Department of Information
Engineering and Computer Science

Embedded Software for the Internet of Things
2020/2021

TETRIS

Pietro Farina

Problem Statement

History of the game

Tetris is a video game created by Russian software engineer Alex Pajitnov in 1984. The game was not immediately patented, which contributed to its rapid diffusion. After a significant period of publication by Nintendo, the rights reverted to Pajitnov in 1996, who co-founded "The Tetris Company" to manage licensing.

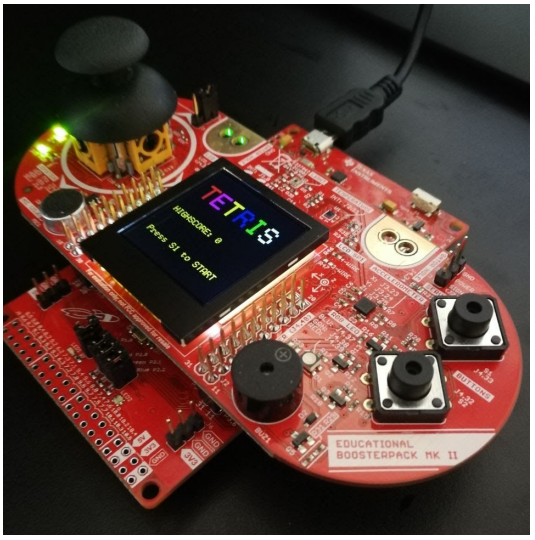
The game is available on over 65 platforms, setting a record for the most ported video games. My idea was to add another platform to the already long list of reached ones.

The rules of the game

The game is composed of a field of play in which pieces of different forms descend from the top of the field. The pieces have different shapes and are composed of four rectangles. During the descent of the pieces, the player can move and rotate them, the player can neither slow down the falling pieces nor stop them, but can accelerate them. The objective of the game is to use the pieces to create as many horizontal lines of blocks as possible. When a line is completed, it disappears and all the above blocks fall down. The player obtains points during the descendent of the pieces and after completing horizontal lines. The game ends when a new piece cannot be placed.

Tetris on MSP432

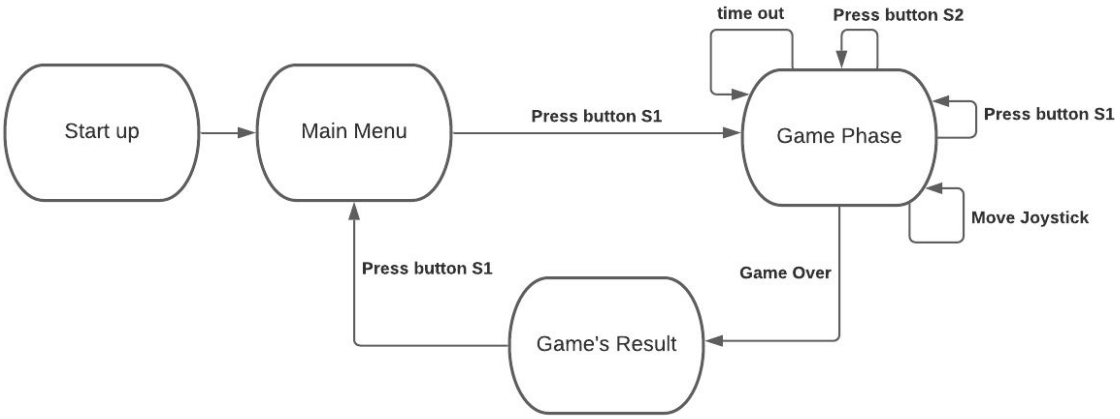
I decided to develop the "Tetris" game for the MSP432 Launchpad paired with the Educational BoosterPack MKII. In order to do it I had to meet different restrictions of the platform. The size of the display was not a constraint I could ignore, in fact the amount of possible information displayed was widely reduced.



The main difference between the version I developed and the original one is that the field is more small, and some shapes are composed of three rectangles. Lastly, while playing it is not possible to see the actual points obtained and the following piece.

Basic Working Scheme

The system is developed for the platform MSP432 paired with the Educational Boosterpack MKII. It uses the following peripherals: joystick, button S1 and S2, LCD.



