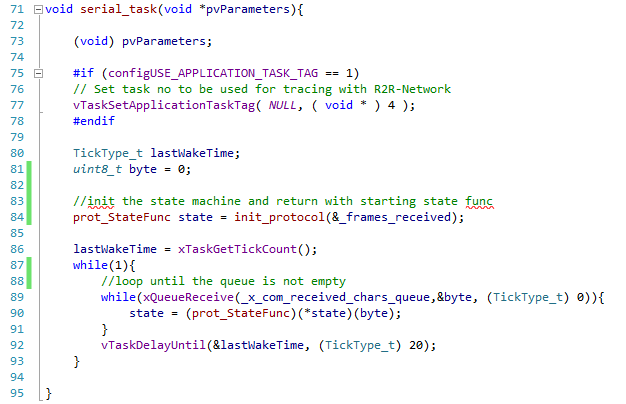


Here it can be seen how was the send method implemented. As it can be seen, every frame is tried maximum 5 times, after unsuccessfully try, the frame will be thrown away and the next one will be tried. This implementation is really simple, because both the Write() and ReadByte() method in System.IO.Ports.SerialPort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



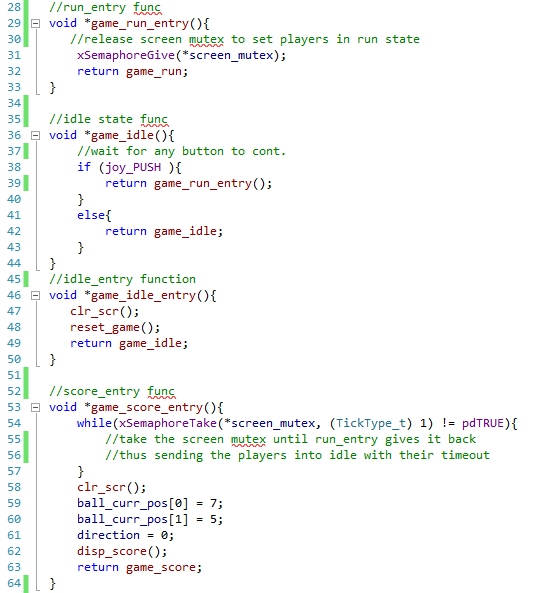
The serial task and game task implement their designed state machines with function pointers. Their FreeRTOS tasks initialize the state machine and variables, then they enter where they call the state function then delay themselves using vTaskDelayUntil(). The vTaskDelayUntil() was chosen over the simpler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

