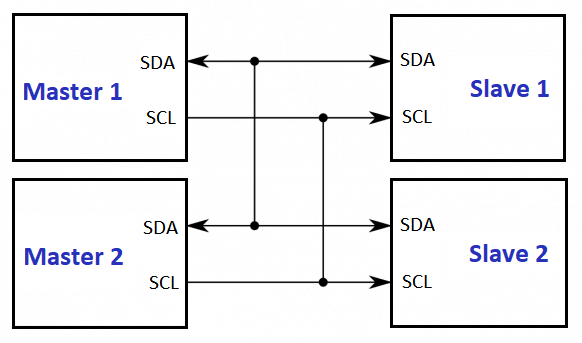
**I2C multi-master**



Marijn Verschuren 12-05-2023

# Table of contents

[1. Table of contents 2](#_Toc1)

[2. Table of figures 2](#_Toc2)

[3. Introduction 3](#_Toc3)

[4. Design 3](#_Toc4)

[5. Testing 6](#_Toc5)

[6. Reflection 6](#_Toc6)

# Table of figures

[Figure 1. Sequence diagram 3](#_Toc1)

[Figure 2. Game state packet 4](#_Toc2)

[Figure 3. Tetris block data 4](#_Toc3)

[Figure 4. Token receive function 4](#_Toc4)

[Figure 5. Token send function 4](#_Toc5)

[Figure 6. Main loop 5](#_Toc6)

[Figure 7. Collision detection 5](#_Toc7)

[Figure 8. Logic analizer I2C reading 1 6](#_Toc8)

[Figure 9. Logic analizer I2C reading 2 6](#_Toc9)

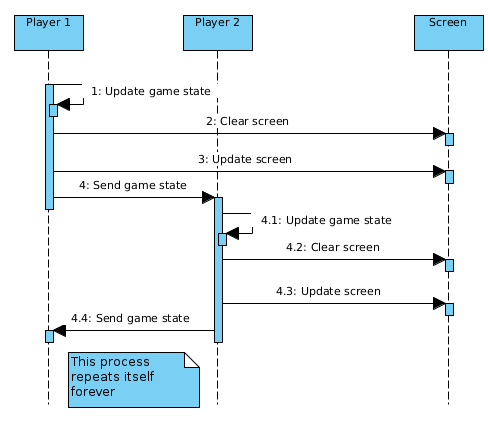
# 3. Introduction

For this project I have chosen to make Tetris, this is a very simple game where blocks are dropped and the player is tasked to fill in rows in the grid which when filled are removed. The main challenge in this project was the fact that both players (micro-controllers) share a single screen, this screen is controlled via a I2C bus that is shared among the players. The issue with a single I2C bus is that only one micro-controller can write to it at the time, this is only made worst by the fact that the screen takes a 400 byte screen buffer so updating the screen takes a significant amount of time. If this issue is not properly dealt with the screen would not show anything useful.

# 4. Design

Bus arbitrage (two micro-controllers writing to the bus at once) is already detected by the Arduino library. Once the micro-controller notices that what it is sending is not actually appearing on the bus it will stop sending, this will however mess up the data that the other master was sending. This is why another level of arbitrage prevention is added, I have done this by sending a ‘token’ back and forth, only the micro-controller with the token is allowed to write to the bus. In this application no actual token is sent, I simply send the game state over to the other micro-controller which sets the token flag once it receives the game state. Sending the game state was the next problem because an instance of the game is way bigger than 16 bytes, this is an issue because the Arduino library only allows you to send 16 bytes of data in one go. This would mean that I have to send my game instance in pieces making it incredibly hard to receive and put together on the other micro-controller. I solved this by only sending the block that is currently moving, this does however open up possibilities for desynchronisation but that is a fair tradeoff for simpler code (also avoiding possible receive errors). In Figure 1 the process is shown as a sequence diagram.

Figure 1. Sequence diagram



In Figure 2 the packet that is sent between micro-controllers is shown, It includes an instance of a Tetris\_Block, the data inside it is shown in Figure 3. Each micro-controller has two game instances, its own which it updates itself and one for the other micro-controller which it only updates with the received game state. When a micro-controller receives such a packet it updates the block at the index in the other game state. The new game flag is used to synchronize all game instances once one of the micro-controllers is reset.

Figure 2. Game state packet

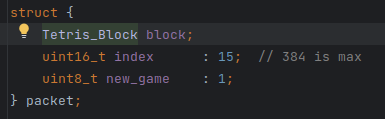
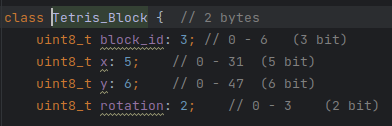


Figure 3. Tetris block data



In the Figure 4 you can see the receive code where the ‘PEER\_GAME’ is updated and the token flag ‘I2C\_master’ is set. And when the micro-controller is done updating its own game and drawing both games on the screen it will run the send function (Figure 5) where it will send its own game and reset the token flag.