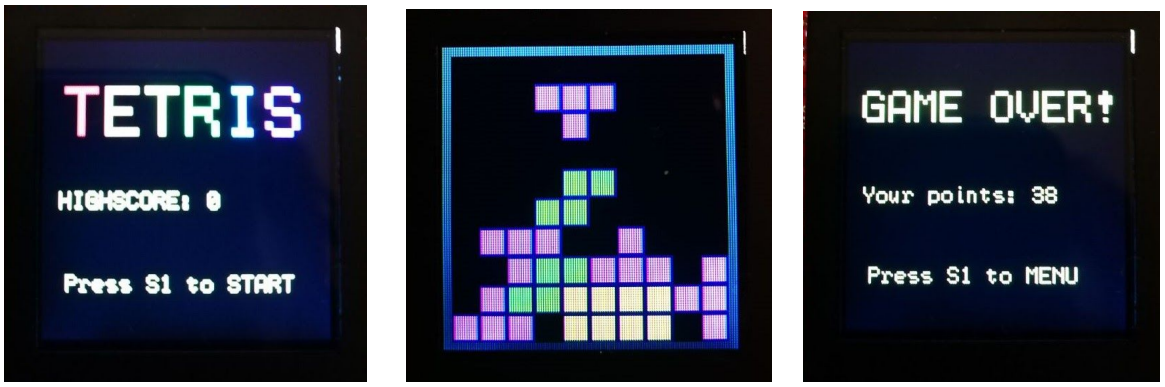
The system has three pages:

- **Main menu:** it introduces the user to the game and shows the highest score reached;
- **Game page:** the user can play to the game and try to beat his record;
- **Result page:** it appears when the user loses and it shows the score obtained.

When the system starts up the **main menu** page will appear on the screen, in this page the user will be introduced to the game and can see the highest score reached (at the start 0).

If the user presses the button S1 the **game page** is loaded and the user can start playing. Now he can interact with the system with the joystick and the buttons, in particular when moving the joystick in one direction, the falling piece will move towards it, when pressing the button S1 the falling piece will rotate and, finally, when pressing the button S2 the piece will fastly reach the ground. When the falling piece touches the ground, the points are evaluated and a new piece will appear, if there is no space for the new piece the game ends and the result page is loaded.

In the **result page** the user can see the points he did while playing and can choose to return to the main menu by pressing the button S1.



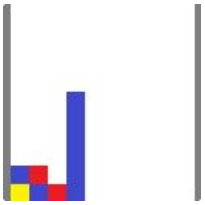
(In order, the main menu, the game page and the result page. Unfortunately the colours in the page are not clearly visible in the photo)

Software Architecture

The software is written in C and is supported by the libraries of DriverLib and Energia. The Energia TI environment provides benefits for the graphic and the sensor sections of the code, but impose limits on the structure and the available resources as well.

Software Design

The first thing to do was to check the limits of the target, to understand them I drawed a bitmap and loaded it on the platform. That gave me an idea of the capacity of the platform and the possible size of the game.



Bitmap used to check the size of the screen

Then I started to design the logic structure of the game and its mechanics. The game was designed from zero without external reference, knowing only what should be the final result. After defining the workflow of the game and the support structure I wrote the logic of the game while testing it on a friendly environment.

Once the code relative to the game was finished, the next step was to run it on the platform. Energia sketches must have two fundamental functions: setup() and loop(). The first function is called as the system starts while the second runs repeatedly. I integrated the code of the game with the functions needed by Energia and the other resources useful for playing on the platform.

The main resources used are:

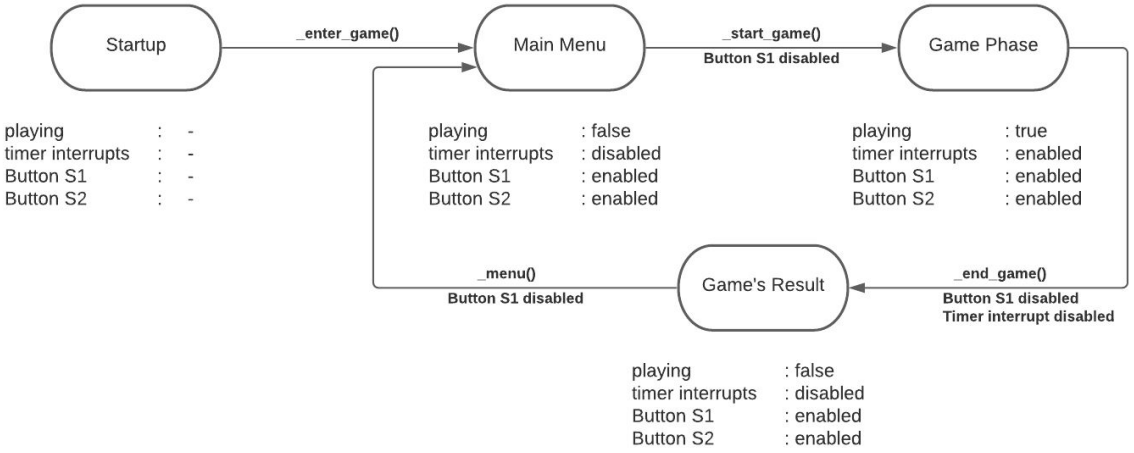
- Joystick: sampled with polling in the loop() function of Energia;
- Button S1, button S2: attached to interrupts;
- TIMER A3: this timer was not used by the libraries of Energia and could be used to generate periodic interrupt;
- LCD: used with the libraries of Energia.

Software structure

The software is structured in different files:

- tetris_energia.ino: contains the setup() and loop() functions, it defines all the interrupts and the usage of the different sensors except the LCD.
- tetris.h: contains all the logic of the game.
- graphic.h: define the usage of the LCD.
- utils.h: contains some useful functions used in different files.

In tetris_energia.ino are defined the variables used to distinguish the flow of the system and to enable or disable the interrupts of the buttons and the timer.



During the function that switches between the states of the program, the button S1 is disabled to avoid change of states while already changing, which could result in system failure.

The button S2 is always enabled because it works only in the game phase. Less important variables are omitted by the graph, and are used from the button S1 to understand when to load the menu and when the game.

The loop() function of Energia is always running, but it is used only when playing to sample the values of the joystick and when the variable playing becomes false it calls the _end_game() procedure.

Interrupts: sensors

The Joystick values are sampled in every iteration of the loop() function when in the game phase. The x-axis and y-axis values obtained are mapped from [0, 4096] to [1, 126]. If the values of the x or y reach a certain threshold it will move the falling piece in the corresponding direction.

- if (x > 110) move the piece to the right,
- if (x < 18) move the piece to the left,
- if (y > 110) move the piece down.

Before actually moving the piece, the other interrupts and sensors are disabled to avoid that another movement request blocks the previous one. This can cause several problems, e.g., some colours are not cleared from the screen or there are invisible blocks.

The other two sensors used are the two buttons S1 and S2, the first is used in game to send the request of rotating the falling piece and outside the game to move between the different pages of the application. The second one, instead, if pressed during the game it requests the system to make the falling piece fastly reach the ground. The two buttons are attached to two different interrupts handlers which implement the functionalities previously explained.

One problem found with the sensors was to limit the stream of new values from the joystick, in fact it was enough that the joystick was inclined for a fraction of seconds and many

requests of movement were forwarded to the system. It was necessary to send the movement request only when the user really wants and to solve this problem, the system “freezes” the sensors after using it, the sensors become available again after a period of time.

Interrupts: timer

The time, in which a sensor cannot be used, is counted using the timer, due the fact that Energia uses timers for its functions, there was only one available, so it has to be shared between the different sensors and the other requests too (periodic descent of the piece).