Figure 4. Token receive function

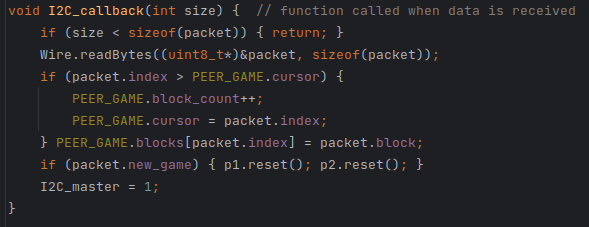
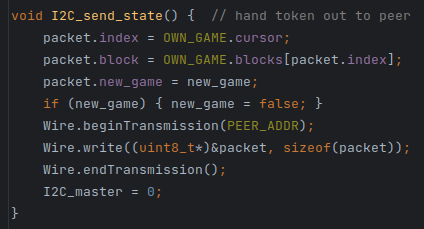


Figure 5. Token send function



In Figure 6 you can see the main code where the micro-controller waits until it is master or until the timeout has passed. After this the debug led is toggled and the screen and screen buffer are cleared. Then the games are drawn, this is done before updating because it allows me to look in the screen buffer when detecting collisions (see Figure 7). After that the moving blocks are drawn and the screen buffer is written to the screen, then finally the game state (token) is sent to the other micro-controller making the current micro-controller wait.

Figure 6. Main loop

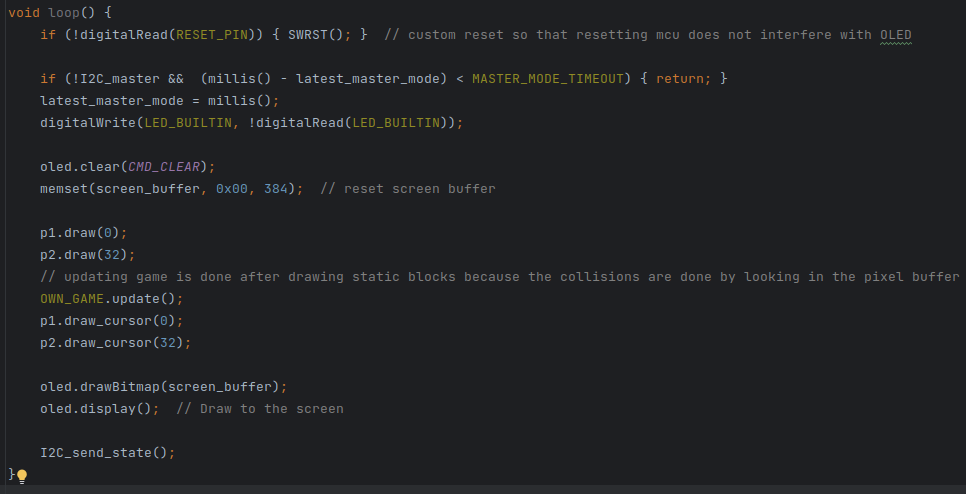
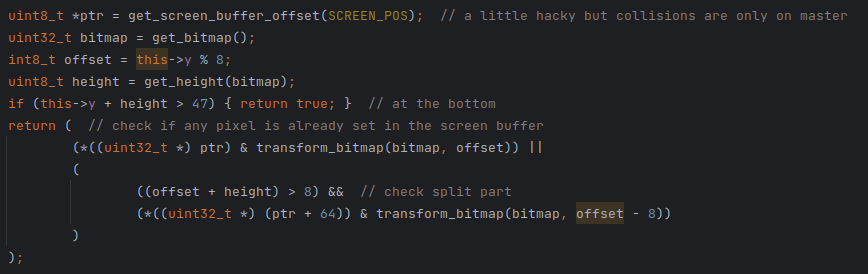


Figure 7. Collision detection



# 5. Testing

To test this application we, for one used the built in led to show which micro-controller is the master so I can check that there is only one master at once. And secondly by using the *Analog Discovery 2* to look at signals sent over the I2C bus. The last method I used was looking if the output on the screen is correct. This is a video showing the debug leds and the screen working correctly: <https://youtu.be/nPfHTobw4es>.

In Figure 8 and 9 you can see the packets sent over I2C captured by the *Analog Discovery 2*. The packet in Figure 8 for example holds:

block\_id = 2  
block\_x = 10  
block\_y = 25  
block\_rot = 2  
index = 16  
new\_game = 0

Which checks out with what can be seen on screen

Figure 8. Logic analizer I2C reading 1

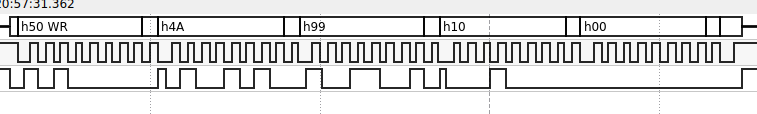
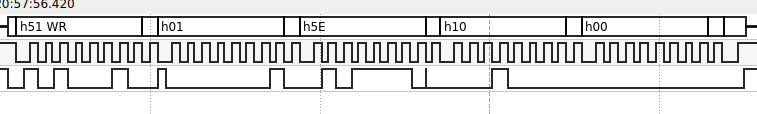


Figure 9. Logic analizer I2C reading 2



# 6. Reflection

During this project I learned the hard way that the STM32F4 is really difficult to program to receive in I2C slave mode. That is why I ended up using two Arduino’s for this project. Furthermore I have learned about arbitrage prevention methods such as the token method that was used. I also came across the limitations of the Ardiono’s and its libraries that were used in this project like running out of memory and being limited to 16 byte transfers over I2C.