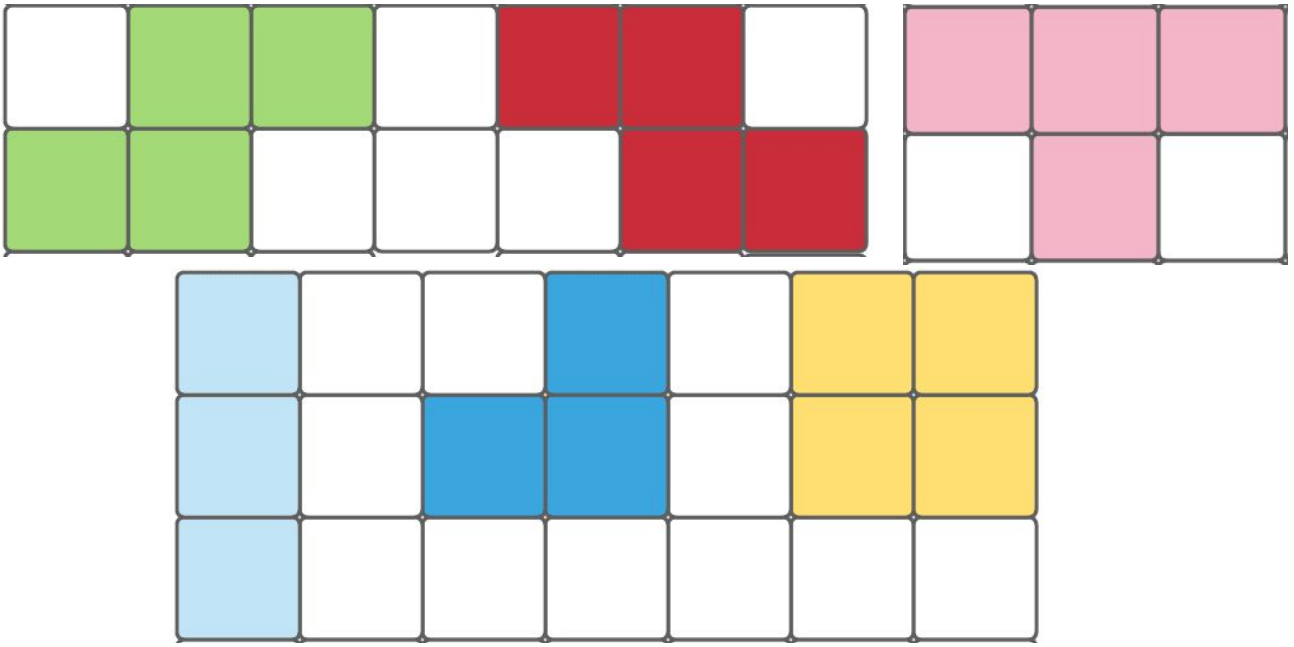
```
void pressed_S2() {
  if (playing && !freezed_S2) {
    TA3CCTL0 = ~BIT4;
    playing = choke_Piece(&mappa);
    TA3CCTL0 = BIT4;

    // freeze it
    freezed_S2 = true;
    unfreeze_S2 = (count + 2) % 4;
  }
}
```

To manage all needs the TIMER A3 increments a variable count at every interrupt, the count starts from zero to three and then restarts. The system, after using a sensor, freezes it until count is increased two times. One time was not enough because the user could press the button just before the timer interrupt.
When the count reaches a certain value it sends the request to move the piece downward.

The timer uses the ACLK (32 kHz) as clock source and it will send an interrupt every ⅓ seconds. The request of movement is sended every 1 second to leave time to the users to move the piece.

The Game: data structure

The playable field is a matrix containing different blocks. Due to the constraints of the screen I decided to represent the game as a 10 x 10 matrix. Each piece is composed of three or four blocks and it depends on their shape.



To distinguish the case between a piece who is falling, and one who already touched the ground I will use different terms, I will call the first as the “falling piece”, while the other “dead piece”. When the falling piece touches the ground it becomes a death piece and another falling piece will be created. The piece has an array where it keeps the different occupied cells, the array has a fixed size of 4.

To represent the map I use two matrices 10 x 10, one to save the id of the piece, which I needed to understand when a cell is empty, occupied by the falling piece or another piece, and one containing the different colours. When a piece occupies a new cell, it will change the id and the colour of the cells with its own values and reset the values of the previous occupied cell. The id on the map have different meanings:

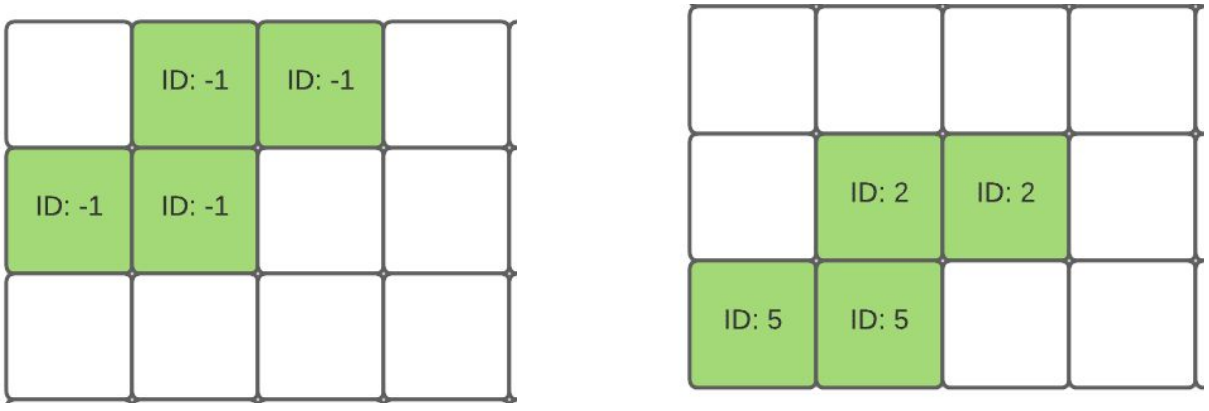
- 0 : the cell is empty;
- -1 : the cell is occupied by the falling piece;
- > 0 : the cell is occupied by a dead piece.

The matrix visible to the user is the one with the colours.
An array keeps track of the row population, when one row is full it means it can be deleted.
A circular buffer is used to maintain the available id, when a falling piece “dies” the system assigns a new id to the piece, instead when one row is deleted, all the id contained are released and returned to the buffer. The range of ID available is [1, 100].

I use other variables to keep other information useful to maintain the points obtained, to rotate the falling piece, or to simplify the algorithm used.

How to assign the ID

When a falling piece collides with a block below or touches the ground it "dies"and receives an ID. The ID is not given to the piece itself but to the different cells of the matrix it occupies. In particular every cell occupied receives an ID in base on their y-axis (which is the row index), that means if two cells of the same piece have the same y, then they will receive the same ID. That is because when later in the game a row is deleted, the above coloured blocks will fall, during this fall they maintain the original structure and they do not break up.



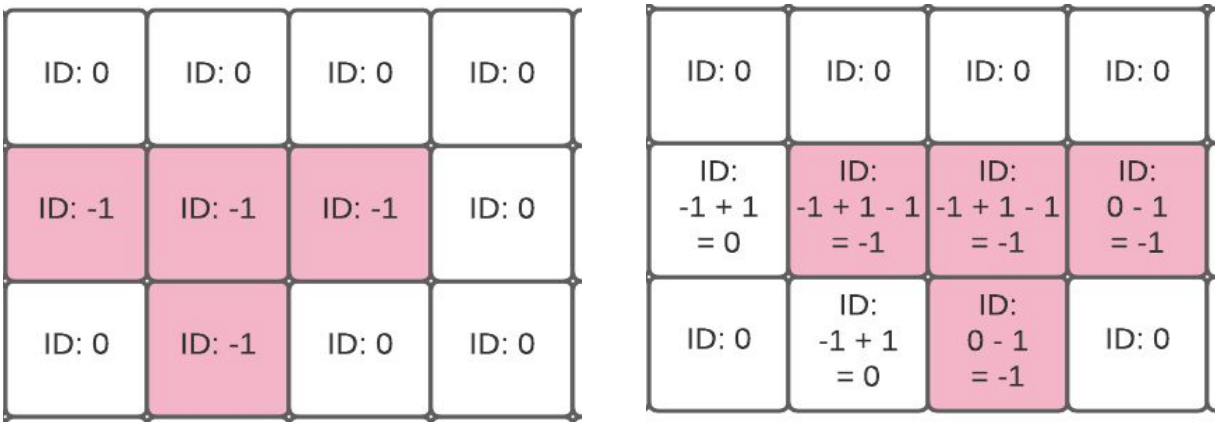
After the piece hits the ground, the system assign an ID to their cell based on the y axis.

Movement of a Piece

The falling piece can move to the left, to the right and downward. After declaring the direction the piece will iterate on its cells to understand if it has space to move. To know if it has the available space it checks the id_matrix of the map, if the target cells have 0 it can move.

If it can move every cell, it will decrease 1 in the id_matrix the new occupied cells and add 1 to the left one. This way when moving the piece, it will not overwrite itself and after the movement, the cell empty will have 0 as id and the cell occupied by itself will have 1.
Now the system will check the id of the displacement region, when it finds a -1 cell it will paint with the colour of the falling piece, when it finds a 0 cell it will paint with the background colour, in our case black (0x0000).

If there is no space to move and the direction is right or left, it does not move anything, but if the direction is down, the system kills the piece, assigns the new id to the different cells and checks for the complete row.



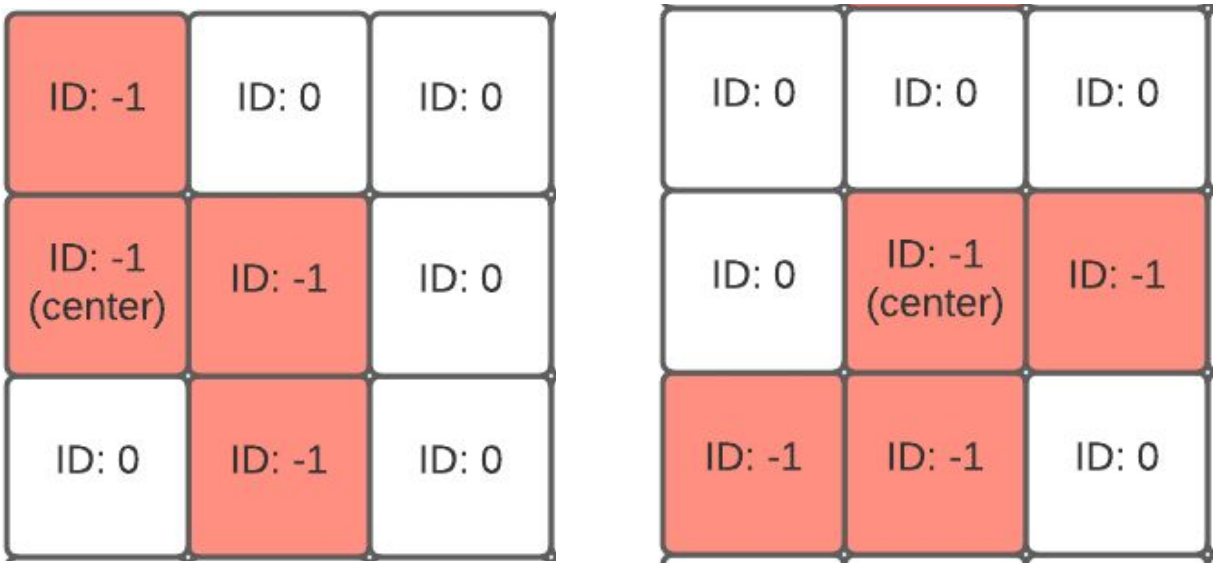
The piece, after checking if the cells are empty, starts moving while changing the ID of the cells

Rotation

The rotation of a piece is a necessary feature to the game, but it depends on different cases, in particular to the shape of the piece, to its position, to the previous rotation and the possibility to move in order to rotate. In fact, if the rotation in place is not possible, the game could assign a lateral movement to perform it. I implemented the standard case of the rotation in place and the sliding rotation when the piece is near the edge.

One cell of the falling piece is defined as the center, the center will work as a rotation axis. With some shape the center does not move while rotating, but in other cases it can.

The code will check the situation of the piece and what are the target cells of the rotation, if empty it will change the cells occupied by the piece, modify the center if needed and increase the number of rotations. After changing the id matrix of the map, the system will modify the colour matrix of the map too.



The piece, cannot rotate in place, so it will performs a movement on the right and then rotate

Falling of a Row

When a horizontal line is completed it is deleted and all the above rows will start falling down. After all the above rows have fallen down, is checked if another line is complete. These two phases are repeated until nothing moved for one iteration.

When the block falls they maintain their structure, which is checked thanks to the ID saved previously. Maintaining the structure means that two blocks with the same id will fall of the same number of space (in terms of lines or rows). e.g., if one block has only an empty cell under it and the other has three empty cells, both will fall of one “row”.

The algorithm, which manages the fall of a row scrolls for the row, when found a block (ID > 0) it sees how much it can fall by checking the under cell. When finding another block, if it has the same ID of the previous it checks the empty cell for at most the max number found before, otherwise it checks all the above cells until it finds no empty one and leaves all the previous blocks with the same ID falling down for the maximum amount found before.

```
void fall_row(Map* map, int8_t y) {
  uint16_t colour;
  int8_t j, max_so_far, previous_id = -1;
  int8_t bond[MAX_COLUMNS];
  for (int i = 0; i < MAX_COLUMNS; ++i) {
    if (map->int_matrix[y][i] != 0) {
      if (map->int_matrix[y][i] == previous_id) {
        bond[i] = bond[i - 1];
        for (j = y + 1; j <= max_so_far; ++j) {
          if (map->int_matrix[j][i] != 0)
            max_so_far = j - 1;
        }
      } else {
        bond[i] = i;
        previous_id = map->int_matrix[y][i];
        max_so_far = MAX_COLUMNS - 1;
        for (int j = y + 1; j <= max_so_far; ++j) {
          if (map->int_matrix[j][i] != 0)
            max_so_far = j - 1;
        }
      }
      if ((max_so_far > y) && ((i + 1 == MAX_COLUMNS) || (previous_id != map->int_matrix[y][i + 1])))

        for (j = bond[i]; j <= i; j++) {
          colour = map->char_matrix[y][j];
          map->int_matrix[max_so_far][j] = previous_id;
          map->char_matrix[max_so_far][j] = colour;
          draw_Tile(j, max_so_far, colour);
          map->row_occupation[max_so_far]++;

          map->int_matrix[y][j] = 0;
          map->char_matrix[y][j] = 0x0000;
          draw_Tile(j, y, 0x0000);
          map->row_occupation[y]--;
          //  UPDATE PIXELS
        }

      }
    }
  }
}
```

Code responsible for making fall down a row

Efficiency of the display

To avoid a continuous refresh of the display, instead of updating the visible map after the change to its logical structure, I decided to send the request of changing the colour of some portion of the screen, directly in the logic game code. With the cost of having direct hardware dependent calls into the hardware independent code I have highly improved the game experience. Otherwise, the refreshing of the complete map was expensive in terms of time and resources consuming, and the gameplay was not smooth and enjoyable.

```
if (map->int_matrix[y][x] != 0) {
  nospace = true;
} else {
  colour = map->moving_Piece.colour;
  map->int_matrix[y][x] = -1;
  map->char_matrix[y][x] = colour;
  //  UPDATE PIXELS
  draw_Tile(x, y, colour);
}
```

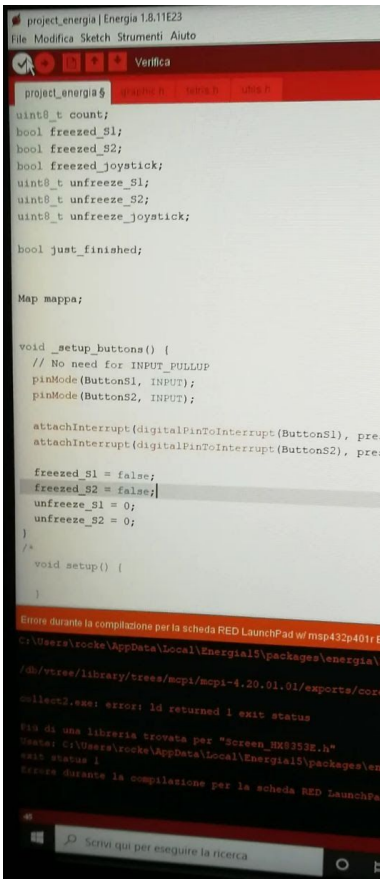
Testing

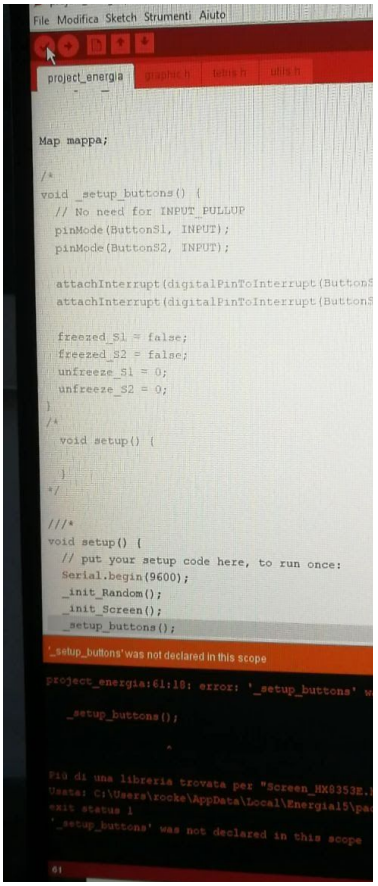
At the start I decided to use the scaffold test to test the hardware independent code. But I found difficulties once I moved to the Energia environment due to the file inclusion which is different from the normal environment. In fact, if using file.c and file.h and including the file.h, the file.c was not included in the main file by Energia and there was no possibility to add manually, that causes different types of errors. So, I decided to write all the code in the different files .h and to avoid writing the .c.

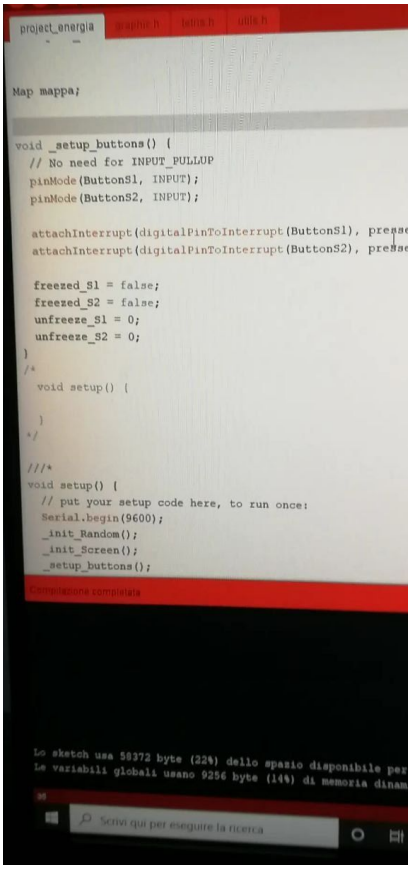
The game logic was tested in a neutral environment (host) and after the porting on the msp432 platform I started programming and testing the hardware dependent code, from the code relative to the LCD and lastly to the interrupts, timers and change of states.

I found a strange behaviour on the Energia IDE, where after a restart of my laptop, the IDE could not find the reference to the main when compiling my project. I resolved the problem by commenting different portions of the codes and writing an empty setup() function, then Energia will find an undefined reference to the functions removed, but the previous error did not appear anymore, after I removed the comments I tried to compile and the project was compiled without errors.

I registered a video about the error and the way I temporary resolved, here are some snapshots:







Conclusions and Future work

I am very satisfied with the work I have done and the progress I have made. It was the first project I did exclusively by myself and the result satisfies my expectations. I have never programmed on a micro controller and it was an interesting experience. I have already heard from colleagues of Arduino and different boards and this was the opportunity to deepen the topic.

I would like to make small improvements to the code, for example in the function which rotates the piece, and other optimization. Furthermore I would like to adjust the structure of the module using the directives to increase the platform compatibility and make the code more clear.
As for the game, I want to add more difficulties with pieces that fall at different speed rates, and provide different game mode as the original game does.